UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER THESIS IN COMPUTER ENGINEERING

Engineering a RAG Chatbot for Technical Manual Navigation through Vector Search and Cloud LLM Integration

MASTER CANDIDATE

Jacopo Righetto

Student ID 2095325

SUPERVISOR

Prof. Nanni Loris

University of Padova

ACADEMIC YEAR
2024/2025

14/10/2025

To my family
and friends.
This is thanks to you.

Abstract

This thesis presents the design and development of an intelligent chatbot system built on the Retrieval-Augmented Generation (RAG) paradigm, with the primary goal of facilitating the navigation and consultation of technical manuals. The system integrates a vector-based retrieval engine, powered by ChromaDB, with a cloud-hosted large language model (LLM) accessed through the Gemini API. Users can interact with the chatbot in natural language, receiving contextually relevant and technically accurate responses drawn directly from domain-specific PDF manuals. The architecture follows a modular and scalable clientserver approach: the backend, implemented in Python with FastAPI, handles document ingestion, semantic chunking, and vector search, while the LLM enriches retrieved passages with generative reasoning to produce coherent and user-friendly answers. On the client side, a Flutter-based application running on a tablet serves as an interactive front-end, ensuring usability in real-world industrial contexts. Persistent memory and efficient retrieval strategies ensure that responses remain accurate even as the document base grows. Beyond implementation, the thesis emphasizes practical concerns such as latency, device limitations, and internet dependency, which are critical when deploying AI systems on constrained hardware in field environments. Experimental evaluations were conducted to measure retrieval accuracy, response quality, and overall user experience. Results demonstrate that the proposed architecture successfully balances performance, scalability, and usability, highlighting the potential of RAG-driven chatbots to support industrial documentation access, training, and decision-making in real-world deployment scenarios.

Contents

Acronyms

AI Artificial Intelligence. 6

ANN Approximate Nearest Neighbour. 17

AOT ahead-of-time. 23

API Application Programming Interface. 2

ARM Advanced RISC Machine. 23

BERT Bidirectional Encoder Representations from Transformers. 6

FAISS Facebook AI Similarity Search. 19

GPT Generative Pre-trained Transformer. 8

HNSW Hierarchical Navigable Small World. 17

HTML Hypertext Markup Language. 24

IVF Inverted File Indexes. 19

JIT just-in-time. 23

LLM Large Language Model. 2

LSTM Long Short-Term Memory. 6

MVP Minimum Viable Product. 24

NER Named Entity Recognition. 11

NLG Natural Language Generation. 9

NLM Neural Language Model. 6

NLP Natural Language Processing. 1, 5

NLU Natural Language Understanding. 7

OCR Optical Character Recognition. 12

OEM Original Equipment Manufacturer. 23

PLM Pre-trained Language Model. 6

POS Part-of-Speech. 11

PQ Product Quantization. 19

RAG Retrieval Augmented Generation. 2

SA Sentiment Analysis. 20

SDK Software Development Kit. 23

SLM Statistical Language Model. 5

SQL Structured Query Language. 10

VDB Vector Database. 16

1 Introduction

1.1 Background and Motivation

In recent years, artificial intelligence (AI) and advances in Natural Language Processing (NLP) have significantly changed the way people interact with digital systems. Conversational interfaces, ranging from customer service chatbots to intelligent personal assistants, are gradually replacing traditional search-based systems thanks to their accessibility and ease of use. This evolution is driven by a growing expectation that digital tools should not only provide information but also understand context, adapt to user needs, and respond in real time.

At the same time, businesses across different sectors are under increasing pressure to provide more innovative, more user-friendly ways of accessing information. The retail and hardware industries, in particular, often rely on extensive product catalogues, manuals, and technical specifications that remain scattered, difficult to navigate, or still bound to paper-based formats. Customers in hardware stores, for example, frequently depend on specialised staff to obtain product details or usage recommendations. While this model has traditionally worked, it shows its limitations when staff availability is reduced, product information is fragmented across different sources, and customer expectations demand instant and precise assistance.

The challenge, therefore, lies in bridging the gap between vast amounts of technical documentation and the customers need for quick, reliable, and comprehensible answers. Intelligent conversational assistants represent a promising approach to this problem, combining natural language understanding with domain-specific knowledge retrieval.

1.2 Objectives of the Thesis

The main objective of this thesis is to design and implement a conversational assistant that can support customers in retrieving information about products directly from technical manuals and catalogues. The assistant is envisioned as an interactive kiosk, based on a tablet device, deployed inside a hardware store. Its purpose is to allow customers to ask natural language questions and receive fast, accurate, and context-aware answers without requiring direct staff intervention.

Concretely, the system aims to:

- Understand natural language queries posed by customers in everyday language;
- Retrieve relevant and precise information from technical product documentation;
- Operate efficiently on a tablet-based interface, providing a seamless user experience within hardware limitations;
- Combine structured document retrieval with the generative reasoning of a large language model for robust fallback and clarification.

The broader objective of the thesis is to show how state-of-the-art NLP methods can be adapted and optimised for resource-constrained, real-world environments such as in-store kiosks.

1.3 Overview of the Solution

To meet these objectives, we developed a hybrid architecture that integrates both retrieval and generative components. The system combines a Flutter-based frontend, designed for a tablet device, with a backend implemented in Python and FastAPI. At its core, the assistant adopts the Retrieval Augmented Generation (RAG) paradigm, which unites the efficiency of vector search with the generative power of large language models.

Product manuals and catalogues, typically available in PDF format, are pre-processed through a custom pipeline. Text and tables are extracted, irrelevant visual elements are filtered out, and the resulting documents are segmented into semantically meaningful chunks. These chunks are embedded and stored in a ChromaDB vector database, enabling efficient semantic retrieval. When a user poses a query, the system identifies the most relevant content chunks through vector similarity search. These are then passed to the Gemini 2.0 Flash Large Language Model (LLM) via an Application Programming Interface (API), which generates a coherent, context-aware response tailored to the query.

Particular attention was given to optimising the processing of technical PDFs, ensuring that retrieved content is both accurate and valuable. The overall design emphasises

responsiveness and lightweight operation, allowing the assistant to run smoothly on commercial tablets while maintaining usability in real-world store deployments.

1.4 CONTRIBUTIONS

This project was carried out collaboratively by a team of four engineering interns. Within this context, the thesis makes the following contributions:

- **System design and architecture:** Development of a modular, scalable architecture that integrates a large language model, a vector database, and an interactive frontend.
- **Implementation of a RAG-based assistant:** Integration of Gemini 2.0 Flash with ChromaDB and LangChain, enabling conversational access to domain-specific documents.
- **Custom document processing pipeline:** Extraction and refinement of content from technical PDF manuals, including text and tables, with targeted filtering of irrelevant elements.
- **On-device deployment:** Adaptation of the solution for an Android tablet environment using Flutter, ensuring responsiveness despite hardware limitations.
- **Evaluation of performance and usability:** Preliminary assessment of system latency, response quality, and user experience in the context of a retail scenario.

In summary, this thesis provides both a proof-of-concept and a reference framework for deploying intelligent conversational assistants in physical environments where quick access to technical information is essential and conventional search interfaces prove inadequate.

2

Literature Review

2.1 Evolution of Language Models: from SLMs to LLMs

The development of language models has been central to the evolution of artificial intelligence systems, particularly in NLP. This evolution has progressed through four key stages, each building upon the capabilities and limitations of its predecessors: *Statistical Language Models (SLMs)*, *Neural Language Models (NLMs)*, *Pre-trained Language Models (PLMs)*, and *Large Language Models (LLMs)*. Understanding this progression is essential to contextualizing the capabilities of modern virtual assistants and chatbots [27].

2.1.1 Statistical Language Models (SLMs)

Statistical Language Model (SLM) represents the earliest formal approach to modelling language computationally. Rooted in probabilistic methods, SLMs model the likelihood of word sequences using techniques like *n-gram models*, based on the Markov assumption that a word's probability depends only on a fixed number of preceding words [17].

These models were widely adopted in early applications such as information retrieval and speech recognition. However, their major limitation lies in their reliance on sparse, discrete probability distributions, which struggle to generalize from limited training data and suffer from the so-called *curse of dimensionality*.

Smoothing techniques, such as back-off and GoodTuring estimation, were introduced to mitigate these issues but offered only incremental improvements.

2.1.2 Neural Language Models (NLMs)

To overcome the limitations of SLMs, Neural Language Model (NLM) emerged as a paradigm shift in the early 2010s. These models leveraged neural networks, such as multilayer perceptrons and recurrent neural networks, to learn distributed representations of words by embedding them in continuous vector spaces.

This allowed models to generalize better and handle much larger vocabularies with improved contextual understanding. A pivotal advancement came with *word2vec*, which introduced efficient training methods for learning word embeddings, significantly boosting performance across a wide range of NLP tasks.

NLMs laid the groundwork for the deep learning era in language processing, enabling the development of task-agnostic representations and moving the field toward more generalizable, end-to-end learning architectures.

2.1.3 Pre-trained Language Models (PLMs)

The next significant milestone was the introduction of Pre-trained Language Model (PLM), which decoupled the learning of general-purpose language representations from task-specific fine-tuning. The first notable implementation, *ELMo*, demonstrated that contextualized word representations learned through bidirectional Long Short-Term Memory (LSTM) could significantly enhance downstream task performance.

This approach was later extended and refined with Transformer-based architectures, leading to models like *Bidirectional Encoder Representations from Transformers (BERT)* and *RoBERTa*. These models were trained on massive corpora using self-supervised objectives (e.g., masked language modelling), and subsequently fine-tuned on specific tasks, achieving state-of-the-art results in question answering, sentiment analysis, and other NLP domains.

The *pre-train then fine-tune* paradigm became standard practice, significantly reducing the need for task-specific data and manual feature engineering [27].

2.1.4 Large Language Models (LLMs)

Building upon PLMs, the latest generation of models, known as *Large Language Models (LLMs)*, have introduced a new era in Artificial Intelligence (AI) language understanding and generation. These models, such as *Claude*, *Gemini*, and *GPT-4*, scale up the number of parameters into the tens or hundreds of billions.

As models grow in size and training data volume, they exhibit *emergent abilities* that are not present in smaller models. These include *in-context learning*, *zero-shot*

reasoning, and *task generalization* without explicit fine-tuning.

LLMs are typically accessed via prompt-based interfaces, where users provide examples or instructions in natural language to elicit specific behaviours. This shift has democratized the use of powerful language models but also introduced new challenges in prompt design and model interpretability.

Unlike smaller PLMs, developing LLMs involves not only model design but also significant engineering efforts, ranging from large-scale distributed training and data curation to alignment with human preferences. The distinction between research and engineering has become increasingly blurred, as training LLMs requires extensive infrastructure and practical expertise.

The societal and industrial impact of LLMs is profound. Applications range from conversational agents and virtual assistants to code generation, multimodal reasoning, and enterprise automation. *ChatGPT*, based on the GPT-series models, exemplifies the leap from language modelling to general-purpose AI interaction [19].

2.2 Traditional Chatbots vs LLM-Based Systems

The evolution of chatbots has undergone a profound transformation over the past decades, moving from rule-based and intent-driven architectures to generative systems based on large-scale language models. While both approaches fall under the broader category of conversational AI, they differ significantly in terms of design philosophy, technological foundation, adaptability, and real-world applicability. This section examines the core characteristics of traditional chatbots and contrasts them with the capabilities of LLM-based systems, particularly in the context of modern virtual assistants [2].

2.2.1 Traditional Chatbots: Rule-Based and Intent-Based Architectures

Early chatbots were built using rule-based systems where each input was mapped to a corresponding output based on pattern matching or keyword detection. These systems often operated on static decision trees or finite-state machines, which made them predictable but rigid.

Over time, this evolved into *intent-based* chatbots, which attempt to classify user input into predefined intents and extract relevant entities through structured Natural Language Understanding (NLU) pipelines.

Intent-based chatbots typically consist of:

- **An intent classifier**, which maps user sentences to specific categories (e.g., "book appointment", "get weather");
- **Slot filling mechanisms**, which extract entities (e.g., date, location) needed to fulfil the intent;
- **Dialogue management modules**, which guide the conversation flow;
- **Predefined responses**, often templated or rule-generated.

Frameworks such as Rasa, Dialogflow, and the Microsoft Bot Framework have standardized this approach by offering platforms that abstract much of the low-level implementation. These systems are widely used in customer support, banking, retail, and enterprise virtual assistants due to their predictability, transparency, and ease of deployment within tightly scoped use cases.

