POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Next-Generation Cloud-Based Chatbots

Enhancing Customers Interactions Using Generative AI on AWS

Supervisors
prof. Riccardo Coppola

Candidato
Francesco Recchia

Internship Tutor
dott. ing. Davide Pezzolla

Anno accademico 2023-2024

Summary

This thesis proposes the development and deployment of a cloud-based chatbot on Amazon Web Services (AWS), tailored to enhance the online-shopping process with responses generated starting from real data of the customer. The research focuses on advanced Generative AI technologies, particularly the Retrieval Augmented Generation (RAG) methodology and the integration of Agents.
The primary objective is to enhance the customer's existing Question and Answer (Q&A) system, leveraging customer-specific data to contextualize and personalize the responses, thereby minimizing or potentially eliminating the need for human intervention. Moreover, the developed architecture enables the chatbot to interact with the system choosing the appropriate action to perform starting from the user prompt, including product search and shopping cart creation.
A series of experiments are conducted to determine the optimal solution to build a RAG system within the chatbot. Various factors are considered, including alternatives for storage and retrieval systems as well as the integration with the AWS environment. Special focus is placed on the design of a structured method to evaluate the chatbot's responses. The architecture is then built and deployed on AWS, integrating Agents.
The experiments demonstrates the effectiveness of the proposed RAG system, revealing the optimal configuration for the storage and retrieval phase. The Agent integration enables the chatbot to autonomously complete a set of predefined tasks in the online-shopping process.
In the end the developed cloud-based chatbot improves the efficiency and autonomy of online-shopping systems. Utilizing RAG and Agents, it enhances the response accuracy and reduce human intervention, ultimately improving customer experience and making it a viable option for e-commerce platforms.

Contents

Chapter 1

Introduction

The rapid evolution of online-shopping has transformed the way customers interact with businesses. While customers seek efficient solutions, they prefer personalized and human-like interactions over generic machine responses. The role of chatbots has become crucial to provide instant support and handle relatively simple tasks. In the online shopping context, assistance with product search and personalized offers and recommendations ultimately streamline the purchasing process. However, traditional chatbots fail to satisfy the customer expectations, due to their dependence on a predefined list of answers and limited ability to understand customer questions.

The advent of Generative AI introduced new possibilities for enhancing the chatbot's capabilities. The core of Gen AI is the ability of neural network models to understand and generate human-like text. This capability is known as Natural Language Processing (NLP). Modern chatbots leverage NLP to address the limitation of the precedent ones. The Retrieval-Augmented Generation (RAG) methodology, in particular, enables the system to search for relevant information in response to the user questions. The information are contained in a predefined knowledge base, consisting of documents, files, tables and more. This provides the chatbot the ability to generate relevant responses based off his knowledge, offering dynamic and personalized interactions that significantly improve the user experience.

The thesis focuses on the development and deployment of a chatbot with Generative AI capability within the AWS environment, designed to improve the shopping online experience. The proposed solution leverages the RAG technology and the interaction with intelligent Agents. Agents enable the chatbot to autonomously select and execute actions based on the user query. In this way, the customer experience is improved reducing the manual intervention and simplifying the purchasing process.

The primary objective of this work is to improve the existing Question and Answer (Q&A) system used in e-commerce integrating customer specific data. A

set of experiments were performed to identify the best components to build an effective RAG system within the AWS environment. In order to evaluate the results in a structured form different metrics were computed to assess the system performance and the quality of the responses.

The thesis is organized as follows:

- **Chapter 2: Background -** In this chapter is present the general study about the basics of the technology used. Focusing on Natural Language Processing (NLP), Embedding and Large Language Models (LLMs), which ultimately represent the foundation for Generative AI and Retrieval-Augmented Generation (RAG). It also discusses challenges and potential new directions for the research.
- **Chapter 3: Experimental Methodology -**The chapter contains a description of the research objectives followed by an introduction to the AWS environment and the more relevant services for the project. Finally, the custom RAG library to conduct the evaluation is introduced.
- **Chapter 4: Experimental Results -**Here the results of the experiments are presented. In order to identify the best configuration various comparisons are performed.
- **Chapter 5: Case Study | GDO -**Application of the chatbot to a real world application. A Web Application consisting of a back-end and front-end infrastructure. Additionally, a Generative AI stack includes the implementation of the RAG system and the Agent.
- **Chapter 6: Conclusion and Future Work -**Summary of the complete project and discussion on possible improvements.

Chapter 2

Background

2.1 Generative AI

Generative AI, also called gen AI, is a sub-field of artificial intelligence (AI) which is able to create, generate, original content, according to a user request. The content can be of a variety of different types such as text, image, video or audio, similar to the data used to train the model.

Gen AI relies heavily on AI, so it is crucial to understand what is AI and how it has evolved.

What is artificial Intelligence To explain what is artificial intelligence, we can break down the term: "intelligence" refers to the natural property which characterize a human being, the ability to think, while "artificial" indicates that this property is not obtained in a natural way but it was created. Thus, AI is a technology enabling a machine to simulate human intelligence [1].

Introduction to AI AI during the past decades has evolved in order to acquire human intelligence by increasing the ability to understand and reason like a human. We can think of it as a technology giving the possibility to train a machine and giving it all the necessary information to perform some human tasks.

2.1.1 AI architectures

Different kind of architectures were used to reach this scope, each dealing with different level of complexity and performing on different tasks. Starting from the simplest model unable to learn from data arriving to the Transformer which enable human language understanding and the possibility to perform a variety of tasks (e.g., Natural Language Processing, image generation and classification, speech generation)

Rule-Based Systems The first example of AI systems consisted of a set of predefined if-then rules. The input data is compared against the system's rule base and a rule is chosen according to the priority. After the rule selection, the system executes the corresponding action generating the output. This approach performs well only in specific domains where a set of precise rules can be formulated.

Machine Learning and Shallow models When computing power and data availability increased, new mathematical methods emerged introducing the concept of learning from data instead of relying on hard coded rules. New algorithm were designed to learn explicit pattern from data and make prediction using that knowledge.
Some of these are used for classification problems such as decision tree, random forest, Baesyan classifier and Support Vector Machine (SVM). In particular they rely on supervised learning (a starting dataset with label is provided and the model learns to map each input to the expected output) and are able to classify a given input as belonging to one of the possible output groups. But there are also algorithms using unsupervised learning(e.g, KMeans) tring to divide the given inputs in groups (clustering) without knowing the expected output.

Deep Neural Networks After some years the deep learning models arise and took AI on another level. Deep learning is a sub-field of machine learning dealing with neural networks, represented by multiple layers of interconnected neurons, able to capture features and information from data.
Neural network architecture can change according to the task to perform but the core of the model is always the same: an input layer, an output layer and one or more intermediate layers, each composed of a number of neurons with parameters and linked with weight. For neural network it is important the concept of Back-propagation, which enables the model to adjust the neurons weights during training in order to reach the expected output.
Having said that, we can have Convolutional Neural Network (CNN) used for image recognition and computer vision,Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) networks to handle temporal dependencies and NLP tasks, Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs) to generate images or text data.

Even if this models provided a significant advantage to language understanding and data generation they still lack in capturing the context of each word because they are processed one at a time loosing long-range dependencies.

Transformer Architecture With the paper "Attention is All You Need" of Vaswani et al. in 2017 a AI entered in a new phase. The new proposed Transformer architecture was able to address the limitations of the previous neural network models by processing the input data all at once.
In figure 2.1, the Transformer model architecture.

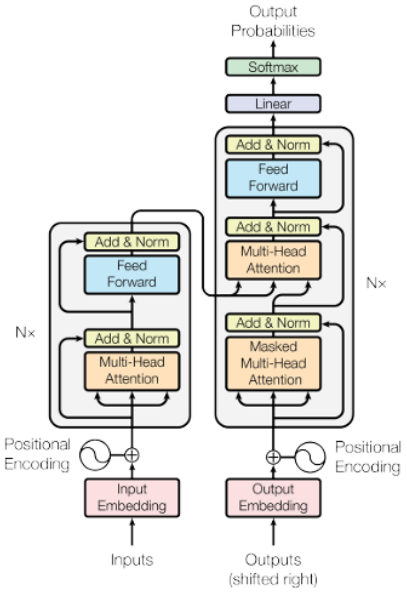


Figure 2.1: The Transformer model architecture (Vaswani et al., 2017).

The Transformer architecture consists of an encoder-decoder structure. Each composed on N identical layers containing a multi-head-attention mechanism and a feed-forward network. Input data is injected in the model after the embedding transformation (to obtain a numerical vector representation for each word) and the Positional Encoding, to keep also the information about word ordering. The Decoder generate the output sequence based on encoder's output and the previously generated tokens.
Now let's analyze each component more in details.

- **Feed-Forward Network:** FFNN is present in each encoder/decoder layer. It is composed of 2 dense network layers applying GeLU activation function[1]. Even if they are fast processing each word in parallel they lose the connection between words in the sentence.

[1]The Gaussian Error Linear Unit, given by the standard Gaussian cumulative distribution function.

- **Multi-head Attention layer**: Attention is used to calculate how much each word is important for all the other words in the sentence. For this scope it calculates a query vector, a key vector and a value vector by respectively multiplying each word representation to a query , key and value matrix. These are randomly initialized matrices and during the processing phase the values will be adjusted in order to minimize the loss function and produce the correct output.
  To calculate the attention of a specific word with respect to all the others the dot product of the query vector and all the key vectors is computed and then divided by the square root of the dimensions. Then the softmax function is applied to obtain a result between 0 and 1. In the end to obtain the word representation it performs the weighted sum of each value vector multiplied by the softmax result and sends each representation to the FFNN layer.
  In this way attention capture the relationships of the data but one layer is not enough to capture them all. For this reason multi-head attention is used to let a network learn more relationship path, using multiple different query, key and value matrices. Each set of matrices is called an attention head and will produce different query, key and value vector and eventually different word representations that will be combined at the end.
  In the decoder is present a different kind of attention layer, the masked one (MLM). This masks some of the inputs words and try to adapt the weights of the layer to complete the sentence with the correct missing ones. By the way the model is still not considering word order, even if it also provide context information for a word.

- **Positional Encoding**: to tell the transformer the relative position of each word it combines each word representation with the position in the sentence. In this way before being passed to the encoder and the attention layer the input representation will also contain information about word position.

- **Residual Connections**: Add & Norm. It adds the inputs of the attention layer to its outputs and then normalize values to stay in the 0-1 range. This is done to make the learning job easier for each layer by keeping track of the differences of the outputs with respect to the inputs after the transformation and it's applied also after the FFNN layer.

- **Linear and Softmax**: the Linear layer is used at the end to further processing the input data by applying a simple linear transformation for a classification problem, for example. Then the Softmax is applied to obtain the probability of each word to generate.

During training of the model the following steps are performed:

1. Input embeddings are calculated to represent the single word.

2. A vector representation of the position of the word in the sentence is added to the previous representation obtaining new embeddings after the Positional Encoder.

3. Vector embeddings reach the Encoders and here are processed to obtain as output a new vector representation of each word reflecting also the context of the sentence it is used into and the relationships with other words.

4. The expected output sentence is sent to the Decoder and an embedding representation is obtained after the Positional Encoding.

5. The masked multi-head Attention layer process the sentence and adjust the model weights to correctly predict the masked word and output the query vector to the next attention layer. This will also receive the key and value vector from the output of the Encoder.

6. The output embedding of the decoder is passed to the linear layer to map it to the word vocabulary and then the Softmax function is calculated.

7. Now the model compare the expected output with the obtained output and calculate the loss, back propagating the error to adjust the model weights.

Other types of Transformer architecture have been developed over the years. For example BERT (Bidirectional Encoder Representations from Transformers) is an autoencoder model (encoder-only transformer) trained using bidirectional representation, masked language modeling (MLM) and next sentence prediction (NSP). It is used, for example, for classification tasks and can generate embeddings of words useful for semantic similarity (more details in the Embedding chapter). Then GPT (Generative Pre-trained Transformer) is an autoregressive model (decoder-only transformer) is trained to learn statistical language representation by predicting the next token given the previous ones. This model is the standard for generative tasks.

The advancements in Transformer based models like BERT and GPT have impacted many AI applications and from now on i will focus on NLP.

2.2 Natural Language Processing for Generative AI

The main objective of a gen AI model is to generate text coherent with the context and similar to the text a human can produce. In order to do this the model needs to understand the human language and learn the meaning of each word to use it in the right context. Thus, we would like to give computers the ability to understand and communicate with human language and this was possible using the Transformer architecture.
This process can be referred as NLP (Natural Language Processing).

NLP NLP is based on computational linguistics, a discipline of linguistics that uses data science to analyze language and speech performing syntactical and semantical analysis, and on machine learning / deep learning. It can be seen as composed of two different parts: Natural Language Understanding (NLU) and Natural Language Generation (NLG). The first one focusing on determining the meaning of a sentence, the second enabling the computer to produce text.

There are three different approach to NLP that have been applied using the latest architecture available for the years:

1. Rule-Based NLP: based on simple if-then decision trees.

2. Statistical NLP: relying on machine learning it extract, classify and labels text. Then assigns a statistical likelihood to each possible meaning of those elements. To do this it introduced the concept of vector representation of a word.

3. Deep learning NLP: it uses neural network models and huge amount of text data to increase accuracy. We can consider different models:

   - Sequence-to-Sequence (seq2seq) based on recurrent neural networks (RNN)
   - Transformer using self-attention (BERT)
   - Autoregressive trained specifically for text generation (GPT)
   - Foundation models prebuilt to support NLP tasks

NLP Tasks There are several NLP tasks that involve understanding, generating, and processing human language. Some of them are:

- Tokenization: Breaking down a text into smaller units, usually words or subwords (tokens).

- Named Entity Recognition (NER): Identifying and classifying named entities like person names, organization names and dates in text.

- Sentiment Analysis: Determining the sentiment expressed in a piece of text (positive, negative, neutral).
- Machine Translation: Automatically translating text from one language to another.
- Part-of-speech tagging: Determining which part of speech (noun, verb, adjective) a word or piece of text belongs.
- Text Generation: Produces text that's similar to human-written text.
- Question Answering: Generating answers to natural language questions.
- Summarization: Generating concise summaries of longer texts.
- Speech Recognition and Synthesis: Converting spoken language into text (speech recognition) and text into spoken language.

2.3 Embeddings

One problem to solve for NLP is how to use textual words in deep learning model like neural networks. The solution relies in embeddings. Embeddings are numerical multi-dimensional vectors representing text, image, video and audio where the distance and direction between vectors reflect the similarity and relationships among the corresponding words. The closer two vectors are the more similar are the two words, for example, in semantic meaning. Using this representation it is possible for a simple or complex model to learn the human language.

During the years many approaches were tried to create the best numerical representation of a text maintaining the semantic meaning and without consuming too much resources.