However, their reliance on structured dialogue paths and static training data limits their ability to generalize to novel queries, deal with ambiguity, or engage in more naturalistic conversation. The need for manual configuration of intents and dialogues can also become a bottleneck as the complexity of the assistant increases [18].

2.2.2 Rise of LLM-Based Chatbots

The introduction of transformer architectures and the emergence of *Large Language Models (LLMs)* such as Generative Pre-trained Transformer (GPT), Claude, Gemini, and LLaMA have redefined the capabilities of conversational agents. Unlike traditional systems, LLM-based chatbots are trained on massive corpora of unstructured text and learn to model language in a generative, probabilistic manner.

These systems are not explicitly programmed with fixed intents or rules but instead generate responses based on learned representations of syntax, semantics, pragmatics, and context.

LLMs differ from traditional systems in several fundamental ways:

- They can process and generate open-ended responses across a wide range of domains without requiring prior intent definition;
- They maintain conversational memory, allowing them to understand multi-turn dialogue with improved contextual awareness;
- They support few-shot or zero-shot generalization, meaning they can often perform well on new tasks or questions with minimal training;
- Their language fluency and naturalness significantly surpass that of traditional intent-based models.

Modern virtual assistants powered by LLMs offer users more interactive and human-like experiences. They can summarize information, follow complex instructions, handle ambiguous queries, generate creative content, and even reason to some extent.

These capabilities have expanded the use cases of conversational agents far beyond what traditional architectures could support, ranging from personal productivity tools and customer-facing bots to assistants for software development, legal research, and healthcare triage [2].

2.2.3 Architectural and Operational Shifts

Traditional chatbot architectures are *modular and deterministic*, requiring careful orchestration of NLU, dialogue management, and Natural Language Generation (NLG) components. Their operation is largely transparent and controllable, which is ideal for compliance-driven environments or tasks requiring consistent behaviour.

In contrast, LLM-based systems are *monolithic and emergent*; they encode language understanding, memory, and generation into a single neural model. While this allows for greater flexibility and expressive capacity, it also introduces challenges in predictability, explainability, and error tracing.

Developers interact with these models primarily through *prompt engineering*, *fine-tuning*, or *Retrieval Augmented Generation (RAG)* techniques, which allow models to pull in external knowledge dynamically during inference.

This paradigm shift reduces the need for complex hand-coded logic but increases the importance of input formulation and system integration. For instance, in virtual assistants deployed in business environments, LLMs must often be grounded with domain-specific data sources to avoid producing hallucinated or irrelevant outputs [6].

2.2.4 Strengths and Limitations

While LLM-based systems bring substantial advancements, they are not without limitations. One of the main challenges is their tendency to *hallucinate*, generating content that sounds plausible but is factually incorrect or misleading. Additionally, they require significant computational resources, especially for real-time interactions or large-scale deployments.

Traditional chatbots, by contrast, are lighter and more controlled. Their behaviour can be fully audited and debugged, which makes them ideal for applications where strict adherence to rules or business logic is required. However, they often struggle with robustness, user engagement, and scalability, particularly when the scope of the assistant expands.

The choice between these approaches often depends on the use case:

- For structured, high-precision tasks with well-defined user flows, traditional chatbots remain effective and efficient;
- For more open-ended, exploratory, or context-rich applications, modern virtual assistants-LLM-based systems offer significant advantages.

2.3 Natural Language Processing (NLP): Foundations and Relevance in Chatbot Systems

Natural Language Processing is a critical area within Artificial Intelligence and Computational Linguistics, dedicated to the interaction between computers and human language. Its primary objective is to develop models and systems that can process, analyse, and generate natural language in a way that is both meaningful and contextually appropriate.

For chatbot systems, NLP forms the technological foundation that enables machines to understand user input and respond in human-like, conversational language.

Unlike traditional human-computer interfaces that require structured inputs (e.g., Structured Query Language (SQL) commands or shell scripts), NLP allows for communication using unstructured natural language, which may vary significantly in tone, syntax, semantics, and formality. This shift from rigid commands to more flexible natural interactions is especially beneficial in modern applications such as virtual assistants, customer service bots, healthcare triage systems, and e-commerce recommendation agents, where users may lack technical expertise or need rapid, intuitive interactions [1].

2.3.1 Historical and Theoretical Background

Although the term *Natural Language Processing* only emerged in the mid-20th century, the aspiration to communicate with machines using human language dates to the 1940s. Early research focused primarily on symbolic methods and rule-based systems, often grounded in formal logic and syntactic parsing.

The foundational work of Noam Chomsky (1965), particularly in the field of generative grammar and syntactic theory, provided the theoretical framework for many early computational language models. Chomsky introduced concepts such as *context-free grammars*, which laid the groundwork for parsing algorithms and later influenced programming language compilers and NLP tools alike [4].

Over the decades, the field transitioned through various paradigms, including:

- **Statistical NLP** (in the 1990s);
- **Machine learning-based approaches**;
- **Deep learning models** leveraging large-scale language representations, such as:
  - Word embeddings (e.g., *Word2Vec*, *GloVe*);
  - Transformer architectures (e.g., *BERT*, *GPT*, *T5*).

These advancements have significantly improved the ability of machines to model language context, ambiguity, and semantics-key challenges in chatbot development.

2.3.2 NLP IN THE CONTEXT OF CHATBOTS

In the design and implementation of chatbot systems, NLP is typically divided into two main components:

- **Natural Language Understanding (NLU)**
- **Natural Language Generation (NLG)**

NATURAL LANGUAGE UNDERSTANDING (NLU)

NLU refers to the process through which machines interpret and understand the structure and meaning of natural language input. It involves several sub-tasks that together allow a system to extract relevant information, determine user intent, and classify inputs based on context. NLU often relies on:

- **Tokenization**: splitting input text into words, phrases, or other units;
- **Part-of-Speech (POS) Tagging**: assigning grammatical categories (e.g., noun, verb, adjective) to each token;
- **Named Entity Recognition (NER)**: identifying and classifying named entities such as people, dates, products, or locations;
- **Dependency Parsing**: analysing grammatical structure and relationships between words in a sentence;
- **Intent Recognition**: mapping input to predefined categories or intents (e.g., “book flight”, “check weather”, “report issue”);
- **Slot Filling**: extracting key parameters associated with an intent (e.g., destination city, date, time).

NLU is particularly crucial for enabling chatbot systems to interpret user queries accurately, even when expressed in ambiguous or informal language [11].

NATURAL LANGUAGE GENERATION (NLG)

Natural Language Generation is the reverse process of NLU: it involves producing coherent, contextually appropriate responses based on internal data representations or structured inputs. In chatbots, NLG is used to convert intent-based outputs, retrieved knowledge, or database queries into human-readable replies. NLG systems typically operate in several stages:

1. **Content Determination**: identifying the specific information to be conveyed;
2. **Text Planning**: organizing information into a structured representation or outline;
3. **Sentence Planning**: determining grammatical structure and lexical choice;
4. **Surface Realization**: generating grammatically correct and fluent text.

NLG is especially important for ensuring that responses are not only correct but also engaging and natural-sounding.

2.3.3 KEY NLP TASKS RELEVANT TO CHATBOTS

Several NLP tasks are directly or indirectly integrated into chatbot systems to improve performance and user experience:

- **Automatic Summarization**: condensing long texts (e.g., FAQs, articles, support documents) into short, relevant summaries;
- **Co-reference Resolution**: determining when different expressions refer to the same entity (e.g., resolving "it" to "the package");
- **Discourse Analysis**: understanding the structure of multi-turn conversations and maintaining coherence across dialogue turns;
- **Machine Translation**: supporting multilingual interfaces by translating input and output between languages;
- **Morphological Analysis**: breaking down words into base forms and affixes to support languages with complex word formations;
- **Optical Character Recognition (OCR)**: enabling text extraction from scanned documents or user-uploaded images;
- **Text Classification**: assigning input to predefined categories (e.g., “technical issue,” “billing,” “product inquiry”).

These tasks often function as intermediate steps within a pipeline or are used in modular chatbot architectures. For instance, an OCR module may extract product codes from an image sent by a customer, while a classification model determines whether the user needs help with shipping or returns [11].

2.3.4 Linguistic Foundations of NLP

Linguistics provides a theoretical basis for understanding how language functions and how machines can simulate this understanding. Key linguistic levels relevant to NLP include:

- **Phonology**: though less relevant in text-based chatbots, it is essential in voice-based assistants;
- **Morphology**: critical for processing compound words and grammatical variations;
- **Syntax**: provides rules for sentence structure and parsing;
- **Semantics**: focuses on the meaning of words and phrases;
- **Pragmatics**: involves context-dependent interpretation, crucial for resolving ambiguity or implied meaning.

For chatbot applications, integrating semantic and pragmatic understanding allows systems to handle a wide variety of user inputs, including idiomatic expressions, implied requests, and context-dependent questions.

2.4 Retrieval-Augmented Generation (RAG): Theory and Use Cases

Large Language Models (LLMs) have transformed natural language processing by demonstrating remarkable capabilities in text generation, summarization, question answering, and other linguistic tasks. These models, trained on massive corpora of web data, exhibit strong performance in general-purpose applications and are capable of capturing complex linguistic patterns, semantic relationships, and contextual dependencies [9].

However, despite their success, LLMs are not without limitations. A core issue lies in the static nature of their training. Once trained, the model's knowledge is embedded within its weights and remains frozen, with no inherent mechanism for integrating new information or correcting outdated knowledge. This leads to several critical shortcomings:

- **Hallucination**: LLMs often generate plausible-sounding but factually incorrect or fabricated content, especially when queried about topics outside their training distribution.
- **Temporal Staleness**: As LLMs are trained on static datasets, they cannot reflect recent developments, emerging facts, or dynamic domain-specific updates.

- **Lack of Transparency and Explainability**: LLMs cannot cite sources or explicitly trace the origin of their generated information, which makes it difficult to verify the accuracy of responses— an essential concern in domains like healthcare, law, and finance.

To mitigate these challenges, the *Retrieval Augmented Generation* (RAG) architecture has been proposed as a hybrid solution that combines the generative power of LLMs with the factual grounding of external knowledge sources. This paradigm is instrumental in knowledge-intensive applications, such as domain-specific chatbots, where it is critical to ensure high levels of factual correctness, domain alignment, and adaptability [13].

2.4.1 Core Architecture of RAG

A typical RAG system comprises two core components:

1. **Retriever**: The retriever component is responsible for identifying and retrieving relevant documents or passages from a pre-indexed external corpus. This is typically achieved using dense vector representations (embeddings), where both the query and documents are projected into a shared semantic space using a sentence embedding model (e.g., Jina Embeddings, OpenAI Embeddings, or Sentence-BERT). Similarity is computed via vector distance metrics (e.g., cosine similarity), allowing the system to fetch the most semantically relevant chunks of information to the user's query.

2. **Generator**: The generator is typically an autoregressive LLM (e.g., Gemini, GPT, or Mistral) that takes the retrieved documents as additional context during inference. This augmented context is appended to or structured into the prompt, enabling the model to ground its output in retrieved knowledge. In this way, the generator can produce answers that are both coherent and factually supported by external evidence.

This two-stage pipeline—retrieval followed by generation—enables the system to operate beyond the boundaries of the model's training data and adapt to new knowledge without requiring full retraining [9].

2.4.2 Advantages of the RAG Paradigm

The benefits of Retrieval Augmented Generation are various:

- **Factual Grounding**: By referencing documents in real-time, RAG systems significantly reduce hallucination rates and ensure traceable, evidence-based outputs.

- **Domain Adaptation**: RAG allows easy customization for specialized domains (e.g., fashion retail, manufacturing, medical diagnostics) by simply updating or curating the external knowledge base.
- **Real-Time Knowledge Updates**: The knowledge base can be modified or extended independently of the model, supporting dynamic updates with no need for expensive fine-tuning.
- **Explainability**: Retrieved documents can be surfaced alongside the response, enhancing transparency and user trust by making the source of the information explicit [26].

These features make RAG particularly suitable for applications requiring high degrees of accuracy, such as:

- Enterprise knowledge assistants and internal documentation search,
- Customer support systems based on product manuals or FAQs,
- Healthcare or legal decision support tools,
- Scientific Q&A systems grounded in research literature.

2.4.3 RAG IN THE ERA OF LARGE LANGUAGE MODELS

The development of RAG is closely tied to the evolution of LLMs and Transformer-based architectures. Early explorations in this field aimed to enhance language models with auxiliary knowledge during pre-training (e.g., using entity embeddings or knowledge graphs). However, the introduction of models like GPT-3 and ChatGPT, with their in-context learning capabilities, shifted the focus toward runtime augmentation through retrieval.

Recent advancements in the RAG field include:

- **In-Context Retrieval**: Dynamically injecting retrieved content into prompts during inference.
- **Multi-hop Retrieval**: Fetching information across multiple documents to answer complex queries requiring reasoning.
- **Retrieval-Augmented Fine-tuning**: Fine-tuning both the retriever and the generator components in an end-to-end manner for improved alignment and coherence.
- **Memory-Augmented Transformers**: Integrating long-term memory with RAG-style retrieval to persist conversational context across sessions.

Despite rapid growth, research in RAG has often focused more on method development than on systematic evaluation. Key open challenges include:

- Designing reliable evaluation metrics for factual accuracy and relevance in generated responses,
- Managing retrieval noise, i.e., irrelevant or low-quality documents that can degrade output quality,
- Balancing efficiency and latency, especially in real-time applications like chatbots [8].

2.5 Vector Databases in LLM-Based Systems

The integration of LLMs into real-world virtual assistants has driven a rapid evolution of data infrastructure, particularly in how unstructured data is stored, indexed, and retrieved. In this context, Vector Database (VDB) has become a foundational component, enabling systems to manage and search high-dimensional vector representations of data efficiently. Unlike traditional relational databases, which are optimized for structured data and exact-match queries, VDBs are specifically designed to handle semantic similarity search over large-scale datasets consisting of dense embeddings derived from textual, visual, and multimodal sources [10].

2.5.1 The Role of Vector Databases in Modern AI Pipelines

Vectors-numerical representations of data-are central to many AI applications, especially those involving deep learning and natural language processing. When working with LLMs, raw input such as text, audio, or images is first transformed into dense vector embeddings using model-specific encoders. These embeddings encode the semantic meaning of the input and are used for a wide range of downstream tasks, including document retrieval, ranking, clustering, and recommendation [20].

However, the effective use of such vectors in production environments requires specialized infrastructure. Traditional databases are not designed to support fast, approximate similarity search or the high dimensionality typical of modern embeddings (often ranging from 256 to over 1,000 dimensions). This is where vector databases become essential.

VDBs enable:

- Efficient similarity-based retrieval of documents or objects based on vector distance (e.g., cosine similarity, Euclidean distance);

- Scalable storage of high-dimensional vectors with low-latency access;
- Support for unstructured and semi-structured data, often including metadata for filtering and post-processing;
- Real-time performance, even with large volumes of documents, queries, and concurrent users.

These characteristics make VDBs ideal for building intelligent virtual assistants that require dynamic knowledge retrieval from large corpora, including proprietary knowledge bases, FAQs, internal documents, or user-generated content.

2.5.2 Core Technologies Behind Vector Databases

At the heart of VDB functionality is the Approximate Nearest Neighbour (ANN) search, which allows systems to retrieve semantically relevant vectors from millions of entries with sub-second latency. ANN algorithms trade some precision for speed, using advanced indexing methods such as:

- Tree-based indexing (e.g., KD-Trees, Ball Trees);
- Hashing techniques (e.g., Locality Sensitive Hashing);
- Graph-based models (e.g., Hierarchical Navigable Small World (HNSW) graphs);
- Quantization-based techniques (e.g., Product Quantization or Scalar Quantization).

These indexing strategies reduce computational complexity and memory overhead while maintaining high retrieval accuracy. Many modern VDBs also implement hardware acceleration (e.g., GPU support) and distributed architectures to scale effectively in real-time environments.

Additionally, most VDBs support metadata filtering, enabling more fine-grained retrieval by combining vector similarity with structured filtering criteria (e.g., date, category, source). This is particularly valuable for virtual assistants that operate across multiple knowledge domains or need to retrieve context-specific information [10].

2.5.3 Advantages Over Traditional Databases

The differences between traditional relational databases and vector databases are both architectural and functional. While relational databases excel at structured queries and transactional integrity, they struggle with:

- Semantic search (e.g., "find documents similar in meaning");

- Managing high-dimensional feature vectors;
- Efficient indexing of unstructured or dynamic content.

By contrast, VDBs are purpose-built for these tasks and offer three major advantages:

1. **Semantic Similarity Search**
   VDBs can retrieve the most relevant entries based on conceptual similarity, rather than exact matches. This is crucial for LLM-based assistants that interpret intent and context dynamically rather than relying on predefined keyword mappings.
2. **Scalability and Real-Time Performance**
   Thanks to distributed storage and parallelized indexing, VDBs can handle vast datasets with high availability and low latency, making them suitable for enterprise-grade virtual assistant deployments.
3. **Support for Unstructured Data**
   Unlike relational systems, VDBs are well-suited for managing heterogeneous data types, such as text, image, audio, etc., and can combine them in multimodal retrieval pipelines. This flexibility opens the door to more advanced virtual assistants capable of answering across formats.

2.5.4 VECTOR DATABASES IN RETRIEVAL-AUGMENTED GENERATION (RAG)

One of the most impactful uses of VDBs in the context of LLMs is their integration into RAG architectures. RAG enhances the generation capabilities of LLMs by grounding their responses in retrieved content from a vector index. When a user submits a query, the system:

1. Converts the query into a vector;
2. Searches the VDB for top-k most similar documents;
3. Feeds these results into the LLM as external context for response generation.

This approach reduces hallucinations, ensures better alignment with the domain-specific knowledge, and makes the assistant more accurate and reliable. Without an efficient VDB underpinning the RAG pipeline, such systems would be too slow or inaccurate for real-time use.

2.5.5 FAISS: Performance-Oriented Vector Search

Facebook AI Similarity Search (FAISS) is one of the most widely adopted open-source libraries for efficient similarity search and clustering of dense vectors. Developed by Meta AI, FAISS is designed to operate at scale, targeting scenarios where raw performance is the top priority.

FAISS implements a wide range of Approximate Nearest Neighbour (ANN) algorithms, including Inverted File Indexes (IVF), Product Quantization (PQ), and HNSW (Hierarchical Navigable Small World graphs), which allow fast and accurate search over billions of vectors. A notable strength of FAISS is its GPU acceleration capabilities, which significantly enhance performance for large datasets and reduce response latency in production environments.

However, FAISS was not originally designed with modern LLM workflows in mind. While it excels at high-speed indexing and retrieval, it exhibits several limitations:

- No built-in metadata filtering, making it less suited for use cases requiring contextual search (e.g., filtering by category, document type, or timestamp);
- No native persistence layer, meaning developers must implement additional mechanisms to store and reload the vector index between sessions;
- Lack of a RESTful API or server-based interface, requiring direct use of Python bindings and custom integration in backend systems.

These constraints make FAISS an optimal choice for large-scale, performance-critical applications, such as recommendation systems, social media search, or scientific computing tasks. However, for modular and rapidly evolving projects involving Retrieval-Augmented Generation (RAG) or prototyping with LLMs, FAISS may impose additional development overhead [7].

2.5.6 ChromaDB: Flexibility for LLM-Centric Workflows

ChromaDB is a modern, open-source vector database designed explicitly for LLM-first applications. It supports persistent storage, rich metadata filtering, and document versioning, making it particularly well-suited for use cases that combine semantic search with contextual constraints.

Unlike FAISS, ChromaDB provides out-of-the-box integration with popular frameworks used in LLM development, including LangChain, LlamaIndex, and HuggingFace Transformers. Its Pythonic API and embedded setup model allow developers to experiment with and deploy local or small-scale RAG pipelines rapidly.

ChromaDB is not built for the same scale as FAISS but instead focuses on developer productivity and seamless integration within conversational AI workflows. Its key advantages include:

- Simple setup and local usage, allowing quick iteration during development;
- Support for metadata queries, such as filtering results based on source, category, or tags;
- Tight compatibility with LangChain pipelines, facilitating smoother implementation of intelligent virtual assistants.

For this project, developing a modular and domain-adaptable LLM-based virtual assistant, ChromaDB was selected as the primary vector storage solution. Its combination of flexibility, ease of use, and alignment with the RAG architecture made it a better fit than FAISS, particularly in the early stages of development and prototyping. While FAISS remains a powerful option for scaling performance in future iterations, ChromaDB provides a more balanced foundation for experimentation, rapid growth, and integration with other components of the system [3].

2.6 Sentiment Analysis in Conversational Systems

Sentiment Analysis (SA), also referred to as opinion mining, is a subfield of Natural Language Processing (NLP) that focuses on identifying, extracting, and classifying the affective states conveyed in text. In the context of conversational systems, such as chatbots and virtual assistants, sentiment analysis plays a key role in enhancing user experience, detecting frustration, and enabling contextually appropriate responses. This section presents an overview of sentiment analysis, its main challenges, and its relevance to the development of intelligent conversational agents [22].

2.6.1 Definition and Scope

Sentiment analysis involves determining the polarity (positive, negative, or neutral) of a given text. More advanced approaches may attempt to classify emotions (e.g., joy, anger, sadness), detect sarcasm or irony, or assess the intensity of sentiment. In chatbot systems, SA is typically employed to understand user satisfaction, prioritize responses, or escalate conversations to human operators when negative sentiment is detected.

The scope of sentiment analysis can vary depending on the level of granularity required. At a document level, it determines the overall sentiment of a message or user utterance. At a sentence level, it classifies individual statements. At an aspect-level, the analysis is more fine-grained, associating sentiments with specific topics, entities, or features (e.g., "The assistant is helpful, but the UI is confusing") [16].

2.6.2 Subjectivity and Polarity Detection

Two central components of sentiment analysis are subjectivity detection and polarity classification. Subjectivity detection determines whether a sentence expresses a personal opinion or an objective fact. This distinction is particularly relevant in virtual assistants that need to differentiate between requests for factual information and expressions of dissatisfaction or preference.

Once a text is classified as subjective, polarity detection identifies whether the sentiment is positive, negative, or neutral. For example, a statement like "This assistant never understands me" clearly conveys a negative sentiment, while "Thanks, that was helpful" reflects a positive one. Polarity classification is often performed using supervised machine learning models trained on labelled datasets or, more recently, using pre-trained language models fine-tuned for sentiment-related tasks.

2.6.3 Challenges in Sentiment Analysis

Despite its apparent simplicity, sentiment analysis presents several non-trivial challenges:

- **Context Dependence:** Sentiment is often highly dependent on context. A phrase such as "You're too clever" could be complimentary or sarcastic, depending on tone, prior exchanges, or user history.
- **Ambiguity and Irony:** Human language is full of nuances, idioms, and rhetorical devices that are difficult for machines to interpret. Sarcasm, irony, and understatement often lead to misclassification.
- **Domain Specificity:** Sentiment expressions vary across domains. Words that convey negative sentiment in one context (e.g., "hot" for electronics) may be positive in another (e.g., "hot" for fashion or trends).
- **Multilingual and Code-Switching Texts:** In multilingual environments or informal conversations, users often mix languages (code-switching), making sentiment classification more complex.
- **Emotion Intensity:** Beyond simple polarity, capturing the intensity or subtlety of an emotional response (e.g., irritation vs. anger) is an ongoing research challenge, particularly relevant to empathetic virtual assistants [24].

2.6.4 Sentiment Analysis in Virtual Assistants

In intelligent conversational agents, sentiment analysis can serve multiple roles. It can help adapt the tone of responses, personalize user interactions, and manage escalation paths. For instance, persistent negative sentiment from a user may trigger the system to offer human support, provide alternative answers, or initiate a recovery strategy.

Additionally, tracking sentiment over time enables long-term profiling of user satisfaction and conversational success. Virtual assistants equipped with sentiment-aware mechanisms can proactively detect dissatisfaction, misunderstandings, or even user churn risk in customer service scenarios.

Moreover, sentiment analysis can improve the training of dialogue models by providing feedback signals about the effectiveness of a system's utterances. In reinforcement learning settings, sentiment can be treated as a reward signal guiding the evolution of more empathetic and user-aware systems [15].

2.6.5 Related Tasks and Integration with Dialogue Systems

Sentiment analysis is often integrated with other NLP tasks in chatbot pipelines, including:

- **Intent Recognition:** Combining sentiment with intent detection allows systems to distinguish between benign and problematic requests (e.g., a refund request expressed with frustration).
- **Dialogue State Tracking:** Sentiment is used to enrich the dialogue state with emotional cues, which can influence the strategy for follow-up responses.
- **Response Generation:** In generative models, sentiment classification can guide the emotional tone of generated replies, making interactions more natural and human-like.
- **User Segmentation and Analytics:** Aggregate sentiment trends help developers understand how different segments of users interact with the system, highlighting areas for improvement.