2.3.1 Traditional Embedding Strategies

One-Hot Encoding *Description*: This method considers categorical data and at first assigns a numerical value for each different category. After that it represents each categorical value as a binary vector where only the element in the position corresponding to that category is "hot" (set to 1), while the others remain "cold" (or, set to 0).
In figure 2.1 we can see an example applied to different categorical values of the column *Colors*.

Colors	Numerical	Red	Green	Yellow
Red	1	1	0	0
Green	2	0	1	0
Red	1	1	0	0
Yellow	3	0	0	1

Table 2.1: One-Hot Encoding of Colours

In this way machine learning algorithms, such as a decision tree, can work directly with categorical data mapped to binary vector. The resulting vectors are easy to interpret and prevents the model from interpreting the data as ordinal (having an inherent order).

But it's pretty clear that dealing with many unique categories, like words in a sentence, can significantly increase the size of the feature space and therefore it is not practically applicable. Also, using this method, the semantic relationships and similarities between words are lost. For this reason similar words are represented as completely different ones, making it difficult to handle tasks requiring understanding of the language.

Bag of Words (BoW) *Description:* It considers different documents where each unique words is seen as a separate dimension in the vector space. Given n

different words the document will be represented by a vector of dimension n where the value of each dimension is equal to the frequency of that dimension's word in the document.

Consider for example the following documents:

- Document1 = *"Red Yellow White Red"*
- Document2 = *"Yellow Green Black"*

The vocabulary is defined as {*Red, Yellow, White, Green, Black*}. The vector representation of each document is shown in Table 2.2.

Document	**Red**	**Yellow**	**White**	**Green**	**Black**
Document 1	2	1	1	0	0
Document 2	0	1	0	1	1

Table 2.2: Bag of Words (BoW) Encoding of Documents

Despite being a versatile, easy to implement solution providing interpretable results, this technique does not consider the order of words inside documents nor the word correlation. For this reason semantic information are lost. Moreover it is not able to handle the case of words having different meaning in different context. In the end, also in this case, the results can be big sparse vectors when the size of documents increase which ultimately can create problems for some machine learning algorithms.

TF-IDF (Term Frequency–Inverse Document Frequency) *Description:* TF calculate how frequently a word appears in a document.

$$\mathrm{TF}(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

IDF measures how important a word is, giving less weight to word appearing frequently but less important (e.g,"a", "is", "the").

$$\mathrm{IDF}(t) = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents with term } t}\right)$$

Then TF-IDF is calculated for each document multiplying the TF and IDF value for each word. In this way it is possible to calculate how each word is relevant to each document.

Consider for example the following documents and calculate the TF-IDF for each word 2.3:

- Document1 = *"Red Yellow White Red"*
- Document2 = *"Yellow Red Green Black"*

Document	Term	TF	IDF	TF-IDF
Doc1	Red	0.5	0	0
Doc1	Yellow	0.25	0	0
Doc1	White	0.25	0.693	0.173
Doc2	Yellow	0.25	0	0
Doc2	Red	0.25	0	0
Doc2	Green	0.25	0.693	0.173
Doc2	Black	0.25	0.693	0.173

Table 2.3: TF, IDF, and TF-IDF values for each term in both documents

In this way we can say that, for example, the word *White* is as relevant to *Doc1* as the word *Green* is relevant to *Doc2*.

With this method it's easy to identify important terms in a document to understand what is the document about and helps differentiating between common and rare terms. It is scalable and language independent. But it ignores word order, it only measures term frequency and does not consider the meaning of each term so it's difficult to recognize synonyms and similar words. Also, it can result in very high-dimensional vector as the number of words increase.

2.3.2 Neural Network-Based Embedding Strategies

Strategies to represent a word as a vector presented up to now all have some limitations: the output are sparse vectors wasting lots of memory and they do not take into account the context and semantic meaning of words. For this reason *Embeddings* were introduced to represent words using dense vectors and maintaining the relationship of similarity also in the embedding space. And this was possible thanks to the usage of Neural Network.

Word2Vec *Description:* It takes the One-Hot encoded words and, considering the context, generates the *embeddings*(vector representation). W2v first define a window size of n words and give this in input to a neural network with just one hidden layer. This neural network's input and output layer are typically large, representing the vocabulary size, and the hidden layer dimension is the desired embedding size.

The model will try to predict a target word given the context of n words (Continuous Bag of Words version) or to predict the target context of *n-1* given one word (Skip-Gram version).
Then the models weight are adjusted according to the possible error signal sent by the network if it failed to predict the target word or context words.
After that, the window is shifted and a new group of words is processed by the network.

In this way the model is trained on a large corpus of text and learns, using a supervised task, to correctly associate similar words and remembering the context they are used in. The result for each word is an *embbedding*, with a number of dimensions equal to the size of the hidden layer, such that in the vector space similar words are placed close to each other (to calculate the distance the cosine-similarity metric can be used). Also, embeddings are able to represent relationships and associations between words. For example in the vector space, words representing countries (e.g., Italy, Spain, France) will be close to each other and the same happens for words representing cities (e.g., Rome, Madrid, Paris). Moreover the equation Rome-Italy + Madrid will return Spain, since embeddings remember relationships between words learned during training.

Still this technique has some limitations: a word can have more than one meaning depending on the context, but it has always the same vector representation; the fixed context window size makes it difficult to understand more complex long-range dependencies; to perform well a huge number of words is needed and this can affect memory and computation resources. The last one can be solved with optimization techniques such as Negative Sampling and Hierarchical Softmax(to reduce computational complexity during training).

FastText
Description: It is an extension of Word2Vec developed by Facebook to handle out-of-vocabulary words and understand the different structure of words (e.g, verb tense, plural, prefix and suffix, compound names).
The first step consists in breaking down words in *n-grams* (subwords) and then performing the same training of the Word2Vec using the skip-gram or CBOW model. In this way it is possible to obtain the embeddings for each *n-gram*.

For example using *n=3* (trigram) the word *playing* will be split in *"pla", "lay", "ayi", "yin", "ing"*. Embeddings are gerated for each sub-word and the sum of this will be the vector representation of *playing*.

Using this approach the neural network model is trained to predict words based on the *n-grams* it has seen and not only on the target words. This results in a better embedding representation of words not seen during training and also increase the embedding precision by capturing the morphology of each word.
But this strategy results in an increase of model size, resource consumption and training times.

ELMo (Embeddings from Language Models):
Description: This strategy was published in 2018, "Deep Contextualized Word Embeddings", and for the first time it was possible to generate different vector representation for the same word used in different context.

The first step consists in splitting down words into characters and inputs them to a Convolutional Neural Network to obtain a vector representation. It is the same approach of FastText to capture morphological meaning of words and handling words that were not seen during training, but using character-level tokenization to increase robustness and spell checking.
After that, a bidirectional Long Short-Term Memory (LSTM) neural network is used to process text reading from left to right and right to left capturing context from both directions: for each word the LMST keep a forward and backward hidden states. These states, at the end, are combined together to obtain the final embedding representation.

The main advantage of this strategy was to obtain embeddings which are context sensitive. For example the sentences:
I read a book
Did you book the room?
Contains the same *book* word used with different meaning and therefore their embeddings will be different.

2.3.3 Statistical Embedding Strategies

GloVe (Global Vectors for Word Representation): *Description:* It is an algorithm developed by researchers at Stanford to capture global statistical information about a corpus of text.
At first it computes the word-word co-occurrence matrix. This is a large sparse square matrix with dimensions equal to the number of unique words in the corpus of documents. For each entry in the matrix (combination of two words) it counts the number of times this two words appear together in the corpus of documents according to a predefined window size. This window specify how far the words can be on the same documents to say they appear together.

Consider for example the following documents and calculate the co-ocurence matrix 2.4:

- Document1 = *"Red Yellow White Red"*
- Document2 = *"Yellow Red Green Black"*
- Window = 2

	Red	Yellow	White	Green	Black
Red	0	2	1	1	1
Yellow	2	0	1	1	0
White	1	1	0	0	0
Green	1	1	0	0	1
Black	1	0	0	1	0

Table 2.4: co-occurence matrix for GloVe

We can see that using a window size equal to 2 "Yellow" and "Green" increment the counter while "Yellow" and "Black" not.

In the end to compute the vector representation of each word an objective function is computed:

$$\text{Minimize } \sum_{i,j} f(X_{ij}) \left(\mathbf{w}_i^T \mathbf{w}_j + b_i + b_j - \log(X_{ij})\right)^2$$

where:

- $\mathbf{w}_i$ and $\mathbf{w}_j$ are word vectors,
- b_i and b_j are bias terms,
- X_{ij} is the co-occurrence count of words i and j,
- $f(X_{ij})$ is a weighting function that adjusts the influence of each co-occurrence count.

The solution is found through iterative optimizations and at the end for each word we obtain a dense vector representation.
These vectors encode semantic and syntactic information and allow to find similarities between words and to understand global word usage patterns.
By the way, it requires more memory to store the co-occurrence matrix and doesn't perform well with small corpus of text.

2.3.4 Transformer-Based Embedding strategies

BERT (Bidirectional Encoder Representations from Transformers) *Description:* First introduced by Delvin et al. in 2018 this model is composed of multiple layers of bidirectional transformers and it is characterized by two dinstict phases: the pre-training and the fine-tuning.
During pre-training Masked Language Modeling (MLM) is performed by randomly masks some of the input tokens and trying to predict them. Also Next Sentence Prediction (NSP) is used to predict whether a given sentence follows another sentence. By doing this the model is able to understand the context of words and sentence relationships.
After this the model is fine-tuned adding a simple output layer to be specialized on specific tasks.

To generate embeddings, at first first the input is split in words (tokens) and two special tokens are added at the beginning (CLS) and at the end (SEP). Then a vector representation of each token is obtained together with Segment and Position embeddings. The first indicating the belonging sentence and the second the position in the sentence. At this point the input is fed to the model which, at the end will contain the vector representation of each word inside the hidden layer of the model.

In this way it was possible to obtain full context aware embeddings, able to represent word and sentence relationships and to handle words with different meanings.
Also it provides a way to start with general context trained model and fine-tune it for specific tasks using only the data necessary for that task making it versatile and efficient.

2.4 Large Language Models

Large Language Models (LLM) are a class of AI models based on architectures like Transformers, pre-trained on a large amount of text data and fine-tuned for specific tasks. They provide many improvements for NLP and AI applications.
Using LLMs is possible to generate coherent and relevant text for tasks like text completion, question answering and summarization. They are very versatile since it is possible to start from an LLM and fine-tune it using a relatively small dataset of labeled data for different applications and domains. Nevertheless, they require lots of computational power and memory and are expensive to deploy. Also, LLMs rely on the data they were trained on which can become outdated and resulting in inaccurate outputs.

Applications of Large Language Models LLM have a wide range of applications in many different fields and industries.
Chatbots and virtual assistants can respond instantly and accurately to user questions enhancing user experience. In the Educational field they can be used to support students learning, in customer service to speed up the process of information retrieval and increasing efficiency. Their ability in NLP can contribute in creating new original contents and having meaningful conversations with users. They can be used to summarize texts or write new articles or translate conversations in real time or provide personalized suggestions in e-commerce platforms.
Moreover they can increase human productivity as virtual assistants answering mails or generating code to help developers and much more.

Optimizing LLM Scaling laws describe the trade-offs between model and dataset size for a fixed budget of computational resource [4]. These laws states that is possible to obtain better performance by increasing the number of words used to train the model or the number of model parameters. By the way increasing the size of the model keeping the same dataset size will result in an increase of the computational resources required.
The Cinchilla paper compared model performance of various model and dataset size combination [5]. According to this study they stated that many of the big model on the market are over-parameterized and could achieve the same performance by just increasing the dataset size, reducing the model size.
In particular the paper claims that the optimal dataset size (expressed as number of tokens) has to be 20x the number of model parameters.

2.4.1 Fine-Tuning

Fine-tuning allows to specialize the LLMs capabilities and optimize its performances on narrower and specific dataset. This in particularly important since training of a model is a significant cost in term of resources and time. With fine tuning it is possible to leverage pre-trained LLM models and their ability to understand and generate text and use them for domain specific tasks. Using a smaller specific dataset it is possible to increase the accuracy of the response of the LLM model without consuming too much resources.
There are different types of fine-tuning:

- **Supervised fine-tuning:** The model is further trained on a labeled dataset specific for the target task such as text classification and entity recognition.
- **Few-shot learning (instruction fine-tuning):** When is not easy to find/create a labeled dataset it is possible to provide few examples of the required task to the input prompt.
- **Transfer learning:** In case the desired task to perform is different from the one the model was trained on.
- **Domain-specific fine-tuning:** This type of fine-tuning is used to adapt the model to a specific domain. A dataset containing information about a particular domain is used to train the model to understand and generate domain specific language.
- **Fine-tuning with reinforcement learning:** The model learns to generate outputs that maximize the reward received. This strategy is used when a labeled dataset doesn't exist and human evaluate manually output of the model giving it rewards or penalties.

Parameter-Efficient Fine-Tuning (PEFT) In general Fine-tuning is a process that starts from the weights of the pre-trained model and update them, but not all of them, since it would require lot of time and resources and could cause the problem of catastrophic forgetting (loss of model's core knowledge). The technique called parameter-efficient fine-tuning (PEFT) aims to reduce the number of trainable parameters decreasing the computational resources without impacting on performances.
It is possible for example to use adapters. These are new layer, trained for specific tasks, added to the end of the transformer model keeping the other network parameters stable.

Low Rank Adaptation (LoRA) Another approach to reduce the costs of fine-tuning is to apply a reparameterization on high dimensional matrices to capture the low-dimensional structure of model weights and obtaining lower rank matrices. Also, QLoRA further reduces the computational complexity by quantizating the transformer model.

Limitations One of the primary issue of fine-tuning LLM is that they need high-quality, domain specific dataset which could be difficult to find or create. Using low-quality dataset can introduce biases into the model and decrease quality and performances.
Another important limitation of fine-tuned models is that their knowledge is limited only to data used to train them (maybe old). The only way to be up-to-date is to fine-tune again the model with the new data resulting in increasing complexity and consumption of resources.

2.4.2 Retrieval-Augmented Generation (RAG)

This approach was developed to address some of the fine-tuning challenges. In particular using this method is possible to retrieve relevant documents from large corpus of text (also the Internet) and generate text, based on those documents (dynamic information retrieval). Documents can be easily updated and this allow the model to always have access to up-to-date information. Also it provides a effortless strategy to make the AI model aware of proprietary data and can increase accuracy by referencing only this data in the response.

Implementation In Figure 2.2, the RAG architecture.

1. Relevant documents are stored in some data store system in any possible format or also the Internet can be considered a sort of place where documents are stored.

2. All the documents are sent to so a machine to be pre-processed.

3. (a) The documents are split in chunks. There exists a variety of different chunking strategy such as Recursive Character Text Splitting or Semantic chunking.

   (b) Each chunk is then converted to an embedding representation using an Embedding model (described in previous section).

4. The vector embeddings are stored in a Vector Storage. This are data storage specifically designed to wok with vectors and providing different functionalities like indexing and auto-update. One of this is FAISS [2].

5. When the documents are correctly saved the user can perform a query to a model. This starts the retrieval phase.

6. The query is converted to an embedding representation and arrive to a search engine (provided by the vector storage).

7. The search engine using a similarity function [3] search for documents in the vector store relevant to the given query.

8. The vector store returns one or more selected documents.

9. The search gives back the documents to the generative model.

10. (a) The documents are added to the original query in the form of context (a precise prompt template is necessary).

    (b) The text generation model receives the augmented prompt and respond to the query referencing information contained in the retrieved documents.

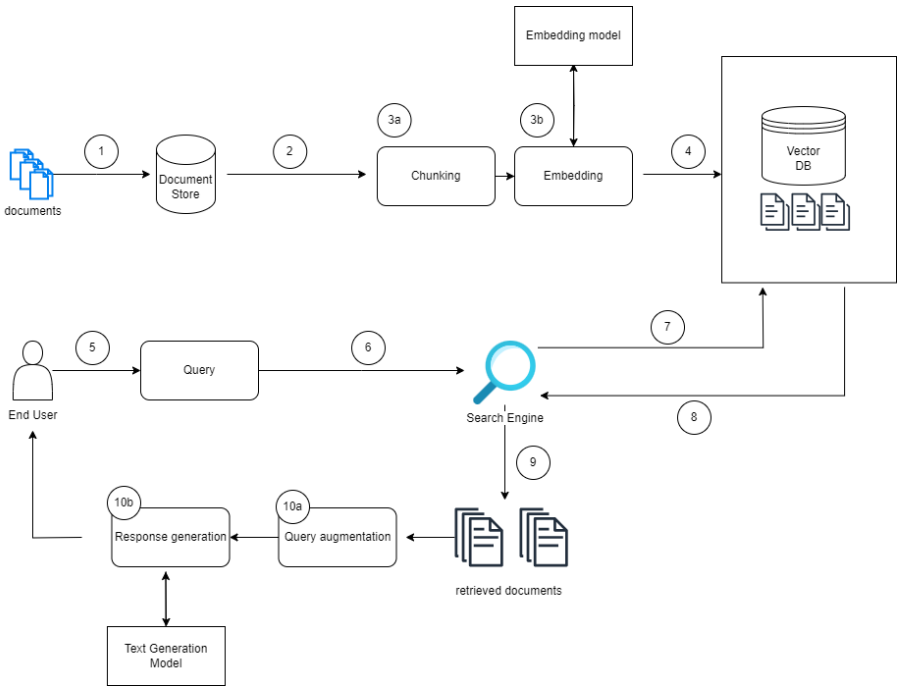


Figure 2.2: RAG system architecture.

RAG technique can be useful in a variety of task such as question answering and dialog systems like chat bots. It speeds up the process of specialization of the AI model with respect to fine tuning and can also provide evidences for the responses. Nevertheless, it suffers of the same problem of fine-tuning related to data quality: poor quality or incomplete data can introduce biases in the responses.

Prompt Engineering Even if a LLM is trained on a big dataset and has lots of parameters it is not going to perform well on generative tasks if the prompt (input to the model) is not good enough. Prompt structure are model specific but are always composed of: Instruction, the piece of text describing the task to perform, and Context, relevant information passed to the model helping him to solve the task.
Prompt Engineering is a new skill focused on effectively apply generative AI models to specific tasks and use cases. Set up properly a prompt requires many iterations on the model responses, adjusting the prompt piece by piece. In fact, for a model is possible to learn from the context before generating text using the so-called *in-context learning*.
There are different techniques based on passing one or more example in the prompt to generate text based on the examples provided. One-shot inference adds one instruction in the prompt with a possible query and generation format. Few-shot

[2]FAISS is an opensource library for efficient similarity search and clustering of dense vectors

[3]Similarity functions are used to determine the proximity of two vectors in the embedding space. Some of these are Euclidean distance and Cosine similarity.

inference include more than one example. While zero-shot does not add any example to the prompt.
The generated text resulting from a well structured prompt, completed with possible examples, are far more accurate than the ones generated with zero-shot inference. This increase the model accuracy without modifying the model itself, which is a valuable result.

Agents RAG is useful for tasks where we want the model to acquire specific knowledge from pre-saved documents before generating text, but it is also possible to give the model the ability to think and perform the steps necessary to solve a task using an Agent. This piece of software is able to create a step-by-step plan to solve the task performing a RAG workflow and collecting responses from data lookup and/or invoking API. These responses are used to augment the prompt and generate the final result.
The Agent often use a ReAct framework combining chain-of-thought reasoning with action planning providing prompt with a structure containing: *Question*, the user requested task, *Thought*, reasoning step to decide how to solve the problem, *Action*, invoking API or retrieving documents and *Observation*, the result after the action. The Observation step is used to check if the task has been completely solved, otherwise restarting the process adding in the context the previously found results.

2.5 Challenges and new directions for LLMs

Large Language Models (LLMs) have made significant improvements in natural language processing and generation over the last year. Despite these advancements, there are still significant challenges to address to improve efficiency, scalability and reliability.
Many limitations are caused by the cost and resource requirements for this models to always perform better. But also general concerns regarding the response interpretability and ethical considerations.

Computational Cost and Resource Requirements The training cost increase as the size of the model and the dataset and require specialized hardware and significant energy consumption. This represent a problem in terms of cost and also scalability.
Models should be able to handle large volumes of concurrent requests and for this reason the computational resource requirement increase. Also they should achieve real-time performances and maintain low latency, which is important for applications like chatbots.

Training Dataset It is crucial to use dataset big enough to increase model performances but the data quality is important as well. The ability of the model to generate coherent and reliable text is based on the data used to train it. If the dataset is not big enough or the data are not so diverse and maybe incorrect, responses will be characterized by the same biases of the dataset. For this reason it is important to evaluate the quality of the data before feeding them to the LLM. Moreover in the last period techniques to generate new data have been studied to increase the performance of the models. This *Syntetic data* can increase diversity in the dataset but they need to preserve data privacy and security.

Multimodal Models NLP is not the only application of LLM. Multimodal models can combine different data format like text, images and speech to increase the understanding of the model and address more complex tasks. It is possible to generate text from speech or videos from text for example.
These are based on the Diffusion architecture, trained using unsupervised learning and are able to capture the relationships and interactions of different elements also over time.

Model Evaluation Evaluating a model generated text without human intervention is not easy since it is non-deterministic. There exists different evaluation metrics to evaluate models on specific tasks like Rouge and Bleu. The first focusing on summarization tasks and the second on the translation ones. Both of them consider the *n-grams* in which input and output can be split and Rouge calculate the recall while Bleu the precision.
For more general tasks the evaluation of *n-grams* matching is not suitable. One

approach it is being studied is *LLM evaluating LLM*. It can be useful when in the dataset is present an expected output, like an expected answer to a question, generated by a human, and asking the model to evaluate how much the model generated response is similar to it. Some libraries exists like RAGAS, which also provide metrics to evaluate the RAG workflow.
In general, metrics like GEVAL can be used to evaluate a model output without any ground truth and there are also metrics to evaluate Hallucinations, Biases and Consistency of the generated text.

Chapter 3

Experimental Methodology

3.1 Objectives

The objective of this thesis is to explore and develop innovative solutions in the field of Artificial Intelligence, with a particular focus on Generative AI. The core project involve the creation of a shopping assistant for a leading enterprise in the large scale retail sector (GDO) which will enhance the customer's capability to interact with the main site.
This project leverages the RAG technology and intelligent Agents to deliver precise and contextual answers to customer inquiries. All the development is carried out on the AWS (Amazon Web Services) platform, which offers specialized services to support the implementation of these technologies within the chatbot.

One of the central goals is to harness the power of the RAG technology to obtain more relevant answers, based on a predefined knowledge base [1]. In contrast to traditional chatbots, based on predefined answers or simple retrieval mechanism, RAG enables the retrieval of the documents related to the given question and generate a response using the acquired knowledge.
For the RAG technology to function effectively, several steps are necessary, including indexing documents in the vector storage, configuring retrieval parameters, and selecting the appropriate text generation model. Therefore, the first part of my thesis project focused on developing a library designed to speed up the creation of the RAG pipeline and to evaluate it. This custom library offers a comprehensive overview of various implementation strategies for the technology, focusing on different vector storage options for embedding documents in the knowledge base. It also explores diverse chunking strategies to enhance storage and retrieval efficiency and analyzes the results of different text generation models and retrieval parameters, such as the number of documents retrieved per query.
Additionally it offers a method to evaluate the outcomes using the RAGAs (Retrieval-Augmented Generation Assessment System) library, enabling the identification of

[1] A database designed to store and retrieve complex documents, especially in support of artificial intelligence systems

the most effective solution.

Moreover, since the AWS platform offers two distinct services to implement this kind of solution, a comparative study was conducted to determine the most suitable approach for predefined scenarios. Specifically, the focus was on identifying the *break-even point* to guide the selection of the optimal service for each specific situation.
In summary, the objectives of this thesis are multi-faceted, encompassing the development of a state-of-the-art chatbot, the creation of a custom library for optimizing the RAG technology, rigorous evaluation of the system quality and the identification of the most effective service options.

3.2 AWS Ecosystem

Amazon Web Services (AWS) provides a robust and versatile environment for developing advanced artificial intelligence (AI) solutions, including chatbots. Among its extensive suite of services, AWS Bedrock and AWS SageMaker are the way to go for implementing Generative AI (Gen AI) technologies.

General Services In order to complete the project and design the library many supporting services have been used, including:

- S3: The S3 service offers the possibility to define *Buckets*, a container for objects or files. It is possible to organize it in sub folder and to store documents of different format for long period of times.
- Lambda: This enables developers to run function's code in a server-less manner. The code is executed in a pay-per-use basis on an high available infrastructure, without the need to specify any compute resources.
- CloudFormation: This service helps to model and configure AWS resources dealing with creation, update and delete situations. Related resources are organized in *Stacks* to efficiently manage them together.
- Elastic Container Register (ECR): A container image registry to easily store, share and deploy docker container images.

3.2.1 AWS Bedrock

AWS Bedrock is a service designed to simplify the creation and deployment of generative AI applications. It offers:

- Pre-trained Models: there are many high quality models, some proprietary other open source, already trained and ready to be adapted to specific situations accelerating the development process.
- Scalability and Management: it manages the underlying infrastructure, ensuring scalability and reliability. Therefore the developer can avoid to focus on infrastructure related problems.
- Bedrock KnowledgeBase: is a service relying on the OpenSearch (another AWS service) vector storage, which automatically index documents and configure the retrieval mechanism, also providing an automatic sync functionality to handle document updates.
- Bedrock Agents: the default Bedrock implementation of an Agent, providing the Orchestration prompt and enabling the creation of action-groups to tell the LLM which API (defined in precedence using an openApi model) to invoke.

3.2.2 AWS SageMaker

AWS SageMaker is a comprehensive service that provides tools for building, training, and deploying machine learning models. It is particularly valuable for developing custom AI solutions, including those required for advanced chatbot functionalities. It offers:

- Integrated Development Environment: an integrated development environment (IDE) with built-in Jupyter notebooks, allowing developers to experiment with machine learning models.
- Training and Tuning: SageMaker simplifies the process of training and tuning models with automated machine learning (AutoML) capabilities and hyperparameter optimization.
- Model Deployment: SageMaker provides streamlined deployment options. This includes the deployment of the model on callable endpoints for real-time inference.

3.2.3 Break Even Point

The study to determine the optimal service for each use case began with an analysis of the functionalities and costs of the available services.
This was followed by a comparison of the RAG pipeline performance when implemented using models from AWS Bedrock and AWS SageMaker for text generation. The goal was to understand the differences in performance between the models provided by the two services.
It is possible to consider different factors impacting the choice of the model:

- Costs: Bedrock offers a "pay-per-use" option based on the number of generated tokens so that the user only pays when he calls the API to generate the response.
  Sagemaker instead needs a deployed endpoint to call the model, and the user pays for the its compute capacity in terms of virtual hardware performances. Bedrock also offers a Provisioned throughput solution paying in advance for usages of a long period of time.

- Vertical and Horizontal Scaling: SageMaker supports automatic scaling, whereas Bedrock does not. In SageMaker, scaling involves selecting a CloudWatch metric, defining a target value, scaling instances accordingly, and setting cooldown periods to manage the scaling rate.

- Security: SageMaker offers complete control over data, with protections for data both at rest and in transit.
  Bedrock gives limited control over infrastructure, as AWS manages and stores custom models and data, which might reduce visibility and control.

- Fine-tuning: both the services offers the possibility to fine-tune (Domain adaptation and Instruction Based) the models.

- Models Availability: as of this project SageMaker offers much more models than Bedrock. Also many of them are fine-tunable.

- Customization: Bedrock offers pre-trained foundation models to rapidly develop application using the Gen AI technology, but it is also possible to customize these models to specialize them for less common scenarios. Nevertheless, Sagemaker is a better choice when we need to built/train/deploy specialized models, providing extensive options and control over the machine learning cycle.

Conclusions AWS Bedrock and AWS SageMaker provide a powerful and integrated environment for developing advanced generative AI solutions, including chatbots.
AWS Bedrock is an ideal choice to quickly develop applications that integrate the Gen AI technology using the already trained foundation models. It is better suited for use cases where model customization is not necessary like customer support

chatbots, requiring just the NLP support offered by base foundation model.
Instead, AWS SageMaker excels in situations requiring more control over infrastructure, model training, customization and deployment. Use cases demanding highly specialized models, such as those in healthcare, would benefit the Sagemaker adoption and will help data scientist and machine learning engineers, by providing a flexible environment to experiment.

3.3 Custom RAG library

In a RAG pipeline is possible to consider the following core components:

- Chunker: to split input documents, when predefined characters are encountered, obtaining relatively small pieces of text called *chunks*.
  The *chunks* are further processed by calculating the embedding representation to efficiently compare them.
- Vector Storage: It is a database to store the embeddings and efficiently compare them. It offers indices optimized for vector search to retrieve documents in a fast way.
- Retriever: The component used to retrieve relevant documents to a user query and generate a response based on those.