2.7 Flutter as a Cross-Platform Frontend Framework

Flutter, an open-source Software Development Kit (SDK) developed by Google in 2017, has rapidly emerged as a leading framework for building cross-platform applications. It enables the development of performant, natively compiled applications for multiple platforms, including Android, iOS, web, desktop, and embedded systems, using a single codebase written in Dart. Due to its ability to streamline development workflows, reduce costs, and maintain a consistent user experience across platforms, Flutter has become an attractive choice for building virtual assistants and conversational interfaces deployed across a wide range of devices.

2.7.1 Overview and Architecture

Unlike traditional cross-platform solutions that rely on WebView components or bridge-based rendering, Flutter renders its UI natively using its own high-performance rendering engine, Skia. This approach allows for pixel-perfect control over the user interface, consistent appearance across platforms, and low-latency interactions. Dart, the programming language behind Flutter, is object-oriented and statically typed, offering both just-in-time (JIT) and ahead-of-time (AOT) compilation, which contributes to fast development cycles and performant runtime behaviour.

Key architectural advantages of Flutter include:

- Native compilation to Advanced RISC Machine (ARM) code, resulting in fast execution on mobile and embedded systems;
- Custom widget rendering, eliminating reliance on Original Equipment Manufacturer (OEM) components or platform-specific UI layers;
- Hot reload, which enables near-instantaneous code updates during development without restarting the app;
- Modular package ecosystem, allowing for easy integration of third-party libraries and system capabilities.

These features collectively support the development of robust and responsive applications, which are essential in virtual assistant systems that demand real-time responsiveness, adaptive interfaces, and interactive multimedia capabilities [23].

2.7.2 Use in Conversational Interfaces

For this virtual assistant project, Flutter was selected primarily for its ability to offer a consistent, high-quality user experience across kiosk, tablet, and mobile form factors. The application relies on Flutter's custom rendering pipeline to build dynamic interfaces, including:

- Streaming response rendering, used to display real-time typing animations for chatbot replies;
- Markdown rendering support, enabling formatted output for LLM-generated messages;
- Multimodal input support, with integration of device capabilities such as a microphone or a camera;
- Support for animated transitions, crucial for interactive and engaging assistant workflows.

Compared to traditional native Android development or Hypertext Markup Language (HTML) based kiosk UIs, Flutter enabled a more rapid iteration cycle, easier maintenance, and greater consistency in visual presentation. Its composable UI system allowed for tailored interfaces that adapt to the conversational state, enhancing the user experience in scenarios where immediate feedback and intuitive interactions are critical.

2.7.3 Cross-Platform and Scalability Benefits

From a technical and operational standpoint, Flutter introduces several strategic benefits to virtual assistant development:

- Unified development team: A single team can build and maintain the app for multiple platforms, reducing coordination overhead and version fragmentation;
- Simultaneous feature releases: New features or bug fixes can be deployed across all platforms at once, ensuring uniform functionality and user experience;
- Faster time-to-market: Flutter's expressive syntax and powerful tooling enable shorter development cycles, making it ideal for prototype iteration and feature validation;
- Cost-effectiveness: By reducing platform-specific overhead, Flutter lowers the total cost of development and maintenance- an important factor for Minimum Viable Product (MVP) and scalable deployments.

These advantages align well with the needs of LLM-based assistant systems, particularly in retail or customer-facing environments where rapid deployment and visual consistency are paramount [5].

2.7.4 Limitations and Considerations

Despite its strengths, Flutter is not without its limitations. Some challenges include:

- Dependency on Dart, a less widespread language compared to JavaScript or Kotlin, which may limit talent availability;
- Initial app size, which may be larger due to the embedded rendering engine;
- OS-level feature lag, as platform updates often reach native SDKs before being integrated into Flutter;
- Limited third-party libraries for niche device features, although this is rapidly improving with community and corporate support.

Nonetheless, most of these issues are mitigated in practice. For example, Flutter allows the injection of native code when needed, ensuring access to the full range of device capabilities. Moreover, strong community support and thorough documentation continue to drive rapid evolution of the ecosystem [23].

3

System Architecture

3.1 General Architecture Diagram

The proposed system adopts a *client-server* paradigm, extended with a **Retrieval-Augmented Generation** (RAG) mechanism and a sentiment-driven feedback loop to refine the chatbot's performance progressively.

The architecture is decomposed into the following core components:

- **Client Application** - A lightweight front-end, implemented either as a command-line interface or a Flutter-based mobile application, responsible for collecting user queries and displaying chatbot responses.
- **FastAPI Server** - Acts as the central orchestrator, coordinating the retrieval, generation, and sentiment analysis modules. It ensures stateless communication for each query while maintaining contextual memory for ongoing conversations.
- **Retrieval Layer (ChromaDB)** - A vector database storing semantic embeddings of pre-processed catalogue documents. This enables fast and accurate retrieval of relevant passages through approximate nearest-neighbour search.
- **Language Model (LLM)** - A large language model, accessed either via the *Google Gemini API* (cloud-based) or a local deployment through Ollama, responsible for synthesising coherent and contextually accurate responses.
- **Sentiment Analysis Module** - A supervised machine learning classifier (Naive Bayes) trained to detect positive, neutral, or negative sentiment in user feedback. Its output feeds into a *global memory* mechanism to influence subsequent interactions.
- **External Interfaces** - Secure API integrations, including *Google Gemini* for natural language generation and *Ngrok* for exposing development endpoints to the public Internet.

- **UI Layer** - A cross-platform graphical interface built with Flutter, designed for intuitive mobile interaction without compromising backend flexibility.

The operational flow proceeds as follows:

1. The user submits a query via the client application.
2. The server queries *ChromaDB* to identify the top-k most semantically relevant chunks of text from the pre-processed catalogue.
3. These retrieved passages, together with the original user query, are passed as context to the LLM.
4. The LLM generates a natural language answer that blends factual information from the retrieved chunks with its own generative reasoning capabilities.
5. The response is returned to the client application.
6. At the end of the conversation, the chat feedback is processed by the sentiment analysis module.
7. The global memory is updated based on the sentiment classification, enabling adaptive improvements to the response strategy over time.

3.2 Choice of Technologies

The technology stack was selected to balance *performance*, *scalability*, and *developer productivity*, while enabling both local and cloud-based deployments.

3.2.1 Gemini API (Google Generative AI)

The *Gemini* family of models was chosen for its high performance in complex question-answering scenarios, advanced reasoning capabilities, and low-latency response times. Being cloud-hosted, it eliminates the need for dedicated on-premise GPUs, reducing infrastructure overhead. Additionally, its fine-grained parametrisation (e.g., temperature control) facilitates precise tuning for domain-specific language tasks.

3.2.2 ChromaDB

ChromaDB is an open-source, in-memory and persistent vector database optimised for similarity search in RAG architectures. It offers:

- High-throughput storage and retrieval of dense vector embeddings.

- Tight integration with the *LangChain* framework.
- Efficient indexing structures for cosine similarity and other distance metrics.

Its lightweight deployment requirements make it suitable for both local prototyping and production deployment.

3.2.3 FastAPI

Selected for its asynchronous request-handling capabilities, built-in support for Python type hints, and straightforward integration with machine learning pipelines. FastAPI serves as the primary orchestration layer, exposing RESTful endpoints for conversation management, retrieval invocation, and sentiment analysis. Its modular design also facilitates future extension to support WebSocket-based real-time interaction.

3.2.4 Flutter

While not strictly required for core chatbot functionality, Flutter was chosen for the mobile user interface due to:

- **Cross-platform development** - Single codebase for Android and iOS.
- **High-performance rendering** - GPU-accelerated UI with native-like responsiveness.
- **Ease of integration** - Can interact with the FastAPI backend via HTTP APIs without requiring protocol modifications.

3.3 Modular Overview: Retrieval, LLM, API, UI

3.3.1 Retrieval Module

This module transforms raw catalogue PDFs into a searchable semantic index. It includes:

- **Pre-processing** - Parsing of PDF documents to extract structured content (headings, paragraphs, tabular data) while removing irrelevant elements.
- **Chunking** - Segmentation of extracted text into contextually coherent chunks suitable for embedding.

- **Embedding** - Transformation of text chunks into high-dimensional vectors via HuggingFace sentence transformers.
- **Indexing** - Storage of embeddings in ChromaDB with associated metadata for fast similarity search.

At query time, the retrieval module executes a k-nearest neighbour search to return the most relevant content snippets.

3.3.2 LLM Module

The LLM module is responsible for the generative reasoning step of the RAG pipeline. It:

- Accepts as input the retrieved text chunks and the original user query.
- Constructs a prompt that injects retrieved content into the model's context window.
- Produces a fluent, factually grounded natural language response.

By switching between *Gemini API* and local *Ollama* models, the architecture supports both high-performance cloud inference and offline operations.

3.3.3 API Layer

Implemented using FastAPI, the API layer provides well-defined endpoints:

- `POST /start_conversation/` - Initializes a session with dedicated chat memory.
- `POST /chat/` - Processes user queries, invoking both the retrieval and LLM modules.
- `POST /sentiment/` - Runs sentiment classification on feedback and updates the global memory.

Ngrok integration allows secure exposure of the local API for testing without deploying to production infrastructure.

3.3.4 UI Layer

The UI layer is designed for modularity and platform independence:

- **CLI Client** - Minimal interface for rapid testing and debugging.
- **Flutter Mobile App** - Rich user experience with cross-platform support.

Both clients interact exclusively with the backend API, preserving loose coupling and facilitating independent evolution of the user interface.

4 Implementation

4.1 Data Pipeline

The data pipeline is a fundamental component of the system, responsible for transforming raw PDF documents into structured and semantically enriched text data, ready for indexing and retrieval in the later stages of the application. Given the complexity and heterogeneity of technical manuals and product catalogues, this pipeline addresses several challenges: accurate extraction of tabular data, contextualisation via titles and headers, isolation of meaningful narrative text, and generation of natural language summaries.

The pipeline is implemented in Python and leverages a combination of open-source libraries and language models to achieve robust extraction and preprocessing.

4.1.1 Table Extraction

The technical manuals of our dataset contain data organised in tables, such as product specifications, codes, and measurement values. Extracting these tables accurately is crucial for downstream processing.

To this end, the `Camelot` library is used in *lattice* mode, which detects tables by identifying the lines and borders that define cell boundaries. This method is especially effective for PDFs with clearly delineated tables.

Each extracted table is converted into a Pandas DataFrame for further cleaning. For instance, tables with only two columns are transposed to ensure consistency in orientation. The first row is treated as the header to label columns properly. Tables that contain only a trivial "Page" column or are empty are discarded to reduce noise.

The cleaned tables are then saved as individual CSV files in a dedicated directory, enabling traceability and facilitating manual inspection if needed.

Pseudocode: Table Extraction and Saving

```
FUNCTION extract_tables_with_bbox(pdf_path, pages):
    USE library to detect tables in 'lattice' mode on given pages
    RETURN list of detected tables with bounding boxes

FUNCTION save_tables_to_csv(tables, csv_directory):
    INITIALIZE empty list csv_files
    FOR each table in tables:
        IF table has only 2 columns:
            TRANSPOSE the table for consistency
        SET first row as header
        REMOVE trivial tables
        ADD a 'Page' column indicating page number
        SAVE table as CSV file in csv_directory
        ADD CSV file path to csv_files
    RETURN csv_files
```

4.1.2 Title Extraction

Document structure and context are enhanced by extracting page titles and section headers. Titles help clarify data, improve semantic understanding, and assist in prompt formulation for language models.

Using `PyMuPDF`, the pipeline analyses text blocks on each page. Since titles tend to be uppercase and located near the top of the page, the extraction filters text lines within the upper 150 pixels. It selects those entirely in uppercase and longer than five characters, excluding strings with numeric prefixes.

The extracted titles per page are stored in a dictionary for efficient lookup during content aggregation.