During the project i developed a library in Python offering different implementation of these components to compare their quality and performance. This library is flexible and can be customized by adjusting different parameters. It can be useful to speed-up the RAG pipeline creation or evaluation and it integrates seamlessly with the AWS environment.
The library offers three different scripts: indexing, testing and evaluation. Using the SageMaker Processing Job functionality, it is possible for users to define the environment's characteristics, such as CPU and RAM, where a *job* is launched to execute the desired code image. Indeed, a docker image is built and stored in the AWS ECR (Elastic Container Registry) for each script. The *job* is enabled to read and execute it in the predefined environment by passing all the necessary configuration parameters.

3.3.1 The Dataset

In order to conduct the experiments to evaluate the quality of the RAG pipeline it was necessary to find a Dataset containing a reasonably big amount of documents. The Sec-10-Q was chosen, where the PDF source documents contain financial information about different tech companies: AAPL, AMZN, INTC, MSFT, NVDA; divided according to the financial quarter they refer to.
It also offers a list of 195 questions about them, labelled with human answers [2]. Therefore, it was possible to compare the model generated answers with the human ones, and determine how similar they are. After all, the final objective for a generation model would be to replicate a human answer in the best possible way in terms of correctness and coherence.
An extract of the q&a table is shown in figure 3.1.

Question	Source Docs	Question Type	Source Chunk Type	Answer
How has Apple's total net sales changed over time?	"AAPL"	Multi-Doc RAG	Table	Based on the provided documents, A…
What are the major factors contributing to the change in Apple's gross margin in t…	"AAPL"	Multi-Doc RAG	Text	In the most recent 10-Q for the quarte…
Has there been any significant change in Apple's operating expenses over the rep…	"AAPL"	Multi-Doc RAG	Table	Yes, there has been a change in Appl…
How has Apple's revenue from iPhone sales fluctuated across quarters?	"AAPL"	Multi-Doc RAG	Table	The revenue from iPhone sales for A…
Can any trends be identified in Apple's Services segment revenue over the report…	"AAPL"	Multi-Doc RAG	Table	Based on the provided documents, th…
What is the impact of foreign exchange rates on Apple's financial performance? Li…	"AAPL"	Multi-Doc RAG	Table	- For the quarterly period ended June…
Are there any notable changes in Apple's liquidity position or cash flows as report…	"AAPL"	Multi-Doc RAG	Table	Based on the provided documents, th…
How does Apple's R&D expenditure in the most recent quarter compare to previo…	"AAPL"	Multi-Doc RAG	Table	In the most recent quarter ended July…
What legal proceedings or contingencies are disclosed in these 10-Qs and how m…	"AAPL"	Multi-Doc RAG	Text	The legal proceedings disclosed in th…
Has Apple engaged in any significant share repurchase activities in the reported q…	"AAPL"	Multi-Doc RAG	Table	Yes, Apple has engaged in significant…
What is the effective tax rate reported by Apple in these quarters and how does it …	"AAPL"	Multi-Doc RAG	Table	The effective tax rates reported by Ap…
Has Microsoft partaken in any substantial stock buyback programs in the reported…	"MSFT"	Multi-Doc RAG	Table	Yes, Microsoft has partaken in substa…

Figure 3.1: List of questions and answer present in the dataset

[2]The answer were generated by an other LLM and manually corrected by the researchers

3.3.2 Indexing

This script represent the core of the library, building the RAG pipeline by processing input documents and storing them in a vector storage [3]. The input documents are read from a *data source* (the S3 bucket) and processed through the chunking phase.
After the this phase the chunks are converted in embedding representation using the Amazon Titan Embedding model.

Chunker

For the initial stage, processing of the documents, I implemented two different versions of the Chunker (Recursive and Semantic), and the Bedrock Titan model was used to calculate the embedding representation of each chunk.

Recursive Chunker The Recursive Chunker leverages the Langchain Recursive Character Text Splitter to split each document in chunks [6]. This method employs a hierarchical splitting strategy, starting by dividing the text at the paragraph level and, if a paragraph is still too large, splitting it again at a sentence level. In this way each chunk is meaningful and coherent and contains semantically related piece of text.
Users can define the chunk size and chunk overlap parameters. The chunk size is important since Large Language Models (LLMs) have a maximum token capacity. Chunk overlap refers to the number of characters at the end of one chunk that are repeated at the beginning of the next chunk. This is done to preserve context among chunks helping the model to understand the continuity of the text.
For my experiments i chose a chunk size of 2000 and overlap of 200.

Semantic Chunker The Semantic Text Splitter in LangChain [7] is an experimental method to split text based on semantic similarity, addressing the limitations of others techniques that might disrupt the semantic flow by relying on predefined delimiters. This approach leverages language models to understand the text's meaning during the chunking process. For this strategy an embedding model (Amazon titan) is used to calculate embeddings for each sentence so that the most related ones are grouped in the same chunks.
It is possible to define the threshold after which two sentences will be assigned to different chunks. For my experiments i choose the default way to split text (based on percentile) so that all differences between sentences are calculated, and then any difference greater than 0.8 is split.

[3] The component of the knowledge base responsible for data storage, in particular using high dimensional vectors

Vector Storage

After processing the documents and generating embeddings for each chunk, various options were explored to store them to allow for a faster retrieval in subsequent steps.

FAISS The first vector storage option considered is FAISS (Facebook AI Similarity Search) [8], an open source library to search for vectors in an efficient way. It offers a method to create a FAISS index, stored locally, starting from the input chunks.
This tool is based on an in memory database to store vectors, for this reason does not scale. Moreover it offers approximate nearest neighbor search to improve speed, but impacting on the accuracy.

Opensearch Amazon OpenSearch Serverless is a managed service offered by AWS to operate and scale OpenSearch clusters (Open Source Search Engine) in the cloud AWS [9]. This is a search and analytics engine using Collections where the user can upload documents, or in this case embeddings. It offers also a specific functionality to work with high dimensional vectors and store them in an index.
This solution provides scalability, efficiency and security features. Also, as a managed AWS service, it simplifies the indexing process for developers, reducing manual steps and overhead.

Bedrock KnowledgeBase Amazon Bedrock provides an even simpler method for indexing documents through its managed service KnowledgeBase [10]. It is possible to define a DataSource (an S3 bucket for example) which is then synchronized with the underlying vector storage, performing automatically the processing of the documents to obtain the embeddings. By default KnowledgeBase is built on top of OpenSearch for storing vectors, simplifying indexing and the development process. Additionally, it supports automatic updates of documents. This solution make use of its default processing phase implementing a Recursive Chunker with configurable parameters.

Amazon Kendra Kendra is another AWS managed search service powered by machine learning [11]. It offers intelligent search functionalities by using NLP to understand the document's context and the meaning of the query. To index documents users need to create a DataSource and upload documents. After that, documents are automatically processed, generating embeddings. Such as the KnowledegBase options it doesn't require manual processing of documents simplifying the indexing process, and it relies on a search engine with performance comparable to OpenSearch.

3.3.3 Testing

After the knowledge base has been created, users can make queries to the system. The knowledge base is searched, and one or more documents (the ones which best answer the query) are retrieved. In the end, the text generation model can generate a response based on the information contained in the retrieved documents.

Retriever

Different vector storage solutions offer different methods for retrieving documents, and all of them have been analyzed to perform the comparison.
In this phase it is possible to specify the number of documents to retrieve for each query (top-K). For my experiment, i used the values 1,3,5,7. This approach allowed to compare the effectiveness and relevance of the search phase, analyzing the impact of the top-K on the quality of the answers.
Also, to compare the Amazon Bedrock and SageMaker services, two different text generation models were considered, one per each service.

Retrieve from Vector Storage

FAISS The FAISS library offers a simple method to retrieve relevant documents to a user query, searching in the pre-loaded FAISS index. This method receive the top-K parameter to specify the number of documents to retrieve and returns them with an associated score representing the L2 or Euclidean distance.

OpenSearch Using this AWS service it is possible to invoke the retrieval function, which by default consider the cosine similarity as the distance metric. It also accept the top-K as a parameter.

Bedrock KnowledgeBase Relying on the OpenSearch engine, it makes use of the same retrieval method of the previous option. Additionally, it provides the functionality to retrieve-and-generate, invoking the search function and automatically passing the returned documents to the text generation model, simplifying the development process.
The previous options instead, require the retrieved documents to be manually processed and included in the prompt for the generation model.

Kendra Amazon Kendra implement a retrieve method based on semantic similarity and leverages NLP to understand the query context to return the top-K most relevant documents. The relevance score is based on the cosine distance, as well as metadata.

Text Generation Model

In order to evaluate the differences between the Bedrock and Sagemaker services two different models were used to generate the final response. From the Bedrock foundation models, Anthropic Claude 2 was used [14], while Meta's LLaMA 2 13B Chat was selected from SageMaker JumpStart [13].
A crucial point in this phase is represented by the design of a prompt for the generation model including the documents retrieved in the retrieval phase.

Bedrock Model Invoking a Bedrock foundation model is very straightforward since they provide the basic API to invoke them. The method receives the prompt, augmented with the context retrieved, and generate a response.

SageMaker Model Amazon SageMaker offers the Jumpstart functionality to use pre-trained models and fine-tune or simply invoke them. There are many available models and each of these, according to the size and other parameters, has its own requirements in terms of CPU and memory.
Moreover, to invoke a SageMaker model it is first necessary to deploy an endpoint to host it. The endpoint correspond to a cloud environment were the model can be accessed using an API. After the model is deployed on the endpoint it can be called in the same way of the Bedrock one, by passing the prompt with the relevant documents included.

Model Parameters All the generation models can be customized by specifying the temperature, the topK (different from the one of the retriever) and the top-P. These are inference-type parameters changing the way models choose the next token.

- Temperature: it refers to the "randomness", modifying the probability of the next word in a sentence. An high value will increase the diversity in the chose of the words, leading to a more creative response. While a low value makes the model more repetitive, increasing coherence.
- TopK: this parameters refers to the maximum number of words from which the model will choose the next one. It limits the model choice to words with greater probability. A model with an high value has more variability, instead models with lower ones makes the output more predictable.
- Top-P: it apply a dynamic filter on the subset of possible next words, considering only the subset of words whose cumulative probability does not exceed the value p. High values increase diversity, lower ones decrease it.

It is crucial to choose a good combination of these parameters to be sure the model can generate coherent response imitating the human creativity.

For my experiments, to compare the two different models in the same situations, the same combination of parameters was used: Temperature=0, Top-P=0.1, topK=40.

3.3.4 Evaluation

Once the test have been conducted, the obtained result is a dataset where each question is associated to a human answer (the ground truth) and a model generated answer. Also, during the retrieval process, the relevant documents for each query were stored to evaluate also the quality of the knowledge base.
The evaluation of LLM generated text, as of today, is still a far from simple process. Some linguistic metrics were considered, traditionally used to evaluate some machine learning task. Also performance metrics, related to the time to retrieve documents and to generate an answer were collected. Lastly, the RAGAs framework (Retrieval-Augmented Generation Assessment) was used to calculate different metrics about the quality of the RAG pipeline.

Evaluation Metrics

All the following metrics were collected and analyzed to compare the different results.

Linguistic These traditional metrics give a score to the generated text, comparing it to the expected one. Their implementation is provided by the Hugging Face *Evaluate* library [15].

- BLEU (Bilingual Evaluation Understudy Score): particularly useful in machine translation tasks, this metric evaluates n-gram precision. It checks how many n-grams (contiguous sequences of n items from a given sample of text) in the generated text are present in the expected output.
- ROUGE (Recall-Oriented Understudy for Gisting Evaluation): this metric can be used to evaluate tasks where the recall is more important, such as summarization. It measures how much of the reference text is captured in the generated one.
- Meteor (Metric for Evaluation of Translation with Explicit Ordering): calculates the harmonic mean of precision and recall, with a higher weight given to recall. With respect to the other two, this metric considers synonyms, stemming and word order in the evaluation, making it more robust. But still it is not able to evaluate complex texts where a semantic understanding is required.

Specialized Metrics for RAG Systems The RAGAs framework [16] offers diverse metrics to evaluate all the aspects of a RAG pipeline. The core phases to evaluate are the retrieval (how relevant are the documents to the query and to the ground-truth) and the generation (how similar is the generated text with respect to the expected answer).
This first set of metrics is used to evaluate the quality of the context (documents retrieved) and the impact it has on the generated answer.

- Context Precision: evaluates how well the items (chunks) of the retrieved context match the ground-truth relevant items for a given query.
- Context Recall: evaluates whether each sentence in the ground-truth can be attributed to the retrieved context.
- Context Relevancy: evaluates whether sentences in the retrieved context are relevant for answering the question.

The following group instead gives insight about the the generated answer, evaluating how closely the generated text matches the expected one and whether it accurately reflects the information from the retrieved documents.

- Answer Semantic Similarity: evaluates the semantic similarity between the answer and the ground-truth by computing their embedding representation.
- Answer Correctness: is computed as a weighted average of factual correctness (using F1 score) and the semantic similarity between the given answer and the ground-truth.
- Answer Relevance: is defined as the mean cosine similarity of the original question to several artificial questions, which where generated (reverse engineered) based on the answer.
- Faithfulness: evaluates whether each sentence in the answer can be inferred from the context.

Most of the RAGAs metrics follow the "LLM as-a-judge" approach, where an other LLM is utilized to evaluate the results. This is particularly evident in metrics like the Faithfulness or Context Relevance, where the model evaluates the generated content against the retrieved context or ground truth and provides a score. Instead, metrics relying on answer similarity focus on the cosine distance between the embedding representations of the generated answer and the reference one to give a score.
For my experiments i used the Mistral 7B instruct model, from Bedrock foundation models, to perform the evaluations.

Other Metrics for RAG Evaluation Before RAGAs was chosen, a study on other different solution to evaluate a RAG system was conducted.
The Hugging Face *Evaluate* library offers basic metrics to evaluate machine learning models in diverse NLP tasks. Nevertheless, there aren't metrics specialized in evaluating the RAG retrieval and generation phase.
The DeepEval library [17] implements many of the metrics present in the RAGAs library, also introducing a new one called G-EVAL. This metric is particularly interesting because it is able to evaluate *ANY* custom criteria making use of the "LLM as-a-judge" approach. Despite its flexibility, G-EVAL is more difficult to set up and configure. Given the complexity and the time required to evaluate a single answer, I decided to use RAGAs for my experiments.

Performance Metrics This set of metrics is used to evaluate the performance of the retriever (to search and return relevant documents from the knowledge base) and of the generation model to answer in a fast way.

- Time to Retrieve: refers to the time needed by the vector storage to search the relevant documents based on the user query and return them. This is especially useful to compare the different vector storage solutions.
- Time to Answer: indicates the number of seconds the generation model needed to generate an answer after receiving the prompt with the query and the context. This can offer a comparison on the performance of the two types of models considered: the Bedrock and the SageMaker ones.

Chapter 4

Experimental Results

Thanks to the custom library it was possible to easily index documents using diverse chunking strategies and vector storage solutions. For each distinct configuration tests were conducted on the same dataset using the different models offered by the Bedrock and SageMaker service, while also considering different top-K values. In the end, the quality of the generated response and the general performances have been evaluated.
The experiments were conducted using the following parameters:

- Chunking strategies: Recursive Character Text Splitter and Semantic Text Splitter, both by LLangchain. Bedrock KnowledgeBase and Amazon Kendra also offer their default chunking strategies (referred as Custom for KnowledgeBase and Default for Kendra).

- Vector Storages: FAISS, Amazon OpenSearch, Bedrock KnowledgeBases and Amazon Kendra.

- Number of retrieved Documents (top-K): values 1, 3, 5, 7

- Text Generation Models: Claude2 from Bedrock Foundation Models and LAMA2 13b from SageMaker Jumpstart.

The total number of tests using all the possible combinations of these parameters would be 64, but due to the large dataset size (195 questions) and budget and time limitations, some configurations were not tested.

- For Amazon Kendra, tests were conducted using both its Default chunking strategy as well as the Recursive and Semantic Splitters. This approach was essential to evaluate the impact of the Chunker on the same vector storage.

- For the Bedrock KnowledgeBase vector storage solution, only its default document processing was performed. This is based on a Recursive Character Splitter with a chunk size and overlap of 1000 and 99, respectively. This limitation was introduced because a comprehensive comparison of processing strategies had already been carried out using the Kendra vector storage.

- Tests using the LLaMA model were conducted only considering the top-k value 3. This decision was made because all the tests conducted using the other model already provided good insights into the impact of the number of retrieved documents on response generation. The value 3 represent an average of the values used by Bedrock model and it gives the possibility for a direct comparison on the same configuration.
  Moreover, the Lama model comes with a lower limit in term of prompt size, and for this reason higher top-k values could not be used.
  Still it was possible to compare results of the different generation models on 40 different configurations.

Considering the introduced limitations, the total number of test becomes 40 (32 using the Claude model and 8 using LAMA model).
In table 4.1 is present a visual overview of the different configurations tested.

VECTOR STORAGE	CHUNKING STRATEGY	TEXT GENERATION MODEL	TOP-K
FAISS	RecursiveCharacterTextSplitter	CLAUDE 2	top1
			top3
			top5
			top7
		LLAMA2 13b	top3
	Semantic chunking	CLAUDE 2	top1
			top3
			top5
			top7
		LLAMA2 13b	top3
OPENSEARCH	RecursiveCharacterTextSplitter	CLAUDE 2	top1
			top3
			top5
			top7
		LLAMA2 13b	top3
	Semantic chunking	CLAUDE 2	top1
			top3
			top5
			top7
		LLAMA2 13b	top3
KENDRA	RecursiveCharacterTextSplitter	CLAUDE 2	top1
			top3
			top5
			top7
		LLAMA2 13b	top3
	Semantic chunking	CLAUDE 2	top1
			top3
			top5
			top7
		LLAMA2 13b	top3
	Default chunking	CLAUDE 2	top1
			top3
			top5
			top7
		LLAMA2 13b	top3
KNOWLEDGE BASE	Custom chunking	CLAUDE 2	top1
			top3
			top5
			top7
		LLAMA2 13b	top3

Table 4.1: Test Configurations

4.1 RAG Analysis

In order to perform the analysis of the results, each test produced a CSV file containing the utilized parameters and the values of the computed metrics. Once all the files have been uploaded on a S3 bucket, a Glue Crawler job [1] was launched to derive an SQL table from these.
Amazon Glue Crawler enables to populate a table starting from different file format, specifying the corresponding delimiter (comma separated value in this case). This service will scan all the files, deriving the table schema from the file content and folder organization, and at the end populating it with the contained values.
The `Metrics` table schema is structured as follows:

- **vector_storage**: Refers to the vector storge solution (FAISS, OpenSearch, KnowledgeBase, Kendra).
- **chunking_strategy**: Describes the approach used to break down the data into manageable chunks for processing (Recursive, Semantic, Default for Kendra and Custom for KnowledgeBase).
- **topk**: Refers to the number of retrieved documents (1, 3, 5, 7).
- **meteor**: Measures the precision and recall of predicted text against reference text.
- **sacrebleu**: The BLEU metric to evaluate the precision.
- **rouge-l**: Variant of ROUGE that evaluates the overlap of longest common sub sequences between predicted and reference text.
- **context_precision**: calculate the precision of the context for which the answer is retrieved.
- **context_recall**: Measures the recall of the context used to generate the answer.
- **context_relevancy**: Determines the relevance of the context with respect to the query.
- **answer_similarity**: Evaluates the semantic similarity between the generated answer and the correct answer.
- **answer_correctness**: Assesses whether the generated answer is factually correct.
- **answer_relevancy**: Measures the relevance of the generated answer to the posed query.

[1] AWS Glue crawler is an automated tool to discover and catalog data.

- **faithfulness**: Evaluates how accurately the generated text reflects the information in the retrieved context.
- **time_to_retrieve_documents**: The time taken to retrieve the necessary documents for generating an answer.
- **time_to_answer**: The time taken to generate the answer.

4.1.1 Vector Storage Comparison

Analyzing the results of the Bedrock model, it is possible to compare the different vector storage solutions by calculating averages of the computed metrics.
In figure 4.1 it is possible to visualize the different scores obtained for the quality metrics.

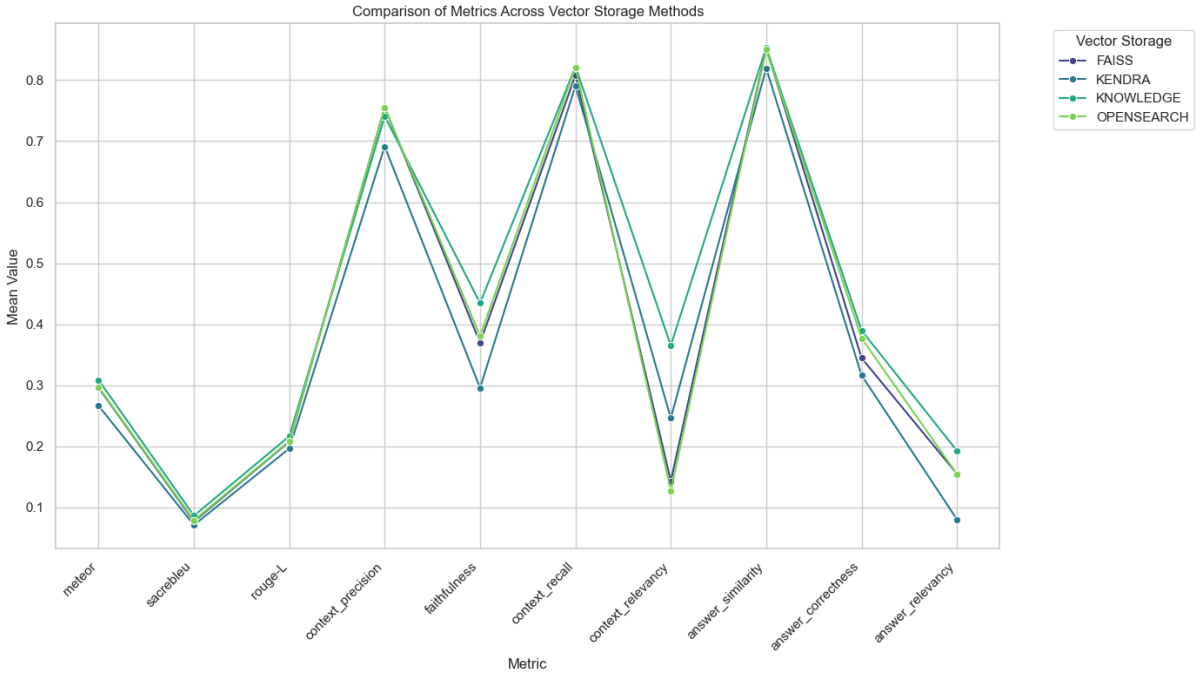


Figure 4.1: Comparison of the quality metrics for the different vector storage solutions

Vector_storage	Meteor	Rouge	sacrebleu	Context_Precision	Context_Recall	Context_Relevancy	Faithfulness	Answer_similarity	Answer_correctness	Answer_relevancy
FAISS	0.28	0.20	0.076	0.81	0.83	0.14	0.37	0.85	0.38	0.17
KNOWLEDGE	0.28	0.21	0.070	0.81	0.84	0.38	0.42	0.85	0.40	0.20
OPENSEARCH	0.28	0.20	0.086	0.81	0.83	0.14	0.37	0.85	0.39	0.16
KENDRA	0.25	0.19	0.078	0.76	0.81	0.26	0.32	0.81	0.36	0.08

Table 4.2: Tabular representation of the vector storage results

Overall, linguistic metrics average score(meteor, sacrebleu and rouge) is below 0.3. This reveals their limitations in evaluating LLM text generation, since they are useful in tasks like summarization and machine translation and are not effective to determine if two sentences express the same concept.
RAGAs metrics regarding context retrieval and recall obtained relatively high values, indicating the retrieval process of the RAG pipeline is effective. All the vector storage solutions performed similarly, with the Amazon Kendra implementation scoring the lowest.
Nevertheless, with the exception of answer similarity, the other results are below 0.4. This indicates that, in general, the generated answers do not align to the expected ones.
It is also important to note that these metrics heavily rely on the scores assigned

by the evaluator model (Mistral), therefore the reason for the low scores may be attributed to the model's limitations in understanding.
With this in mind, Amazon Kendra got the lowest scores for these metrics while the KnowledgeBase solution achieved the highest ones. Anyway, as it is clear from the figure, OpenSearch and FAISS performed similarly to KnowledgeBase.
Analyzing the metrics more in detail:

- The answer similarity metric scored high values, indicating the generated and expected answers have similar vector representations. This is a particularly important metric to determine the quality of the generated answer, as it's score is independent of others LLM evaluation.
- Context precision and recall also have high scores, demonstrating the retrieved documents contains the information to generate the response.
- Context and answer relevancy are the RAGAs metrics with the lowest score. These two heavily rely on the computation performed by the evaluator model and they will probably increase by using a more powerful model.

Time Performances From the time performances point of view, the average retrieval time is relatively high for the Kendra and KnowledgeBase solution, though staying under one second. FAISS seams to be the fastest solution to retrieve documents, closely followed by OpenSearch.
The time to generate answers across the different solutions is similar and remains under 10 seconds 4.3. This was expected since it is primarily influenced by size of prompt sent to the model, depending on the top-k value.

vector_storage	avg_time_to_retrieve	avg_time_to_answer
KENDRA	0.712	7.787
FAISS	0.094	8.524
KNOWLEDGE	0.483	7.678
OPENSEARCH	0.148	8.291

Table 4.3: Average times for vector storage solutions

4.1.2 Chunking Strategy Comparison

Analyzing the results obtained from the Bedrock model it is possible to point out the effect of the chosen chunking strategy on the generated response quality.
In figure 4.2 it is possible to see the differences in the quality metrics results.
The results are consistent with the previous ones where only context precision, context recall and answer similarly achieved high scores.
The Chunking strategy is affecting only the processing phase. When chunks are created in a more intelligent way they preserve better context understanding, facilitating the retrieval of highly relevant documents to the user query. For this reason, i initially expected the Semantic strategy to obtain better results. However,in the end the scores were nearly identical.
The Default chunking strategy used by Amazon Kendra vector storage got the lowest results, while the Custom one, implemented by Bedrock KnowledgeBase, has the highest ones.

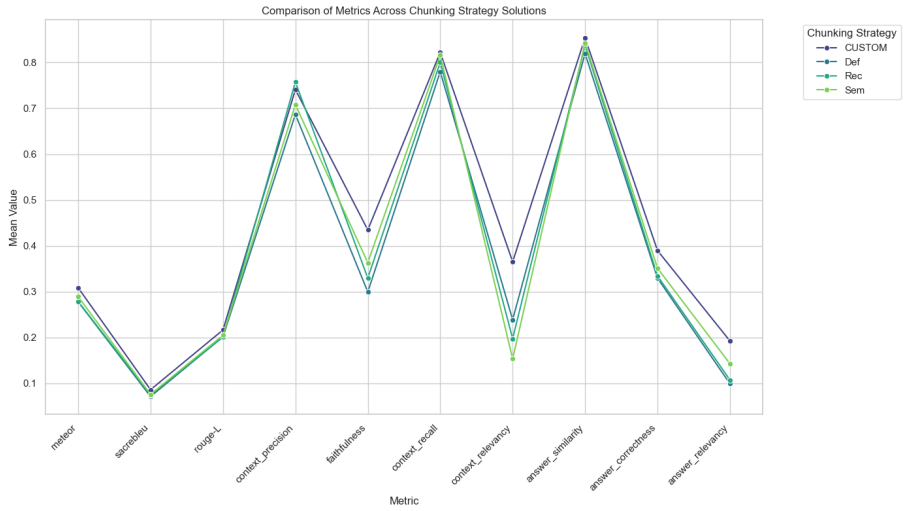


Figure 4.2: Comparison of the quality metrics for the different chunking strategy solutions

Chunking_Strategy	Meteor	Rouge	sacrebleu	Context_Precision	Context_Recall	Context_Relevancy	Faithfulness	Answer_similarity	Answer_correctness	Answer_relevancy
Custom	0.2811	0.2069	0.086	0.8061	0.8438	0.3768	0.4190	0.8502	0.4015	0.1995
Def	0.2591	0.1937	0.071	0.7603	0.8094	0.2422	0.3064	0.8101	0.3573	0.0913
Rec	0.2669	0.1978	0.074	0.8016	0.8121	0.2107	0.3410	0.8375	0.3674	0.1137
Sem	0.2744	0.2006	0.075	0.7923	0.8358	0.1593	0.3601	0.8406	0.3861	0.1565

Table 4.4: Quality metrics for different chunking strategies

We can focus on a single vector storage solution, Amazon Kendra, to analyze the effect of the different chunking strategies, and in particular of the Semantic strategy. In figure 4.3 only RAGAs metrics results are present for the Kendra vector storage solution.
Even in this case the scores between the different strategies remain the same, without any noticeable improvement from using the Semantic strategy. Therefore, at least at the time of this project, i do not consider particularly relevant this

experimental feature.