Pseudocode: Title Extraction

```
FUNCTION extract_titles(pdf_path, max_pages):
    OPEN pdf document
    INITIALIZE empty list titles_per_page
    FOR each page up to max_pages:
        LOAD page text blocks sorted by vertical position
        INITIALIZE empty list title_lines
        FOR each text block near top of page:
            SPLIT block into lines
```

```
            FOR each line:
                IF line is uppercase, > 5 chars, no leading digits:
                    ADD line to title_lines
                ELSE IF title_lines not empty:
                    BREAK inner loop
            IF title_lines not empty:
                BREAK outer loop
        COMBINE title_lines into full_title string
        ADD dictionary {page_number, full_title} to titles_per_page
    RETURN titles_per_page
```

4.1.3 Text Extraction Excluding Tables

A critical challenge in PDF processing is separating narrative text from tabular content to avoid duplication and confusion. The pipeline achieves this by combining the bounding boxes of tables detected via `Camelot` with the text blocks extracted from `PyMuPDF`.

For each page, the bounding boxes of all tables are transformed into PyMuPDF's coordinate system to allow geometric comparison. Text blocks overlapping with any table bounding box are excluded. Additionally, text blocks located within 50 pixels of the top or bottom of the page are discarded to avoid headers and footers, which contain repetitive or irrelevant information.

Blocks that contain image placeholders or watermark texts are also filtered out. The remaining text blocks are sorted vertically to preserve the reading order and concatenated into clean textual content.

Pseudocode: Text Extraction with Table Exclusion

```
FUNCTION extract_text(pdf_path, max_pages):
    OPEN pdf document
    INITIALIZE list text_per_page
    FOR each page up to max_pages:
        EXTRACT tables on current page with bounding boxes
        CONVERT table bounding boxes to coordinate system
        INITIALIZE list text_blocks
        FOR each text block on page:
            IF block overlaps any table bounding box:
                SKIP block
            ELSE IF block is near page header/footer:
```

```
            SKIP block
        ELSE IF block is empty or very short:
            SKIP block
        ELSE:
            ADD block text and position to text_blocks
    REMOVE image placeholders and wmk texts from text_blocks
    SORT text_blocks by vertical position (top to bottom)
    CONCATENATE block texts into page_content
    ADD {page_number, page_content} to text_per_page
RETURN text_per_page
```

4.1.4 Natural Language Table Description

To facilitate semantic retrieval and improve the quality of prompts fed to the language model, each extracted table is summarised with a concise, natural language description generated in Italian.

This is achieved by formatting a prompt template that includes the table's CSV representation and the relevant product or page title, and feeding it to the LLM through LangChain's `LLMChain` abstraction. The prompt instructs the model to describe the data content, highlighting key columns and relationships, constrained to a maximum of 50 tokens.

The resulting summaries enrich the dataset with human-readable interpretations of tabular data, enhancing downstream retrieval and generation.

Pseudocode: Table Description Generation

```
FUNCTION describe_table(title, csv_table):
    DEFINE prompt_template with placeholders for title and csv_table
    INITIALIZE language_model with specified parameters
    FORMAT prompt with title and csv_table
    RUN prompt through language_model to get description
    RETURN generated description
```

4.1.5 Content Aggregation and Output Formatting

The pipeline aggregates titles, tables, descriptions, and free text on a per-page basis to produce a unified, structured representation of the document content. Each page's data includes:

- The page number and file name for traceability.

- The extracted or inferred title.
- The natural language summary of any tables present.
- The CSV-formatted raw table data.
- The narrative text content excluding tables.

This aggregated content is saved as a text file with explicit markers for each data type and page, facilitating easy ingestion into the embedding and retrieval modules of the system.

Pseudocode: Content Aggregation

```
FUNCTION join_content(pdf_path, max_pages, titles_dict, tables,
                      text_without_tables):
    EXTRACT pdf file name from path
    INITIALIZE empty list results_joint
    FOR each page number in 1 to max_pages:
        GET title from titles_dict or empty string
        GET narrative text for page
        GET all tables for page
        IF tables are present:
            FOR each table:
                ADD dict with file, page, title,
                    table summary, raw table, text to results_joint
        ELSE IF narrative text present:
            ADD dict with file,pag,title,narr text to results_joint
    RETURN results_joint
```

4.1.6 Tagged Content Parsing

Once the per-page structured content is generated, it is formatted with explicit section markers (tags) that identify the type of information contained, such as titles, tables, table descriptions, and narrative text. These tags are embedded in the text using a predefined pattern, enabling automated parsing through regular expressions. This approach ensures that the structure can be easily extended to accommodate new content types in the future.

Pseudocode: Tagged Content Parsing

```
FUNCTION parse_tagged_files(folder_path):
```

```
    INITIALIZE empty list parsed_docs
    FOR each file in folder_path:
        READ file content
        SPLIT content by marker "===PAGE==="
        FOR each section in split content:
            EXTRACT tag-value pairs using regex
            UPDATE last seen page number and title if tags found
            FOR each tag-value pair:
                IF tag is not "page" or "title":
                    ADD entry to parsed_docs with:
                        id: generated UUID
                        page: last seen page
                        product_name: last seen title
                        type: tag
                        content: value
    RETURN parsed_docs
```

4.1.7 Metadata Preservation

For efficient storage and retrieval, each parsed entry retains associated metadata, including page number, document title, and content type. Before indexing, all metadata is sanitised to conform to database constraints: only strings, integers, floats, and booleans are allowed, and missing fields are replaced with empty strings.

Pseudocode: Metadata Sanitisation

```
FUNCTION sanitize_for_db(metadata):
    FOR each key, value in metadata:
        IF value is None:
            SET value to empty string
        ELSE IF type of value not in (str, int, float, bool):
            CONVERT value to string
    RETURN sanitised metadata
```

4.1.8 Document Chunking

To optimise retrieval and preserve semantic coherence, each parsed document is split into chunks. A semantic chunking algorithm is employed, which uses the same embedding model intended for vectorisation to identify natural breakpoints, avoiding arbitrary sentence cuts and preserving context.

Pseudocode: Semantic Chunking

```
FUNCTION semantic_chunk(documents, embedding_model):
    INITIALIZE empty list chunks
    FOR each document in documents:
        SPLIT document.content into semantically coherent segments
        FOR each segment:
            CREATE chunk object with segment text and metadata
            ADD chunk to chunks
    RETURN chunks
```

4.1.9 Embedding and Vector Database Storage

Each chunk is embedded into a dense vector representation using the ***alikia2x/jina-embedding-v3-m2v-1024*** embedding model. This *Model2Vec* model is a distilled version of the ***jinaai/jina-embeddings-v3 Sentence Transformer***. The resulting vectors, along with their associated metadata, are stored in a ChromaDB instance. This enables semantic search and filtering in downstream tasks. The database is persisted locally to avoid repeated embedding operations for the duplicate content.

Pseudocode: Embedding and Storage

```
FUNCTION process_and_embed(data_path, db_path, embedder_name):
    PARSE all tagged files from data_path into list of raw documents
    INITIALISE an empty list called documents

    FOR each document in raw documents:
        EXTRACT metadata from the document
        SANITIZE metadata so values are strings, numbers, booleans
        CREATE document obj containing content & sanitised metadata
        ADD the document object to the documents list

    SPLIT all documents into semantically coherent chunks
        USING the embedding model specified by embedder_name

    CREATE a new ChromaDB vector store from the chunks
        USING the same embedding model

    SAVE the ChromaDB vector store to db_path for future retrieval
END FUNCTION
```

4.2 RAG Pipeline step-by-step

The Retrieval-Augmented Generation (RAG) pipeline combines a dense vector index of the processed catalogues with a conversational LLM. At query time, the system retrieves the most relevant catalogue chunks and injects them, along with session and global memories, into a domain-specific prompt. The LLM then produces a grounded answer prioritising catalogue evidence. Post-hoc sentiment analysis labels user-assistant interactions and persists useful exchanges as global memory.

4.2.1 Knowledge Base Initialization

At service startup, the system either builds the vector store (by embedding the previously extracted and chunked documents) or loads an existing `Chroma` database from disk. The same embedding model is used both for indexing and for the retriever.

Pseudocode: Vector Store Initialisation

```
IF vector store directory does not exist:
    CALL embedding(data_path, db_path, embedder_name) to build index
ELSE:
    LOAD vector store from db_path using embedder_name
```

4.2.2 Retriever and Chain Construction

A retriever is created from the vector store to surface the most similar chunks for each user query. These retrieved chunks are then passed to a *stuff*-style document chain. The final RAG chain injects:

1. retrieved catalogue context,
2. current session memory,
3. global memory (sentiment-labelled past dialogues),

into a domain-specific prompt.

Pseudocode: RAG Chain Creation

```
FUNCTION create_rag_chain(retriever, prompt_template, model_name,
                          temperature):
    INITIALIZE LLM with (model_name, temperature)
    CREATE document_chain by combining LLM with prompt_template
```

```
CREATE retrieval_chain that:
    - uses retriever to fetch relevant documents
    - feeds retrieved docs into document_chain
RETURN retrieval_chain
```

4.2.3 Prompt Assembly and Answer Generation

The prompt frames the LLM as a hardware-store expert. It enforces evidence-grounding (use catalogue tables and return exact article codes when present), dictates the order of information sources (catalogues > session memory > global memory), and forbids exposing internal machinery (embeddings, DB IDs, or sentiment labels). The `{context}` slot is filled by retrieved chunks; `{t_memory}` and `{chat_memory}` pass in-session vectors and global memory summaries/IDs, respectively.

Pseudocode: Query Execution

```
FUNCTION answer(user_input, vector_store, prompt_template, model_name,
    temperature, session_memory, global_memory):
    CREATE retriever from vector_store
    RAG = create_rag_chain(retriever, prompt_template, model_name,
    temperature)
    PREPARE inputs:
        input := user_input
        context := retriever
        t_memory := session_memory for current session
        chat_memory := global_memory (sentiment-labelled history)
    INVOKE RAG with inputs to generate grounded answer
    RETURN answer
```

4.2.4 Design Choices and Practical Notes

- **Embedding model** - A single HuggingFace model is used for both indexing and retrieval to keep the vector space consistent.
- **Grounding order** - The prompt enforces catalogues first, then session memory, then global memory. If no evidence is found, the assistant requests clarification rather than hallucinating.
- **Global memory curation** - Only clearly positive/negative interactions are stored; neutral utterances are omitted to reduce noise and bias.
- **Privacy and exposure** - Sentiment labels and DB IDs are never surfaced to the user; they only steer retrieval via prompt slots.

- **Asynchrony** - The LLM call is executed without blocking the event loop; shared state updates are performed within an async lock.

4.3 BACKEND ARCHITECTURE (FASTAPI ROUTES, ASYNC PROCESSING)

The backend is implemented using `FastAPI`, chosen for its asynchronous request handling and native data validation via `Pydantic`. It exposes a minimal set of endpoints to support the RAG pipeline:

1. **/start_conversation** - bootstraps a new chat session and returns a unique identifier.
2. **/chat** - processes a user query, runs retrieval and generation, and stores the turn in ephemeral memory.
3. **/sentiment** - performs post-hoc sentiment classification on the session, persisting relevant interactions to the global vector store.

Stateful information (e.g., per-session transcripts and temporary vector memories) is kept in in-memory maps keyed by `session_id`. At the same time, the durable knowledge base is stored in a `Chroma` database on disk.

4.3.1 SESSION BOOTSTRAP

The `/start_conversation` endpoint is a simple entry point that generates a unique identifier (UUID) for the new session. This identifier links subsequent `/chat` and `/sentiment` calls to the correct in-memory containers.

Pseudocode: `/start_conversation` Endpoint

```
ENDPOINT POST /start_conversation ():
    session_id = GENERATE UUID
    RETURN {session_id}
```

4.3.2 CHAT ORCHESTRATION AND SESSION HANDLING (FASTAPI)

Each request to `/chat` is tied to a `session_id`. The server:

1. Ensures the session containers exist (chat transcript, temporary embeddings, memory IDs).

2. Passes the user query to the RAG chain to generate a grounded response.
3. Appends both user and assistant turns to the transcript.
4. Embeds the latest turn and updates the ephemeral session vector memory.

To avoid race conditions in concurrent requests, these updates are performed under an asynchronous lock.