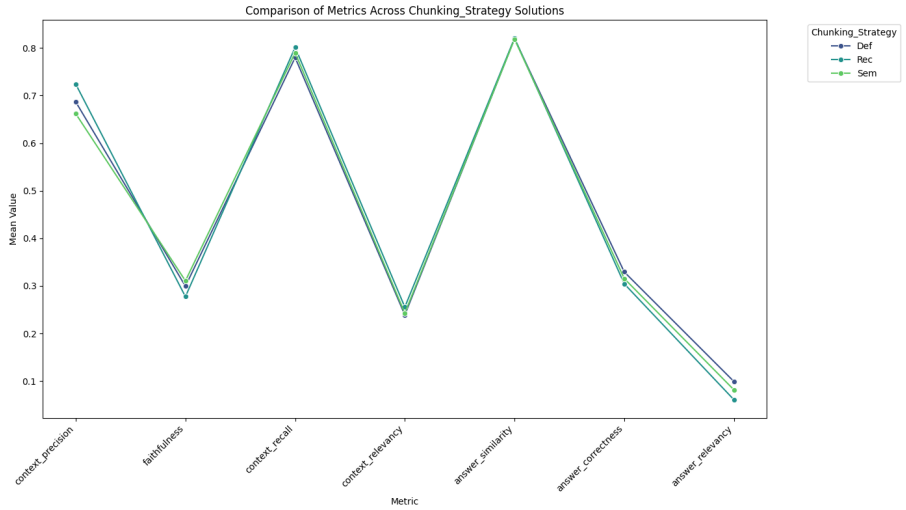


Figure 4.3: Comparison of the quality metrics using different chunking strategies and the Kendra vector storage

Time Performances Regarding the time performances, the considerations made during the previous vector storage comparison remain valid, where the Default and Custom strategy (corresponding to the Kendra and KnowledgeBase solutions) take the more time to retrieve documents. The Recursive strategy appears as the best solution to retrieve documents and generate the final text, from the time point of view.

Table 4.5: Average times for chunking strategy solutions

chunking_strategy	avg_time_to_retrieve	avg_time_to_answer
Def	0.7190	8.050
Custom	0.4835	7.679
Rec	0.3062	7.677
Sem	0.3279	8.638

4.1.3 Top-K Comparison

Comparing the results obtained retrieving different number documents (top-k parameter), enables to show the differences in term of quality of the context and time to retrieve.
High values of this parameter will more likely give the model more understanding of the context to answer the question. However, it will also increase the possibility that the model answer is not so precise. Also the retrieval and generation times will be negatively affected since the model need more time to analyze the context to produce a response.
In figure 4.4 and table 4.6 we can analyze the scores of the quality metrics.
As we already said, the metrics which scored the highest are context precision,

Chunking_Strategy	Meteor	Rouge	Sacrebleu	Context_Precision	Context_Recall	Context_Relevancy	Faithfulness	Answer_similarity	Answer_correctness	Answer_relevancy
k1	0.2582	0.1952	0.072	0.8268	0.8188	0.3879	0.3700	0.8307	0.3557	0.1166
k3	0.2697	0.1978	0.073	0.7943	0.8068	0.2186	0.3702	0.8326	0.3719	0.1272
k5	0.2756	0.2009	0.079	0.7816	0.8315	0.1487	0.3459	0.8391	0.3945	0.1477
k7	0.2785	0.2041	0.078	0.7716	0.8415	0.1097	0.3172	0.8448	0.3870	0.1593

Table 4.6: Quality metrics for different top-k values

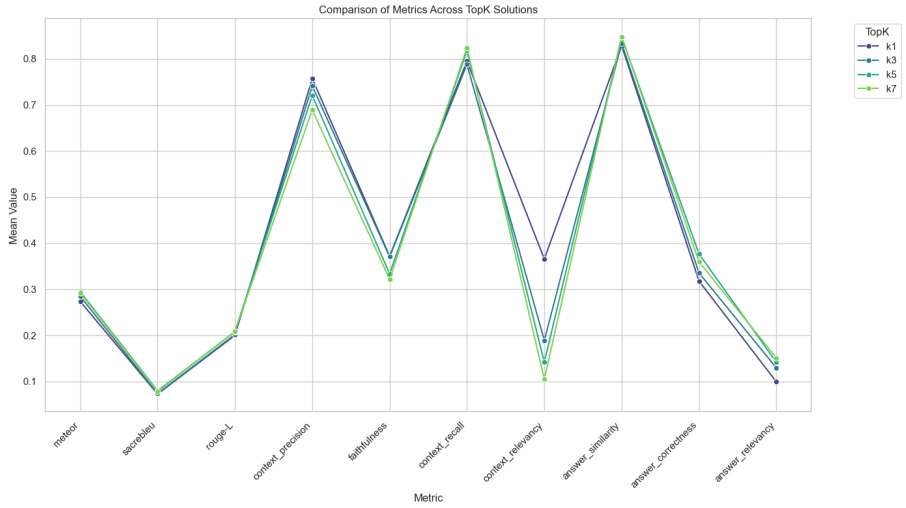


Figure 4.4: Comparison of the quality metrics for the different top-k values

context recall and answer similarity.
The context precision is evaluating how closely the context aligns with the expected answer. This is done by counting the number of *items* present in the context matching those in the ground truth. Given a larger context, higher top-k value, the amount of items in the context not present in the answer increase, scoring lower for that metric. As a matter of fact, the figure is showing this behaviour, with the highest score for top-k 1 and lowest for top-k 7.
The context recall, on the other hand, count how many items in the expected answer are present in the context retrieved. For this reason a higher number of retrieved documents generally increase this score.
Analyzing the context relevancy results, despite the low scores, it is evident that using a lower top-k increases the number of statement in the context relevant to

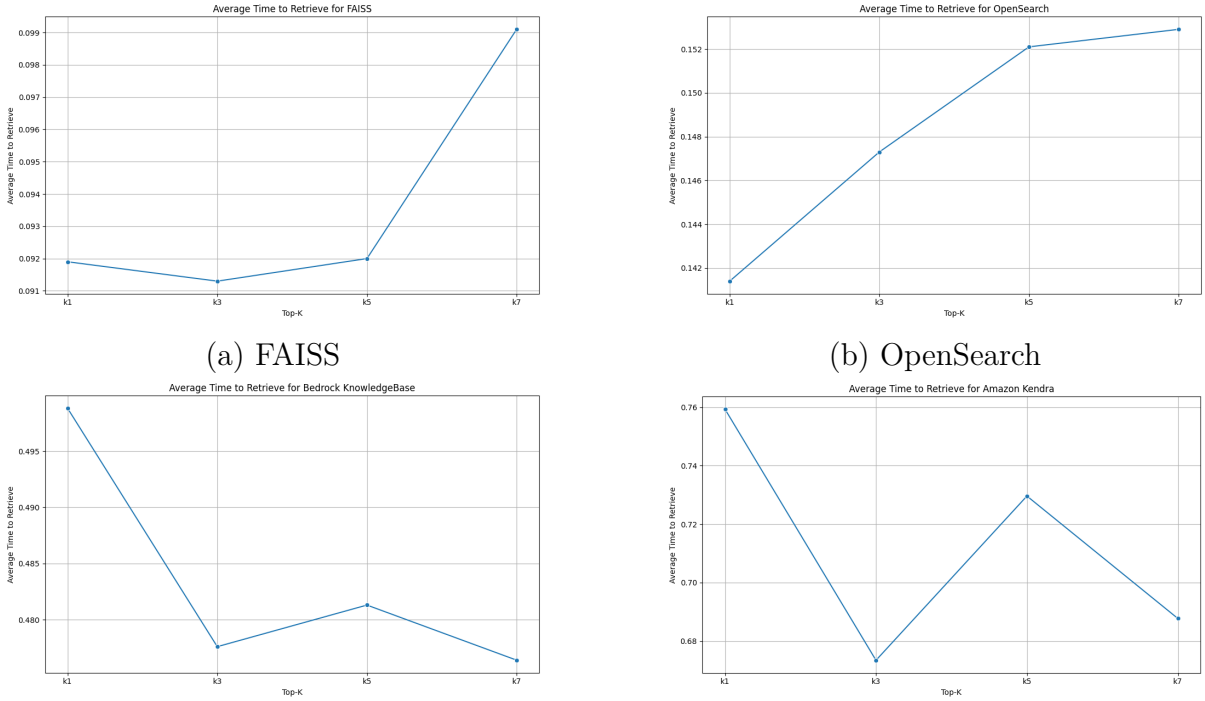
the answer.
These three metrics highlights the fact that even if retrieving more documents increases the chances of correctly answering the question, if the retriever is efficient and it is able to return the most relevant documents to a user query, just 1 or 3 documents provide all the necessary information.
Despite this considerations, the answer similarity metric still scores high for all the values of the top-k. This is showing that the generated answer is matching closely the expected one regardless of the size of the context used. It also stress the fact that a single well-chosen document can provide all the necessary information to produce a correct answer, as long as it is the most relevant.

Time Performances Analyzing the average time results 4.7, as expected, the time to answer increases with the number of retrieved documents. However, the time to retrieve decreases as more documents are searched. This is counterintuitive since one would expect that searching more documents in the knowledge base would take longer.

chunking_strategy	avg_time_to_retrieve	avg_time_to_answer
k1	0.4054	6.779
k3	0.3719	7.766
k5	0.3947	8.635
k7	0.3804	9.155

Table 4.7: Average times for top-k solutions

To better understand this strange results, it is possible to focus on the different vector storage and compare the effects of the top-k values as in figure 4.5.
The results show a strange pattern for the Amazon KnowledgeBase and Kendra vector storage, where the time to retrieve using top-k 1 and 5 is higher than the time for top-k 3 and 7. A possible explanation could be that these Amazon services are optimized for batch operations, requiring less time to retrieve a higher number of documents.



(a) FAISS

(b) OpenSearch

(c) KnowledgeBase

(d) Kendra

Figure 4.5: Average Time to Retrieve for Different Vector Storage Methods Across Top-K Values

4.1.4 General Comparisons

Some general considerations can be done analyzing the results of all the test for the Bedrock generation model. In particular, the objective is to find out the best solution to realize a RAG pipeline.
The first analysis we can perform regards the average time to retrieve documents. From the previous paragraphs it is already known that the FAISS and OpenSearch vector storage have the best scores.
The figure 4.6 is confronting all the solutions revealing that FAISS with top-k 1 and the Semantic chunking strategy is the one taking less time with an average of 0.088s. The OpenSearch solution with top-k 1 and Recursive strategy instead scored 0.146s. The worst results comes from the Kendra vector storage.

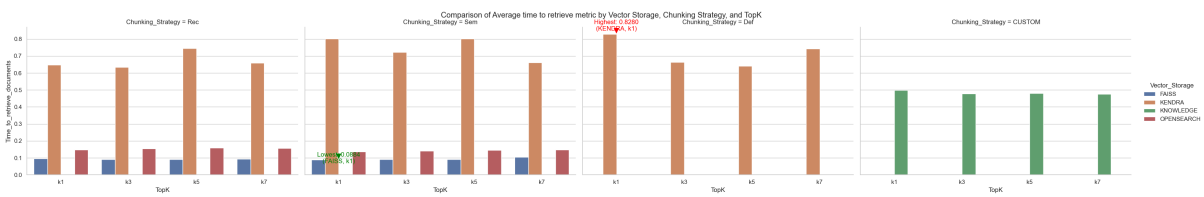


Figure 4.6: Comparison of the time to retrieve metric

To analyze the best solution from the quality point of view i aggregated the metrics

evaluating the context and the ones evaluating the answer in two different metrics representing the averages of all the values.
In figure 4.7 it is possible to see that the solution using Bedrock KnowledgeBase with top-k 1 is the one with the highest context quality score, followed by Kendra with top-k 1. These managed services seams to offer the best solution to store and retrieve documents of the context for a given query, containing all the necessary information to generate the answer.
In figure 4.8 the results of the answer quality metrics are shown.
This group of metrics evaluate how closely the generated response aligns to the expected one and to the context producing it. We can see that the FAISS solution with top-k 7 and the Semantic strategy is the one with the highest score (0.491). The KnowledgeBase solution with top-k 5 reached a value of 0.485, while Kendra based configurations are the ones with the lowest scores around 0.39.

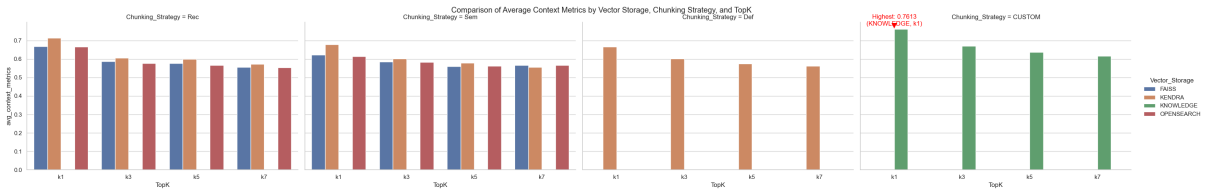


Figure 4.7: Comparison of the context quality metrics

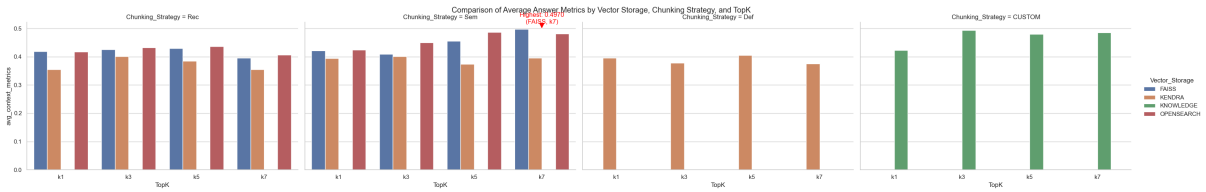


Figure 4.8: Comparison of the answer quality metrics

4.1.5 Bedrock VS SageMaker

To determine the best service for the development of the RAG system many factors need to be accounted.
As already described in chapter 3.2, costs, specialization and final goal need to considered.
In this section a more detailed comparison on the two services is accomplished. The implemented RAG system is designed to work with different text generation models, offered both by Bedrock and SageMaker. Therefore, by analyzing the results obtained using two distinct generation models belonging to different services, it is possible to draw relevant conclusions.
Since the SagkeMaker model was evaluated only for topk 3, the comparison is done only with the configurations with the same topk for the Bedrock model.

Perfomance Comparison The text generation model has only an impact after the retrieval phase, thus the different models do not influence the average time to retrieve documents.
Analyzing the diagram 4.9 it is possible to understand the differences related to the time in generating an answer.