Pseudocode: `/chat` Endpoint

```
ENDPOINT POST /chat (session_id, user_input):
    WITH async lock:
        IF session_id not present:
            INITIALIZE temporary_memory[session_id]
            INITIALIZE chat_session[session_id]
            INITIALIZE t_memory_ids[session_id]
    bot_response = answer(user_input, vector_store, prompt, model,
         temperature, temporary_memory[session_id], global_memory)
    APPEND user_input, bot_response TO chat_session[session_id]
    EMBED [user_input, bot_response] -> vectors
    APPEND vectors TO temporary_memory[session_id]
    RETURN {session_id, bot_response}
```

4.3.3 Global Memory with Sentiment Analysis (Post-hoc)

After one or more turns, the client calls `/sentiment`. The system runs a classical ML pipeline (vectoriser + Bernoulli Naive Bayes) on user interactions from the session. Only interactions labelled *positive* or *negative* are persisted; *neutral* is discarded. The persisted entries are stored as short documents in the vector store, and their IDs are aggregated as *global memory* for future chats. Importantly, the sentiment label is *never* exposed to the user in the answer, but it influences future retrieval via the global memory prompt slot.

Pseudocode: `/sentiment` Endpoint

```
ENDPOINT POST /sentiment (session_id):
    EXTRACT session transcript temp_mem = chat_session[session_id]
    BUILD user_texts by selecting user turns in chronological order
    PREDICT sentiment_labels =
                classifier.predict(vectorizer.transform(user_texts))
    INITIALIZE list to_persist
```

```
FOR each dialogue pair (user_i, bot_i) with corresponding label:
    IF label in {positive, negative}:
        FORMAT a compact doc:
            {"user": user_i, "bot": bot_i, "label": label}
        APPEND doc to to_persist
ADD to_persist as documents into vector_store
            and COLLECT returned IDs
APPEND returned IDs to global chat_memory (used by the prompt)
```

4.3.4 Concurrency and Robustness

An application-wide asynchronous lock is used to serialise writes to shared in-memory structures, avoiding race conditions when multiple requests target the same `session_id`.

Pseudocode: Concurrency Guards

```
WITH async lock:
    IF session_id not in chat_session:
        INITIALIZE empty transcript
        INITIALIZE empty temporary_memory
        INITIALIZE empty t_memory_ids
    PERFORM updates atomically
```

4.3.5 Error Handling

All routes are wrapped with exception handling blocks:

- Invalid or missing `session_id` HTTP 400.
- Internal RAG chain or embedding errors HTTP 500 with diagnostic details.

Errors are logged for observability but without leaking sensitive data. This ensures easier testing and debugging during the development of the project.

4.4 Prompt Engineering

Prompt engineering is the process of designing, refining, and optimising input prompts for Large Language Models (LLMs) to produce outputs that are relevant, accurate, and contextually aligned with the intended task. A prompt is typically a text string (although it may originate from other modalities, such as transcribed audio) that provides both an instruction and sufficient contextual information to guide the model's reasoning.

The quality and structure of the prompt have a direct impact on the performance of the LLM [14].

4.4.1 Principles of Effective Prompt Engineering

The effectiveness of a prompt depends on several key factors:

1. **Defining the Goal**: Clearly state the expected outcome of the LLM's response. This ensures alignment between the prompt design and the evaluation criteria.
2. **Understanding Model Capabilities**: Tailor the prompt to the strengths and limitations of the specific LLM, including supported formats, context length, and reasoning patterns.
3. **Choosing the Right Format**: Use clear and concise language, ensuring the prompt structure is compatible with the model's natural language understanding.
4. **Providing Context**: Supply additional details (e.g., task-specific background, constraints, prior conversation) to help the LLM avoid generic or inaccurate outputs.
5. **Iterative Testing and Refinement**: Continuously evaluate prompt outputs, adjust phrasing or structure, and measure improvements against defined quality metrics.

These principles serve as a framework, not a rigid checklist; experienced practitioners may deviate from them based on intuition, creativity, and empirical results [25].

4.4.2 Prompt Optimization Techniques

To maximise prompt effectiveness, several strategies are commonly employed:

- Use *few-shot* or *zero-shot* examples to guide reasoning patterns.
- Incorporate domain-specific terminology to increase contextual relevance.
- Apply role prompting (e.g., "You are an expert financial analyst...) to prime the model's tone and reasoning.
- Leverage chain-of-thought or step-by-step reasoning cues to improve multi-step problem solving.
- Employ sentiment or intent classification upstream to adapt prompt structure and tone dynamically.

4.4.3 Prompt Design in This Project

The development of the final system prompt was an iterative process that required extensive experimentation and testing. Throughout the project, several prompt variants were designed, each aimed at improving accuracy, relevance, and the practical usability of responses. Early iterations followed a more generic question-answer pattern, but these often suffered from two common issues:

- The language model tended to hallucinate information when the required data was missing.
- The returned product recommendations lacked precise references to the catalogue item codes, which are essential in a hardware retail setting.

Our objective was to design a prompt that could integrate the Retrieval-Augmented Generation (RAG) pipeline's multi-source context (catalogues, session memory, and global memory) while ensuring that the assistant consistently behaved like a knowledgeable hardware store employee. The resulting prompt enforces a clear order of priority in information retrieval:

1. Search first in the provided *catalogues*.
2. If not found, consult the *current conversation memory*.
3. If still unavailable, check the *global memory* containing previous conversations annotated with user satisfaction labels.
4. If no valid source contains the answer, request clarification from the user instead of providing incorrect or invented information.

This hierarchical approach reduced the risk of hallucination and ensured that, whenever possible, answers were grounded in concrete product data. The prompt explicitly instructs the assistant to return article codes exactly as written in the catalogues (e.g., `Art.1234 567 890`) and to leverage metadata such as page number, section, and product type to refine the response.

Another design choice was to enforce a strictly practical and service-oriented tone, avoiding any mention of technical infrastructure such as embeddings or database identifiers. The assistant is further instructed to interpret possible spelling errors in user queries and ask clarifying questions when necessary, rather than defaulting to "I don't know" responses. Additionally, while the system has access to sentiment labels in the global memory, the prompt prohibits exposing this label in the response, using it only for internal re-ranking, filtering and response improvement.

This final version of the prompt was reached after testing earlier alternatives that:

- Over-emphasised conversation memory, sometimes prioritising outdated or incorrect information over catalogue data.
- Provided verbose, overly technical explanations unsuited for an end-user environment.
- Failed to consistently return catalogue codes, leading to ambiguity for customers and store staff.
- Failed to respond with correct information about the available products.
- Failed to associate the product description and catalogue codes to the right articles.

Through iterative refinement, we achieved a prompt that produced concise, accurate, and actionable answers, consistently grounded in the most reliable source available. This design significantly improved the assistant's usefulness in real-world customer interactions.

4.5 Communication between Frontend and Backend

The communication layer between the Flutter client and the FastAPI backend is designed around a stateless HTTP/JSON interface, with session persistence handled via unique `session_id` tokens generated by the server. This architecture ensures a clean separation of presentation and business logic while enabling scalability and modularity.

4.5.1 Session Management

When a new conversation starts, the client invokes the backend's `/start_conversation` endpoint. The backend responds with a newly generated UUID, which acts as a unique `session_id`. All subsequent requests in the same conversation include this identifier, allowing the backend to retrieve:

- The ongoing chat transcript
- The ephemeral session-specific vector memory
- Sentiment analysis state

This design allows multiple concurrent sessions to be served independently and supports future horizontal scaling.

4.5.2 Request-Response Flow

The interaction between client and backend follows a structured pattern:

1. **Session Creation:** Client sends a `POST` request to `/start_conversation` to receive a `session_id`.
2. **User Query Submission:** Client sends user input and `session_id` to `/chat`.
3. **Backend Processing:**
    - Retrieves relevant context from catalogues, session memory, and global memory.
    - Constructs a prompt and invokes the RAG pipeline.
    - Appends the interaction to the chat history and updates the session vector memory under an asynchronous lock.
4. **Response Delivery:** Backend returns a JSON payload containing the assistant's answer and the `session_id`.
5. **Sentiment Analysis Trigger:** If the conversation reaches at least three user interactions, the client sends a final `/sentiment` request upon termination or timeout.

4.5.3 Timeout and User Experience Management

The Python client implements a custom `input_with_timeout` function to:

- Automatically terminate the conversation if the user remains inactive for a defined period.
- Trigger the `/sentiment` endpoint if the conversation meets the minimum interaction threshold.
- Support real-time input editing before sending to the backend.

This ensures that idle sessions do not persist indefinitely and that sentiment data is captured without requiring explicit user action, and is done only when there's a minimum of information exchanged between the customer and the chatbot, saving only the helpful conversations.

4.5.4 Pseudocode Representation of Client-Backend Interaction

```
START conversation:
    session_id = POST /start_conversation
    count = 0

LOOP until user exits or timeout:
    user_input = input_with_timeout()
    count += 1

    IF user_input in ["exit", "q", "esci"]:
        IF count >= 3:
            POST /sentiment with session_id
        BREAK

    payload = {session_id, user_input}
    response = POST /chat with payload
    DISPLAY bot_response

ON timeout:
    IF count >= 3:
        POST /sentiment with session_id
    EXIT
```

4.5.5 Advantages of the Implementation

- **State isolation:** Each `session_id` is independent, preventing cross-session data leakage.
- **Scalability:** Stateless request handling enables backend horizontal scaling.
- **Robustness:** Timeout and sentiment triggers ensure graceful termination and data completeness.
- **User experience:** Real-time input handling in the client improves usability.

5 Evaluation

5.1 Performance Metrics (Response Time, Accuracy, Relevance)

To assess the performance and usability of the chatbot, we adopted a multi-dimensional evaluation approach focusing on three core metrics:

- **Response Time**
- **Accuracy**
- **Relevance**

5.1.1 Response Time

Response time was defined as the elapsed time between the moment the user sends a query and the moment the chatbot delivers its response. This metric directly affects the *perceived responsiveness* of the system and is critical for preserving the illusion of a natural, human-like interaction.

The measurement was performed by logging timestamps at both the client side (before sending the request and after receiving the response) and the server side (upon request reception and after the generation of the final output). This allowed us to distinguish between:

1. **Network latency** - time spent transmitting the request and receiving the response over HTTP.

2. **Processing latency** - time spent inside the backend, including retrieval from the vector store, prompt assembly, LLM inference, and sentiment processing when applicable.

Our goal was to keep the end-to-end response time below **2 seconds** for most queries, ensuring that customers experience the conversation as fluid and interactive as possible. Short response times were also key to maintaining engagement, especially in contexts where customers may require multiple follow-up clarifications.

5.1.2 ACCURACY

Accuracy measures the *factual correctness* and *linguistic soundness* of the chatbot's answers. Given the nature of the domain - a hardware store catalogue - incorrect or imprecise answers could directly lead to customer dissatisfaction or even operational issues.

Accuracy evaluation was carried out using the following criteria:

- **Factual correctness** - verifying that product codes, specifications, and availability matched exactly the entries in the catalogues indexed in the RAG pipeline.
- **Linguistic correctness** - checking for grammatical correctness, sentence clarity, and adherence to the role constraints (e.g., speaking as an experienced hardware store assistant).
- **Consistency** - ensuring that identical or similar queries yield the same correct information, regardless of session.

To quantify accuracy, a sample of chatbot responses was manually compared against the authoritative data sources (catalogue tables) and annotated with binary judgments (*correct/incorrect*). Accuracy was then calculated as the proportion of correct responses over the total evaluated set.

5.1.3 RELEVANCE

While accuracy ensures that responses are correct, *relevance* ensures that they are *useful* and *directly answer the question posed*. A perfectly accurate answer that is tangential to the user's request would still result in a poor user experience.

Relevance was evaluated through:

1. **Query-Answer Match** - measuring whether the response addressed the specific intent of the user query.
2. **Catalogue Alignment** - confirming that the provided information is drawn from, and fully supported by, the available catalogue data.

3. **Instruction Adherence** - ensuring that the response followed the prompt design priorities (catalogue lookup first, conversation memory second, and global memory last) without introducing unsupported or fabricated information.

To assess relevance, we adopted a manual rating scale:

- 0 = Irrelevant: The answer fails to address the question.
- 1 = Partially relevant: The answer contains some relevant elements but also includes missing or tangential content.
- 2 = Fully relevant: The answer fully satisfies the request and is grounded in catalogue data.

5.1.4 Interdependence of Metrics

These three metrics are strongly interrelated. High accuracy without relevance results in correct but unhelpful answers; low response time without accuracy risks delivering incorrect information quickly; relevance without accuracy risks misleading the customer. The balance among these factors was therefore central to our evaluation process.

We found that optimisations to the RAG retrieval strategy (e.g., semantic chunking, catalogue metadata prioritisation) and prompt refinement had a measurable impact on both accuracy and relevance, while asynchronous processing and in-memory caching significantly improved response times without degrading correctness.

5.2 User Testing: Usability and Effectiveness

The final stage in evaluating the validity and completeness of the project consisted of the assessment of the *usability* and overall *effectiveness* of the chatbot in its final version, namely the mobile application. Particular attention was dedicated not only to the functional aspects of the system, but also to the quality of the interaction, the intuitiveness of the interface, and the overall user experience. The evaluation process was therefore divided into three main phases: **internal testing**, **external feedback** from colleagues, and the **delivery of a proof-of-concept (PoC)** to the industrial partner.

5.2.1 Internal Testing and Debugging

The first phase focused on internal verification and debugging activities, conducted in a controlled and simulated environment. This step was crucial to ensure the

technical stability of the system, the correctness of the conversational flow, and the reliability of the user interface. Particular emphasis was placed on verifying the:

- Responsiveness of the chat
- Consistency of the interaction patterns
- Visual rendering of the application across different devices

An additional design choice was introduced at this stage. While the chatbot was generating a response, the interface displayed a "thinking" animation, instead of leaving the user with an idle blank screen. This solution, inspired by state-of-the-art conversational assistants such as ChatGPT, Gemini, or Claude, was conceived to improve transparency, increase engagement, and enhance the overall perception of fluidity in the interaction.

5.2.2 Peer Testing and Usability Feedback

Following the internal phase, the system was tested by a limited group of colleagues within the division. The goal was to obtain an external but still controlled perspective, allowing us to identify possible usability issues and collect suggestions for aesthetic and functional improvements. This stage provided valuable insights into aspects such as the:

- Intuitiveness of the navigation
- Clarity of the visual elements
- Adequacy of the assistant's responses to user expectations

Peer testers also contributed to defining a preliminary baseline for the application's look-and-feel, complementing the technical validation carried out internally with a more *user-centric evaluation*. Their feedback helped refine elements such as *button placement*, *colour schemes*, and *conversational pacing*, ensuring a more polished and user-friendly interface.

5.2.3 Proof-of-Concept Delivery

The final step of the evaluation process consisted of delivering the chatbot application as a proof-of-concept (PoC) to the partner hardware company. At this stage, the application was considered *stable*, *functionally complete*, and *sufficiently validated* for demonstration purposes. The delivery of the PoC not only marked the conclusion of the

usability and effectiveness testing but also served as a benchmark for future extensions and industrial adoption. This stage represented the transition from a research-oriented prototype to a functional demonstrator capable of supporting real-world usage scenarios. It provided a concrete validation of the design and implementation choices made throughout the project, while also opening the possibility for further refinements based on industrial feedback.

5.3 Comparison with a Baseline (local-only LLM)

The initial requirement expressed by the industrial partner was to develop a virtual assistant capable of running directly on an Android tablet, without relying on any internet connection. This architectural constraint initially oriented the design toward a local-only solution, leveraging lightweight large language models (LLMs) that could be executed within the limited computational resources of the target device.

5.3.1 Baseline Setup: Local LLMs with Ollama

To address this scenario, the Ollama service was selected as the framework for downloading, managing, and executing locally available LLMs. The choice of Ollama was motivated by its ability to handle multiple models efficiently, its relatively simple deployment process, and its compatibility with consumer-grade hardware. Since the target platform was a tablet, significant hardware constraints had to be considered, including limited RAM capacity, reduced CPU processing power, and restricted parallelism due to a smaller number of cores. Consequently, the experimentation phase focused on models with a reduced number of training parameters, such as `qwen3:4b`, `mistral7b`, `phi3:3.8b`, `phi4-mini`, `gemma3:4b`, `gemma2:9b`, `llama3.2:1b`, `llama3.2:3b`, and `llama3.1:8b`. These models were systematically tested to establish a realistic baseline in terms of feasibility and performance.

5.3.2 Evaluation of Local-Only Performance

The results of this evaluation highlighted the severe limitations of adopting a local-only approach. While the smallest models (1B-3B parameters) were technically executable on the device, their generative capabilities were significantly restricted, resulting in incoherent or overly simplistic responses. Larger models (> 4B parameters) provided somewhat better quality in terms of sentence construction and factual correctness, but introduced critical drawbacks:

- **Response Time:** In several cases, the generation of a single response required multiple minutes, which is incompatible with the requirement of sustaining a fluid, real-time conversation with a customer.
- **Accuracy and Coherence:** Even when executed successfully, the answers often lacked correctness, precision, and fluidity. The models struggled to retrieve or generate catalogue-related information accurately.
- **Resource Utilisation :** Running larger models pushed the hardware to its limits, leading to memory exhaustion, overheating, and general instability of the system.

This analysis confirmed that the local-only baseline was insufficient to support the intended use case, particularly for a commercial environment where responsiveness, correctness, and naturalness of interaction are critical factors.

5.3.3 Decision-Making with the Partner

In light of these findings, two possible alternatives were presented to the industrial partner:

1. **Deployment on a Remote Server:** Executing the LLM on a dedicated server with sufficient computational resources and allowing the tablet to act as a lightweight client. This option would have enabled the use of more powerful models while keeping the local device requirements minimal. However, it still implied managing infrastructure and ensuring secure communication.
2. **Adoption of a Cloud-Based LLM:** Leveraging an enterprise-grade LLM service (e.g., Gemini) with contractual guarantees for data security and privacy. This approach would ensure access to state-of-the-art models with superior accuracy and responsiveness, without requiring investment in custom server infrastructure.

After evaluating both alternatives, the partner selected the second option, favouring the integration of the ***Gemini 2.0 Flash*** model. This decision was motivated by its ability to deliver fast, accurate, relevant, and human-like responses while maintaining enterprise-level guarantees in terms of privacy and data security. As a result, the project evolved from the baseline local-only setup to a cloud-based architecture. This transition ensured that the chatbot could sustain real-time, engaging interactions with users while preserving correctness and reliability, thus meeting both technical and business objectives.

6 Challenges and Limitations

6.1 Issues with Cloud LLMs

While the adoption of cloud-based large language models (LLMs) such as Gemini 2.0 Flash allowed the system to achieve significant improvements in terms of response quality, accuracy, and overall conversational fluency, this architectural choice also introduced new challenges. Specifically, two classes of issues became particularly relevant in the deployment phase: **network latency** and **tokenization overhead**. Both factors directly impact the usability, efficiency, and cost-effectiveness of the solution, and therefore require careful analysis.

6.1.1 Latency Issues in Cloud LLMs

Network latency refers to the delay introduced by the transmission of data between the client (the Android tablet running the assistant) and the cloud-hosted LLM service. In real-time applications such as conversational agents, latency plays a critical role in shaping the user experience.

Several aspects make network latency particularly problematic in the context of LLM-based assistants:

- **Real-Time Interaction:** Conversational agents must sustain a fluid, human-like dialogue. Delays longer than a few hundred milliseconds disrupt user engagement, breaking the illusion of natural conversation.
- **Cost Efficiency:** When requests spend significant time in transmission rather than in computation, system resources are underutilized, reducing efficiency and potentially raising operational costs.

- **Scalability:** For systems expected to handle thousands or millions of concurrent requests, even small latency increments can accumulate into significant service bottlenecks.
- **Reliability in Critical Systems:** In contexts where timely responses are essential (e.g., industrial monitoring or autonomous systems), delayed outputs may compromise safety and decision-making.

Latency is not only the result of raw network delays. The number of tokens in the prompt and response directly impacts inference time, as the model generates output token by token. The total response latency T can be decomposed into three components:

$$T = T_{\text{network}} + T_{\text{inference}} + T_{\text{post-process}}$$

Where:

- T_{network}: Transmission delay between client and server.
- $T_{\text{inference}}$: Model computation time, approximately proportional to the number of tokens N.
- $T_{\text{post-process}}$: Overhead for decoding, formatting, and additional pipeline tasks.

As $T_{\text{inference}} \propto N$, reducing the number of tokens per query or optimizing the prompt design becomes a direct strategy to minimize response time [21].

6.1.2 Tokenization Challenges

Another critical issue in the use of cloud LLMs is tokenization, the process of splitting input text into smaller units called tokens, which are then numerically encoded for processing by the model. While tokenization is fundamental to how LLMs function, it introduces challenges that affect both performance and cost.

First, tokenization increases the computational workload. For example, the simple sentence *"Hello, how are you?"* may be tokenized into ["Hello", ",", "how", "are", "you", "?"]. More complex or morphologically rich words, such as "tokenization," might be further decomposed into ["token", "ization"]. The total number of tokens is often higher than the number of words in natural language, with ratios varying across languages.

Second, tokenization has a direct effect on:

- **Inference Latency:** More tokens result in longer processing times during both input encoding and output generation.
- **Memory Usage:** Each token requires additional storage and processing resources during inference.

- **Operational Costs:** Cloud LLM providers typically charge based on the number of input and output tokens, meaning that more extended tokenized sequences directly increase usage costs.

The variability of tokenization across different models further complicates deployment. A paragraph of 25 words might correspond to 30 tokens in one model and 40 in another, leading to differences not only in latency but also in cost calculations. As a result, token efficiency becomes an important design consideration in the choice of LLM provider and in the engineering of system prompts.

6.1.3 Impact on the Deployed System

For the assistant developed in this project, these two issues-latency and tokenization-proved to be tightly coupled. Network delays amplified the perceived slowness of responses, particularly when prompts expanded into hundreds of tokens and the model produced long-form answers token by token. From the user's perspective, this could create noticeable pauses or lags during interaction, reducing conversational fluency.

To mitigate these issues, several strategies were considered:

- Designing shorter and more efficient prompts to minimize token counts.
- Employing partial streaming of outputs, allowing the system to display responses incrementally as tokens are generated, rather than waiting for the whole message.
- Exploring token-efficient models or compression techniques to reduce cost and latency overhead.

Despite these challenges, the benefits of cloud LLMs, in terms of quality, accuracy, and naturalness of conversation, were judged to outweigh the drawbacks. Nevertheless, latency and tokenization remain two of the most critical open issues in real-world deployments of LLM-based assistants, shaping not only performance but also economic sustainability.

6.2 Handling Ambiguous Queries

One of the recurring challenges in the deployment of LLM-based assistants is the management of *ambiguous queries*. While LLMs have demonstrated remarkable proficiency in text generation and question-answering tasks, they often encounter user inputs that are underspecified, vague, or admit multiple interpretations. If left untreated, ambiguity can lead to responses that misrepresent the user's true intent, undermining the assistant's reliability.

Ambiguity arises in many contexts: from casual users who lack the domain-specific knowledge to articulate precise queries, to situations where natural language itself admits multiple meanings. Addressing this challenge requires both technical and interactional strategies to ensure that the assistant remains not only functional but also user-friendly and safe.

6.2.1 Nature and Sources of Ambiguity

Ambiguity in user queries can take several forms:

- **Lexical ambiguity:** A single word admits multiple meanings. For example, the query *"Show me the Java tutorial"* may refer either to the programming language or the Indonesian island.
- **Syntactic ambiguity:** The structure of the query allows more than one interpretation. For instance, *"Find hotels near restaurants with parking"* may mean hotels near restaurants that have parking, or restaurants with nearby hotels that offer parking.
- **Contextual ambiguity:** The user omits critical context, leading to underspecified requests. For example, *"Book me a table for tonight"* leaves open the choice of restaurant or cuisine.
- **Domain-driven ambiguity:** In specialized fields like law or medicine, queries may appear unambiguous to the user but require technical disambiguation. Incorrect interpretation in these contexts can lead to severe consequences.

From the perspective of the assistant, two primary hurdles complicate ambiguity handling:

1. **Recognition gap:** LLMs are not inherently trained to express uncertainty. Even if they internally detect ambiguity, this recognition is not always reflected explicitly in their responses.
2. **Knowledge-dependence:** A model's ability to perceive ambiguity depends on the breadth of its training knowledge. With limited knowledge, the model may misinterpret an ambiguous query as unambiguous, producing an overconfident yet incorrect answer.

6.2.2 Strategies for Managing Ambiguity

To ensure robust and user-centred handling of ambiguous inputs, several strategies can be employed in the design of the conversational pipeline:

- **Clarification prompts:** The system may ask follow-up questions to disambiguate intent. For instance, if the user asks *"Give me some information about the drill"*, the assistant might respond: *"Do you mean the twist drill or the flat-tip drill?"* .
- **Offering multiple interpretations:** Instead of committing to a single reading, the assistant may return several possible interpretations and allow the user to select one. This strategy balances transparency with efficiency.
- **Uncertainty expression:** When ambiguity persists, the model should explicitly express its uncertainty, avoiding the risk of providing misleading information. For example: *"I am not certain whether you meant X or Y. Could you clarify?"* .
- **Contextual enrichment:** Leveraging conversation history, user profiles, or domain constraints can reduce ambiguity by filling in missing context. For example, if a user has previously searched for "Java programming," the assistant can be biased towards that interpretation in subsequent queries.