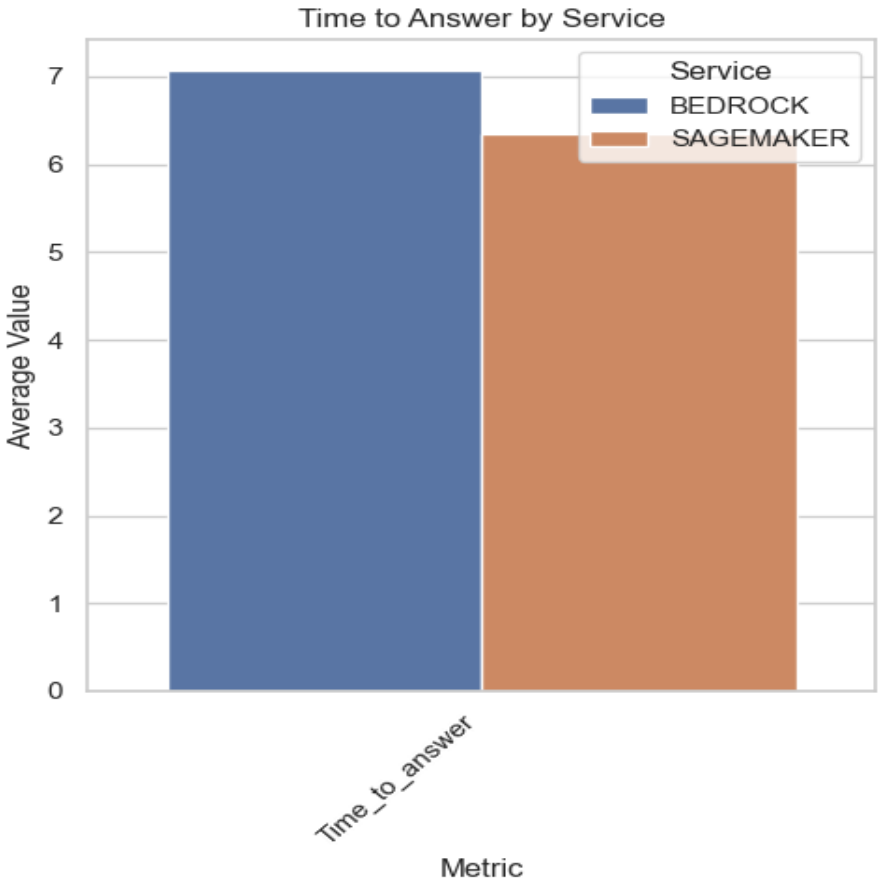


Figure 4.9: Comparison of time to generate answer for different services

The results show that the Lama2 SageMaker model is generating answers faster than the Claude2 Bedrock model. Taking approximately one second less per question.
This indicates the Lama2 model generate text before the Claude2.

Quality Comparison Now we want to focus on the effect of the choice of the model on the quality of the answer generated, in terms of accuracy and relevancy to the context.
It is possible to compare the different results obtained by the two models on the totality of the experiments.
Figure 4.10 shows the computed metrics.

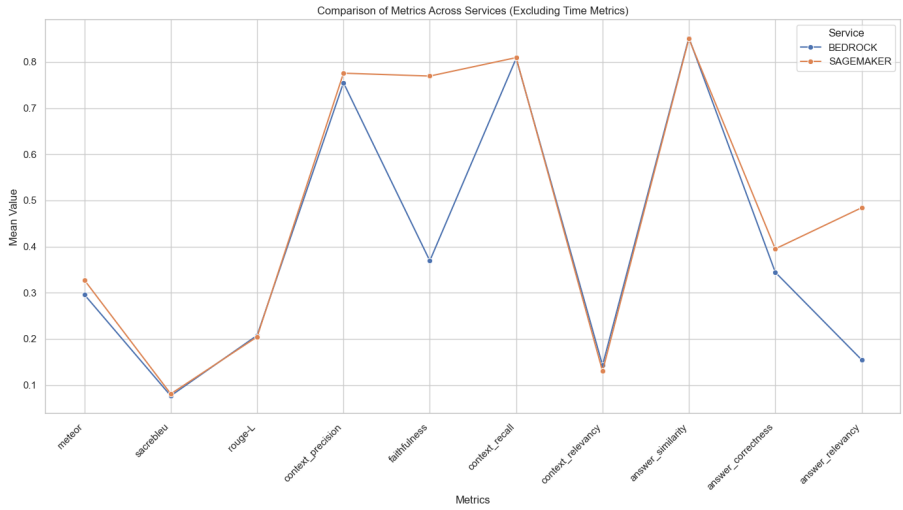
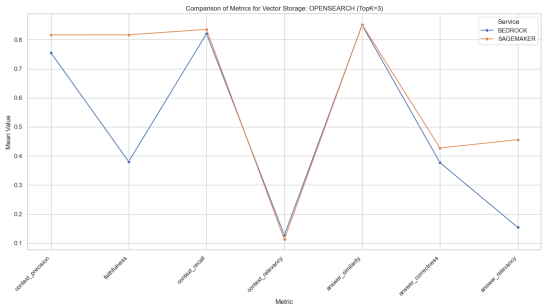


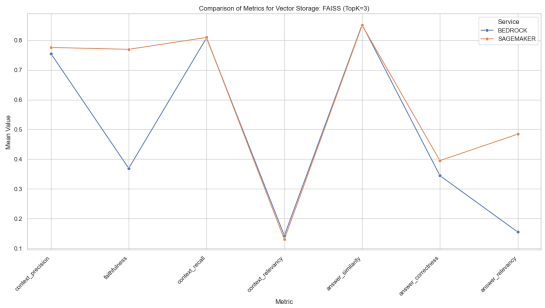
Figure 4.10: Comparison of quality metrics for different services

Most of the metrics score the same results for the different services. Anyway the score of the faithfulness is notably better for the SageMaker model. This indicates that the answer generated aligns more closely to the context. Also answer relevancy is higher for this model, indicating the answer is more relevant to the query.
The results show relatively better quality achieved by the Lama2 model from SageMaker.
In order to further analyze the data we can compare the results obtained by the different vector storage for the two services 4.11.

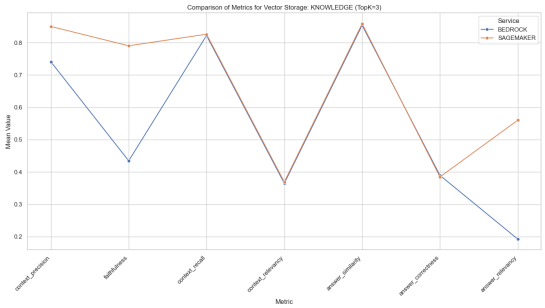
The plots follow the general trend where metrics score are higher for the SageMaker model, in particular faithfulness and answer relevancy. This result underline the fact that the Lama2 model generated more accurate response independently from the vector storage used, and indeed from the retrieval phase of the documents. Since the context generated is the same, only the model ability in generating an answer are evaluated.



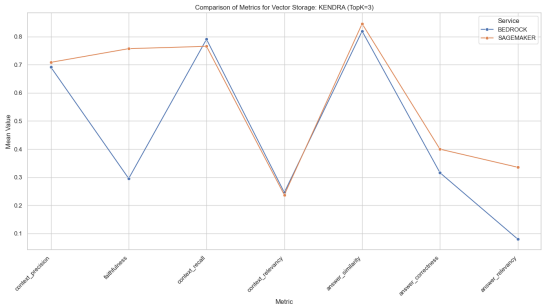
(a) OpenSearch Vector storage



(b) FAISS vector storage



(c) KnowledgeBase vector storage



(d) Kendra vector storage

Figure 4.11: Comparison of quality metrics for the services for the different vector storage implementations

4.2 Conclusions

The custom library succeeded in the objective of simplifying a RAG system creation and testing, allowing to easily customize the several components adjusting the parameters.
The results of the various tests confirmed some expected behaviours but also highlight some discrepancy. By analyzing the obtained results the following considerations can be drawn:

- FAISS vector storage is the fastest vector storage to retrieve documents, but KnowledgeBase with an appropriate top-k value scored higher for context quality metrics. Regarding these two solution they got very similar scores for the answer quality metrics.

- The Semantic chunking strategy seams particularly effective in enhancing the FAISS results for the answer quality metrics.

- The top-k parameter influences the different vector storage in different ways. However, higher values increase the time to generate the answer without always increasing the quality of the answer or the context.

- The considered SageMaker model overall performs better than the Bedrock one. It takes less time to produce an answer and the quality of it seams higher.

In conclusion, the best solution for a RAG system appears to be the FAISS vector storage combined with a small top-k value and the Semantic chunking strategy. Despite being one of the fastest and more accurate configuration, it falls short in scalability, making it less suitable for enterprise projects involving a large number of documents for the knowledge base. Also, it requires the developer to manually index documents and track updates, increasing the development time.
For this reasons, Amazon Bedrock KnowledgeBase may be a better alternative. It highly performs in terms of context and answer quality, and it automatically manages documents updates. Moreover it provides a *retrieve and generate* function, which accelerate the time to production and eliminates the need for developers to manage document processing and infrastructural concerns, simplifying the development.
Considering the small differences in terms of performances and quality of the Bedrock and SageMaker model, the first may be a better alternative keeping in mind the costs.

Chapter 5

Case Study: GDO

The project focuses on developing a shopping assistant for a company in the large-scale retail industry (GDO). The ultimate goal is to enhance customer interaction during the online shopping phase by providing advanced features. Customers will be able to create their shopping carts in a more intuitive and innovative way by asking natural language questions to the chatbot.
Key capabilities of this chatbot includes:

- Natural Language Understanding: The chatbot is capable of understanding and processing customer questions posed in natural language. Whether a customer asks about product availability, requests specific items, or inquires about recipes, the chatbot can comprehend the intent behind the query and respond accordingly.
- Database Interaction: The chatbot has access to the company database to check in (almost) real-time products availability and all the information about them.
- Cart Management: Customers can interact with their shopping cart directly through the chatbot. This is programmed to add or remove products based on user commands, allowing customers to efficiently build their carts without the need to scroll through hundreds of products on the company website. This streamlined process makes shopping faster and more convenient.
- Recipe Creation: The shopping assistant can suggest products based on various characteristics, such as "high-protein," "gluten-free," or "vegan" items. Moreover, it has a knowledge of the basic recipe ingredients and can combine different user filters to recommend the best products tailored to individual dietary needs.
- Information Retrieval: Beyond cart management and product availability, the chatbot can provide detailed information on a variety of topics. This includes general information about the company, answers to frequently asked questions (FAQ), and specific details about the store where the shopping is taking place,

such as opening hours or online shopping hours. This ensures that customers not only have a convenient shopping experience but also have easy access to all the information they need to make informed decisions.

5.1 RAG pipeline

The chatbot is built using Amazon Bedrock services, utilizing the Claude3 foundation model for text generation. The KnowledgeBase service is responsible for developing the RAG pipeline, while Bedrock Agents enable the model to call APIs.

Implementation Based on the precedent study of RAG technology with different solution, we chose Bedrock KnowledgeBase vector storage, with it's default chunking strategy and top-k values. This approach ensures efficient generation time and increments the accuracy and contextual understanding of the generated response.
The source documents are stored in an S3 bucket and cover frequently asked questions (FAQs) on various topics, including registration, loyalty programs, and how to order online. Other documents provide legal information about the company and user privacy regulations.

5.2 Agents

The project solution is based on this advanced technology offered by Amazon Bedrock, allowing a model to invoke previously defined APIs.

- The first step consists in the implementations of the APIs the model will call. These APIs are represented by Lambda functions, with their Docker images built and stored in the Elastic Container Registry (ECR) provided by Amazon.
- The model needs a way to understand the purpose of each API, what inputs they require, and what outputs they return. This allows it to select the best API based on the user's query. To achieve this, the OpenAPI schema is used. This consists in a JSON file describing the API, defining its parameters and expected response. All the schema are saved in an appropriate S3 bucket.
- The main advantage in using the Bedrock Agent service is represented by the built-in orchestration prompt and Agent implementation. When a user submits a query to the model, it determines the best actions to generate a response. Notably, the model is queried recursively, storing the results of each step and taking additional actions until the query is fully answered.
  In practice, this allows the model to call one or more APIs (represented by action groups) if necessary or search the knowledge base to retrieve all the necessary information. Then, it can analyze the context and generate a comprehensive response.
- An important part in the development of such agent is the design of a suitable system prompt. This is used to set up the model answer tone and define the types of questions it can address, as well as those it should avoid. The prompt also includes examples of potential action flows the model should follow to respond to specific queries.

5.2.1 Action Groups

Three action groups where defined to address all the capacities of the shopping assistant:

1. Search product: Check product availability in company's inventory and retrieve information about it.
2. Cart interaction: Display the content of the cart or, given a list of products, add or remove them from the cart.
3. Get Store General Info: For a specific store, return general information (address, contacts, website), store's opening hours or the operating hours of the online shopping service.

Search Products Interacting with the vast client database of products represented a challenge, as querying the entire database for every product search was not feasible. The main issue was that users often searched for a product by name, and the only way to find it in the database was to use a *like* function applied to different fields. However, after a few tests, we found out the results where not satisfactory. For example, a search for milk will return lots of unrelated products, such as milk chocolate or milk buns.
Returning so many results from the database negatively impacted performance and necessitated additional filtering to exclude irrelevant results. Moreover, the Bedrock Agent has limitations on the size of the response returned by the API.
In the end, if a customer asks for milk we cannot answer with detailed information about milk buns instead.
For all these reasons a dump of the products data was saved on an Amazon OpenSearch Collection. This approach enables the creation of an index supporting efficient searching of products by name.
The index is configured to handle various fields with different weights and uses a tokenizer for partial matching. It also leverages an edge-ngram-analyzer to allow partial matching of product names and a lowercase-normalizer for consistent case-insensitive searches. The index includes various field types, such as text, keyword, and float, to handle different product attributes effectively.
The applied search function uses the *multi_match* function to consider matches in different fields. Also the fields weights are different to give relevance to attributes such as the *Legal denomination of sale* more than the *Description*.
With this solution the results were definitely more accurate. Still, given that the product database contains many variants of the same product (milk in different formats, brands, and characteristics) a search for a simple product name would still return an excessive number of results. To address this problem, a score is assigned to each matching product, according to the weight functions, and the top ones are returned to the Agent.
The search function also gives more weights to product matching a specific brand. In this way it gives the possibility to return more frequently products belonging to a given partnership.

Cart Interaction The agent is programmed to track the session ID of the conversation with the user. This session ID uniquely identifies a single user and serves as a key in a database table to store all information related to the cart.
An Amazon DynamoDB table is utilized to store cart content as a dictionary field, where the key is the session ID. This is a non-relational database, document-based like MongoDB.
The action group is designed to manage three distinct cases:

1. Add to Cart: This API adds a new entry to the database table, containing the details of the product(s) to be added.

2. Remove from Cart: This API allows for the deletion of one or more products from the cart.

3. Check Content: The API queries the table and returns the list of products currently in the cart.

Each product is identified by its ID, and details include name, description, image, etc.

Get Store General Info This action group invoke APIs that query the original company database table to retrieve the necessary information. Using the session ID of the request, the specific store can be identified. Based on this information, a query is performed on the company Vertica [1] database to obtain general details about the store, such as its name, address, and contact information.
Additionally, the store's opening and service hours can be retrieved. Each store has different service hours, indicating the times during which online shopping and home delivery services are available.
Overall, these three APIs enable the retrieval of all the essential information about a store.