6.2.3 Implications for User Experience and Reliability

Proper handling of ambiguous queries has a profound impact on both usability and trustworthiness:

- **User Experience:** By engaging in clarification rather than delivering arbitrary answers, the assistant fosters a more natural, human-like interaction pattern that reduces frustration.
- **Reliability:** In sensitive domains such as healthcare or legal support, disambiguation prevents potentially harmful misinterpretations.
- **Adaptability:** Robust ambiguity management allows the system to scale across diverse users and contexts, accommodating different levels of domain expertise without sacrificing accuracy.

Ultimately, ambiguity in natural language is an unavoidable feature rather than an anomaly. Designing conversational agents that explicitly recognize and manage ambiguity not only enhances technical robustness but also builds user trust by aligning responses more closely with the user's true intent [12].

6.3 Device Limitations (Tablet Performance, Internet Dependency)

The deployment of Large Language Models (LLMs) in real-world applications is often constrained by the computational and infrastructural limitations of the devices on

which they run. Unlike server-grade environments equipped with powerful GPUs and high memory availability, end-user devices such as smartphones and tablets face significant challenges in handling the resource-intensive operations required by modern LLMs.

6.3.1 General On-Device Limitations for LLMs

Running LLMs directly on consumer devices is still largely impractical for models beyond a few hundred million parameters. The reasons for this are manifold:

- **Processing Power** - LLMs require substantial parallel computation, typically relying on GPUs or TPUs. Mobile CPUs and integrated GPUs do not offer sufficient throughput for efficient inference.
- **Memory Requirements** - Even quantized models can easily exceed the RAM available on mobile devices. This restricts the size and complexity of models that can be deployed locally.
- **Energy Consumption** - The high computational demand leads to rapid battery drain and heat generation, which is undesirable in portable devices.
- **Model Updates** - Frequent updates or fine-tuning of models require downloading large files, which is inconvenient and impractical for end-users.

For these reasons, on-device deployment of large-scale models is typically replaced with a client-server paradigm, where the model resides in the cloud or on a dedicated backend server.

6.3.2 Tablet Performance and Capabilities

As mentioned in the previous chapter, in the context of the proposed system, the interactive totem is based on a tablet device. While modern tablets are capable of handling lightweight inference for smaller models (e.g., intent classification, keyword extraction, or basic embeddings), they fall short when dealing with large-scale reasoning and retrieval tasks. Along with the limitations already discussed in the *Evaluation of Local-Only Performance* section, there are also some specific constraints:

- **Hardware Constraints** - The tablet's CPU and GPU lack the necessary computational capabilities for real-time LLM inference, particularly when queries require contextual retrieval from a large knowledge base.
- **Limited Memory** - With typical memory ranging between 4GB and 8GB, the tablet cannot load full-scale models (billions of parameters), even in quantized form.
- **Thermal Performance** - Continuous high-load operations would lead to overheating, resulting in throttling and reduced responsiveness.

As a result, the tablet's role is primarily to provide an intuitive user interface, handle local preprocessing of inputs, and communicate efficiently with the backend server.

6.3.3 Internet Dependency and System Reliability

Given these constraints, the architecture depends on a stable internet connection to relay queries and responses between the tablet frontend and the backend server. While this design ensures access to powerful LLMs without overwhelming the tablet's resources, it also introduces potential risks:

- **Connectivity Issues** - In case of poor or unstable internet, the responsiveness of the system may degrade significantly.
- **Latency** - Network transmission adds delays, which must be minimized to maintain a fluid conversational experience.
- **Fallback Mechanisms** - To mitigate connectivity issues, lightweight on-device models or caching strategies can be employed, though these solutions only partially compensate for the absence of backend access.

In summary, while tablets provide a portable and user-friendly interface for interaction, their hardware and reliance on internet connectivity reinforce the necessity of a hybrid architecture in which the backend handles heavy LLM inference, and the frontend manages usability and accessibility.

7

Conclusions

This thesis has presented the design and development journey of a virtual assistant conceived for a hardware company, with the primary goal of assisting customers in retrieving detailed product information and availability directly through an interactive tablet application. The project was shaped by both technological opportunities and the practical constraints imposed by the deployment environment and the requirements expressed by the partner company.

The central challenge faced throughout this work was integrating advanced conversational capabilities into a device with limited computational resources. The initial exploration confirmed that running state-of-the-art Large Language Models (LLMs) directly on the tablet was impractical due to their high memory and processing demands. As a result, the solution adopted was to employ a hybrid architecture where the tablet functions as a client interface. At the same time, the heavy reasoning and natural language understanding tasks are delegated to cloud-based models. This design allowed us to balance usability with performance, ensuring fast responses and accurate retrieval of information, while still respecting the hardware limitations of the deployment platform.

Beyond the technological implementation, the project highlighted broader insights. The use of Retrieval-Augmented Generation (RAG) proved essential to align the assistant with domain-specific knowledge, enabling the system to provide contextually accurate answers based on product catalogues and technical documents. At the same time, sentiment analysis mechanisms were introduced to monitor and improve user experience, paving the way for continuous refinement of the assistant. These architectural choices underline how combining LLMs with structured retrieval pipelines can effectively bridge the gap between generic language capabilities and specialized business needs.

Nevertheless, several limitations remain. Dependency on a stable internet connec-

tion can reduce the reliability of the system in scenarios where network access is weak or unstable. Similarly, the reliance on cloud-hosted models introduces concerns about latency, costs, and data privacy, all of which must be carefully managed in real-world deployments. On the device side, the restricted computational power of the tablet limits the potential for offline fallback strategies and constrains the overall flexibility of the system. These limitations, while acceptable within the current scope, represent areas of improvement for future iterations.

In conclusion, this work demonstrates that deploying an LLM-powered chatbot in a resource-constrained environment is feasible when supported by a carefully designed architecture that offloads demanding tasks to the cloud and enriches responses with domain-specific retrieval. The system developed for this project stands as a proof of concept that combines practicality, technical robustness, and user-centred design. Looking ahead, promising directions for further development include exploring lightweight on-device LLMs, pre-trained on predefined answers, for partial offline support, improving manual and catalogue formats to improve the information embedding and retrieval, and investigating fine-tuned domain models to enhance accuracy and efficiency further.

Ultimately, the project underscores how the integration of modern NLP techniques into constrained devices requires not only technical innovation but also a balance between performance, usability, and practical deployment considerations. The experience gained offers valuable lessons both for the adoption of LLMs in customer-facing applications and for future research in making advanced language technologies more accessible and efficient across diverse platforms.

References

[1] Hussam Abdulla et al. “Chatbots development using natural language processing: a review”. In: *2022 26th International conference on circuits, systems, communications and computers (CSCC)*. IEEE. 2022, pp. 122–128.

[2] Bashaer Alsafari et al. “Towards effective teaching assistants: From intent-based chatbots to LLM-powered teaching assistants”. In: *Natural Language Processing Journal* 8 (2024), pp. 100–101.

[3] Abid Ali Awan. *Chroma DB Tutorial: A StepByStep Guide*. `https://www.datacamp.com/tutorial/chromadb-tutorial-step-by-step-guide`. 2024.

[4] Noam Chomsky and Marcel P Schützenberger. “The algebraic theory of context-free languages”. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 26. Elsevier, 1959, pp. 118–161.

[5] Lukas Dagne. “Flutter for cross-platform App and SDK development”. In: (2019).

[6] Sumit Kumar Dam et al. “A complete survey on llm-based ai chatbots”. In: *arXiv preprint arXiv:2406.16937* (2024).

[7] Matthijs Douze et al. “The faiss library”. In: *arXiv preprint arXiv:2401.08281* (2024).

[8] Aoran Gan et al. “Retrieval Augmented Generation Evaluation in the Era of Large Language Models: A Comprehensive Survey”. In: *arXiv preprint arXiv:2504.14891* (2025).

[9] Yunfan Gao et al. “Retrieval-augmented generation for large language models: A survey”. In: *arXiv preprint arXiv:2312.10997* 2.1 (2023).

[10] Yikun Han, Chunjiang Liu, and Pengfei Wang. “A comprehensive survey on vector database: Storage and retrieval technique, challenge”. In: *arXiv preprint arXiv:2310.11703* (2023).

[11] Diksha Khurana et al. “Natural language processing: state of the art, current trends and challenges”. In: *Multimedia tools and applications* 82.3 (2023), pp. 3713–3744.

[12] Hyuhng Joon Kim et al. “Aligning language models to explicitly handle ambiguity”. In: *arXiv preprint arXiv:2404.11972* (2024).

[13] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[14] Ggaliwango Marvin et al. “Prompt engineering in large language models”. In: *International conference on data intelligence and cognitive informatics*. Springer. 2023, pp. 387–402.

[15] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. “Sentiment analysis algorithms and applications: A survey”. In: *Ain Shams engineering journal* 5.4 (2014), pp. 1093–1113.

[16] Yelena Mejova. “Sentiment analysis: An overview”. In: *University of Iowa, Computer Science Department* 5 (2009), pp. 1–34.

[17] Shervin Minaee et al. “Large language models: A survey”. In: *arXiv preprint arXiv:2402.06196* (2024).

[18] Antonios Misargopoulos et al. “Building a knowledge-intensive, intent-lean, question answering chatbot in the telecom industry-challenges and solutions”. In: *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer. 2022, pp. 87–97.

[19] Humza Naveed et al. “A comprehensive overview of large language models”. In: *ACM Transactions on Intelligent Systems and Technology* (2023).

[20] James Jie Pan, Jianguo Wang, and Guoliang Li. “Vector database management techniques and systems”. In: *Companion of the 2024 International Conference on Management of Data*. 2024, pp. 597–604.

[21] Bhanu Teja Patibandla. “Network Latency in Large Language Models: Why It Matters and How to Optimize It”. In: (2024).

[22] Maite Taboada. “Sentiment analysis: An overview from linguistics”. In: *Annual Review of Linguistics* 2.1 (2016), pp. 325–347.

[23] Aakanksha Tashildar et al. “Application development using flutter”. In: *International Research Journal of Modernization in Engineering Technology and Science* 2.8 (2020), pp. 1262–1266.

[24] Mayur Wankhade, Annavarapu Chandra Sekhara Rao, and Chaitanya Kulkarni. “A survey on sentiment analysis methods, applications, and challenges”. In: *Artificial Intelligence Review* 55.7 (2022), pp. 5731–5780.

[25] Qinyuan Ye et al. “Prompt engineering a prompt engineer”. In: *arXiv preprint arXiv:2311.05661* (2023).

[26] Penghao Zhao et al. “Retrieval-augmented generation for ai-generated content: A survey”. In: *arXiv preprint arXiv:2402.19473* (2024).

[27] Wayne Xin Zhao et al. “A survey of large language models”. In: *arXiv preprint arXiv:2303.18223* 1.2 (2023).

Acknowledgments

My sincere thanks to *Loris Nanni* for their patience, understanding, and expertise, which were essential to the completion of this thesis.

I am also sincerely grateful to *Akkodis* for allowing me to carry out this project within their organization. Special thanks go to my colleagues: *Federico Monelli*, *Giorgia Castelli*, *Angelo Nasole*, and to my tutor *Simone Burigo Simoes*. Your collaboration, advice, and professional environment greatly enriched both my learning experience and this thesis.

To my family, I owe more than words can say. Your unconditional love, patience, and faith in me have been my most significant source of strength. Without your support, none of this would have been possible. You've been the most essential part of this journey, and I couldn't be more grateful. This also goes to my grandparents, who are not here to enjoy this moment with me, but I know they would be proud of me. A piece of this is also yours, and you will always be part of my life.

To my friends, thank you for always being there - for the laughs, the encouragement, the study sessions, the memories, and the moments we shared. You are part of who I am as a person, and you are the people I chose to share this adventure with. You contribute to making my life a little better every day, and this goes beyond anything I could ask for.

And finally, to my girlfriend, *Laura*, I am endlessly grateful for your love, understanding, and encouragement. Even though we knew each other not so long ago, it's like we met years ago. I couldn't be happier. We found each other in the perfect moment, and you helped me throughout this part of my life, rich with changes, important decisions, and key moments. Thank you for welcoming me into your life.