[1] A columnar database designed for analytics, ideal for real time data analysis

5.3 Project Infrastructure

The project do not only deal with Generative AI but also involves creating a suitable back-end and front-end infrastructure to ensure a scalable, secure, and reliable solution. AWS services support the creation of such a server-less application. All the infrastructure is developed using the AWS CDK (Cloud Development Kit), a software development framework for defining cloud infrastructure in code and provisioning it through AWS CloudFormation.

Back-end Stack This stack includes several key components: API Gateway, DynamoDB tables, Cognito User Pools, and Lambda functions. In figure 5.1 there is an high level description of the project infrastructure.

- API Gateway: A REST API is created to handle the different incoming request and forward them to the correct service. Based on the request path the request is sent to the corresponding Lambda function.

- Cognito Authorizer: Attached to the API Gateway to ensure all requests are authenticated. This component is used in the context of the OAuth2 scope to perform user login and authorize requests. It consists of a User Pool containing registered users and enforcing login with a username and password.
  Following the OAuth2 Authorization flow, it returns an access token upon successful login and adds a specific header to the requests to indicate they are authorize, also identifying the user.

- DynamoDB tables: The Cart table is created at this point having the session-id as the key.
  A Session-Parameters table is also deployed to store session-specific parameters such as the identifier of the specific store.

- Lambda function: This function is configured to handle API requests and perform actions based on the request path.
  The Lambda includes different endpoints to interact with the cart (updateQuantity, emptyCart, showContent) but the core endpoint is the invokeAgent. When a request with this path is received, it integrates with the Bedrock service to send the user query to the Agent and format responses.

Networking Stack This Stack aims to create a networking infrastructure in AWS, which is crucial for the back-end components to function securely and efficiently.

- VPC (Virtual Private Cloud): The VPC represent an isolated network where the different components can communicate securely. It is deployed across two availability zones to provide high availability and fault tolerance.

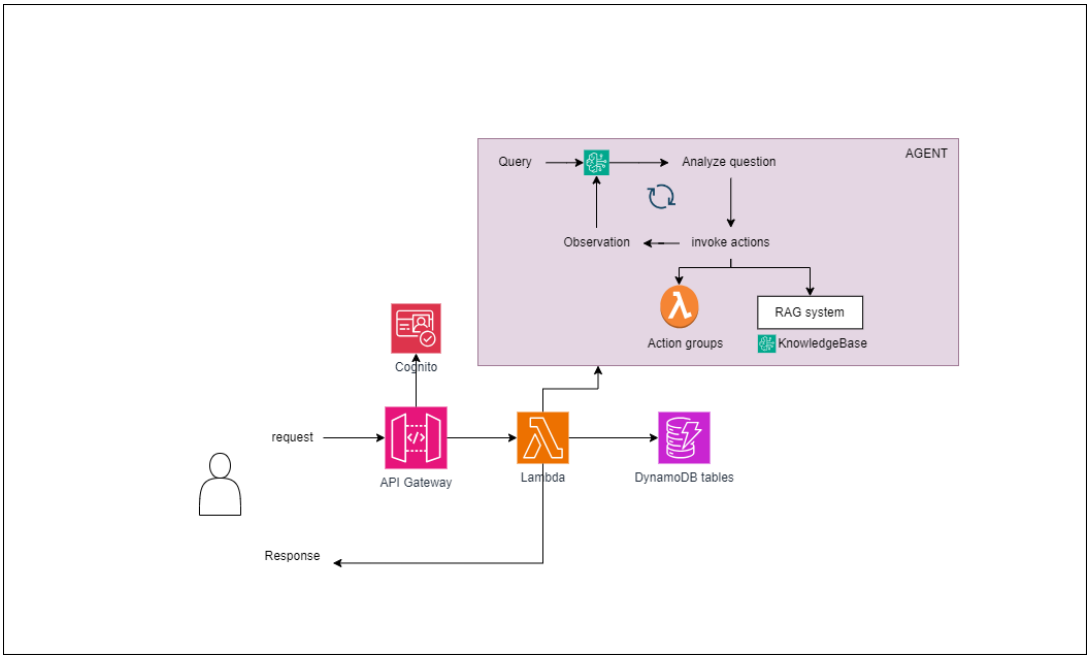


Figure 5.1: Back-end infrastructure

- Subnets: A public subnet contains resources that needs to be accessible from the Internet, such as the API GW, to handle incoming requests from external users. The private subnet, instead, host resources that should remain isolated by the public internet. For instance the Lambda function can only be reached by other resources within the VPC, enhancing security.

Front-end Stack The stack includes several components to ensure a scalable, robust and secure deployment leveraging AWS services. The application utilizes the same VPC created in the networking layer.

- Elastic Container Service(ecs): Cluster set up in the VPC to orchestrate the deployment and scaling of the containerized applications. It ensure the application is always running and provide the auto-scaling functionality to adjust the number of EC2 instances (underlying infrastructure for the running containers).
- DNS: Amazon Route 53 is used as a DNS, managing the domain name. An SSL/TLS certificate ensure security in the connection to the application.
- Fargate Service: automatically deploys a Load Balancer to distribute traffic between the container running the application.

The infrastructure is built using CDK, while the front-end pages employees Next.js framework.

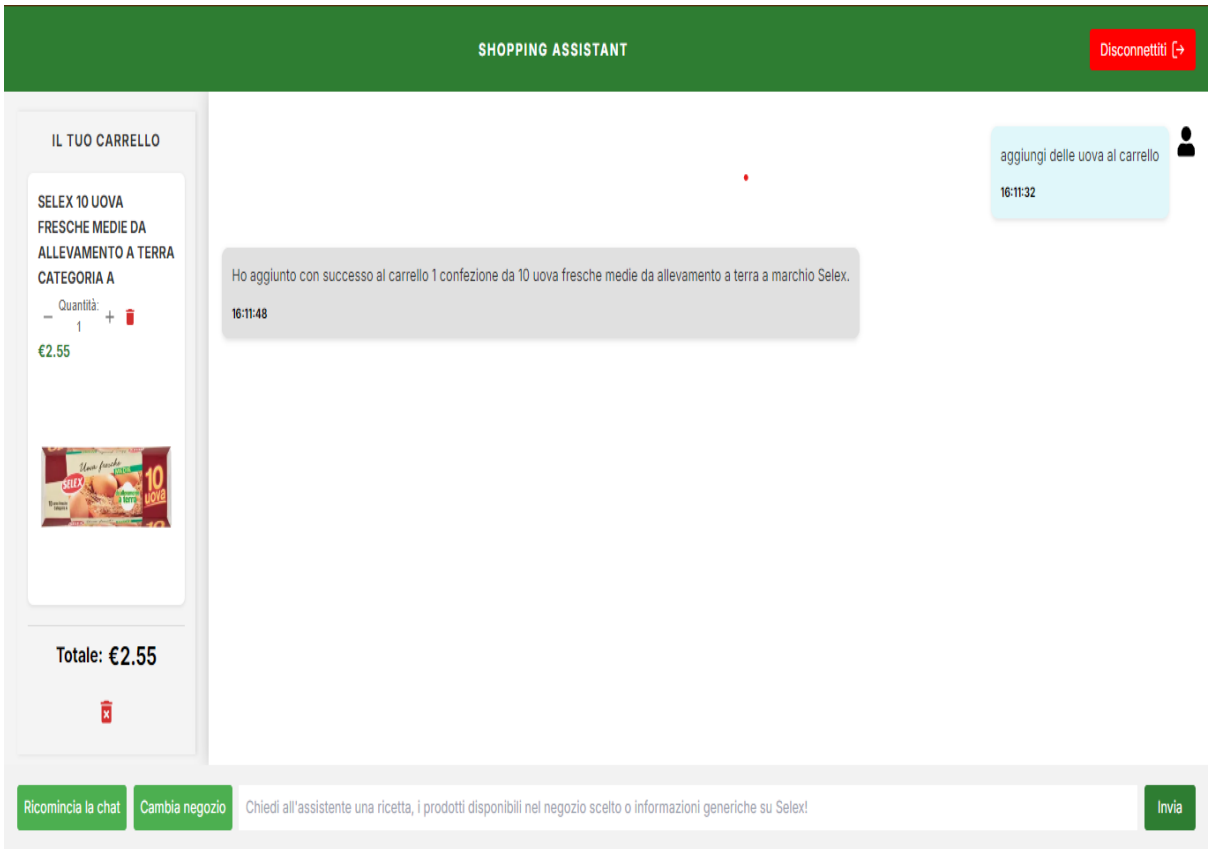


Figure 5.2: Web application appearance

Chapter 6

Conclusion And Future Work

The work proposed a flexible configurable library designed to simplify and accelerate the implementation of a RAG pipeline. It deploys the necessary services and offers the possibility to test the pipeline on a given dataset. Additionally, it provides a functionality to evaluate the results for statistics analysis and make informed decisions based on the quality of the generated responses.
The experiments did not definitely determine a single configuration superior to all the others, since the results were very similar. Nevertheless, the resulting data highlight that the solution combining the FAISS vector storage, the experimental Semantic Chunking strategy, a top-k value of 3 and the SageMaker jumpstart LAma2 model produced the most accurate and high-performing results.

By integrating the RAG methodology, the Agent is able to reason about the user query and search customer data to find the most relevant pieces of information to answer completely and accurately to the question. In this way, the chatbot is capable of satisfying user query without the intervention of a human person.

The system also exploits the Agent capability to call APIs to build the shopping cart in a completely innovative way, allowing clients to simply type the items and quantities they need. Additionally, it is able to perform complex reasoning tasks such as finding the cheapest product or the one the highest protein count.
This innovation saves the user time by eliminating the need to scroll through all the products to choose what to buy.
The chatbot can personalize user interactions, allowing to ask for suggestions such as *Create a cheap protein-rich breakfast* and receiving a cart with the necessary products. Also, the shopping assistant can provide personalized product recommendations and offer the option to request a recipe, automatically adding the necessary products to the cart.

Possible Improvements The designed project solution allows to benefit of the technological features offered by the RAG and Agent technology. Anyway, it comes

with some trade off and possible improvements:

- **Agentic RAG**: The developed system successfully retrieve the correct documents from the data source to generate the answer to the user query. This approach avoid the time and costs needed to fine-tune a model to answer the specific questions about a particular domain.
  Nevertheless, a more recent and advanced approach can be followed. *Agentic RAG* systems leverages an Agent to iterate over the retrieval of documents from a vector storage. The iteration phase increase the quality of the context generated and guarantee only the necessary documents to answer a query are retrieved.
  With the current RAG solution we would need to develop an additional system to evaluate the accuracy of the generated answers. In contrast, an Agentic RAG can assess if the documents retrieved can answer the query, decide to retrieve more or determine the answer is not present in the context provided.

- **Guardrails**: In a system of this kind we need to control the chatbot answers to be sure they align to what we expect. In particular it must not generate text about specific topics or about something it doesn't know.
  For the project we designed a complex prompt explaining what the system should not answer. Anyway this is not a structured solution.
  The Bedrock service offer a new feature called *Guardrails* exactly for this scope. It is possible to define the unwanted behaviour in terms of topics or words, and the guardrail will intervene and possibly mask the answer. Moreover, using the Grounding and Relevance analysis it will evaluate himself on how much the generated answer aligns to the ground truth and to the query, automatically discarding answers under a predefined threshold.

- **Workflows**: One of the greatest limitation in the current project is represented by the time required to answer. Agents are very powerful, but slow in generating the answer. Due to the complexity of the orchestration prompt and the time required to choose the best action to answer a query, the average time to answer is around 10-15 seconds. Which, for some applications, is not acceptable.
  A possible alternative is represented by the design of predefined flow of actions, *Workflows*, which the generation model will follow. In this way it is guaranteed that all the steps required are performed before answering the query. Moreover, the time to answer decrease since the model just have to choose the workflow instead of thinking about the steps to reach a result.
  A possible implementation is offered by Bedrock Prompt Flows, enabling to define the flow of operations to perform and services to call for predefined inputs.

- **Batch Inference**: Another disadvantage of the RAG technology relies in the time and costs required to answer to many questions contained in a dataset all at once. For my experiments the costs for answering all at once all the questions was particularly high.

With the new *Batch Inference* option, Bedrock gives the possibility to decrease the costs by operating on the group of questions in batch, instead of the single ones.

Bibliography

[1] Fetzer, J.H. (1990). What is Artificial Intelligence?. In: Artificial Intelligence: Its Scope and Limits. Studies in Cognitive Systems, vol 4. Springer, Dordrecht. https://doi.org/10.1007/978-94-009-1900-6_1.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, pp. 5998–6008, 2017.

[3] C. Fregly, A. Barth & S. Eigenbrode *Generative AI on AWS*, O'REILLY, Venetia, 1612.

[4] Jared Kaplan et al., in "Scaling Laws for Neural Language Models", arXiv, 2020.

[5] Jordan Hoffman et al., in "Training Compute-Optimal Large Language Models", arXiv, 2022.

[6] Langchain, "Recursively split by character", https://python.langchain.com/v0.1/docs/modules/data_connection/document_transformers/recursive_text_splitter/.

[7] Langchain, "Semantic Chunking", https://python.langchain.com/v0.1/docs/modules/data_connection/document_transformers/semantic-chunker/.

[8] Meta, "FAISS", https://ai.meta.com/tools/faiss/.

[9] OpenSearch, "OpenSearch", https://opensearch.org/

[10] AWS, "Knowledge Bases for Amazon Bedrock", https://aws.amazon.com/it/bedrock/knowledge-bases/

[11] AWS, "Amazon Kendra", https://aws.amazon.com/it/kendra/

[12] AWS, "Anthropic's Claude in Amazon Bedrock", https://aws.amazon.com/bedrock/claude/.

[13] AWS, "Amazon SageMaker JumpStart", https://aws.amazon.com/it/sagemaker/jumpstart/.

[14] AWS, "Anthropic's Claude in Amazon Bedrock", https://aws.amazon.com/bedrock/claude/.

[15] Hugging Face, "Evaluate", https://huggingface.co/docs/evaluate/index.

[16] explodinggradients Repo, "RAGAs", https://github.com/explodinggradients/ragas.

[17] Confident Ai, "Deep Eval", https://aws.https://docs.confident-ai.com/.

Acknowledgements

A Davide e David per avermi accolto calorosamente nel gruppo di lavoro.
A Carlo per avermi seguito con passione e pazienza nel corso del progetto. Alle giornate passate a ragionare insieme su possibili evolutive perché si può sempre fare di meglio e per la precisione che mette in quello che fa.

Ai vecchi amici e compagni di classe con cui ho la fortuna di aver mantenuto i contatti.

Agli amici del mare, per le estati speciali.

Agli amici del poli, ai Caruggi.

A Manu, per esserci sempre.

Ai miei genitori per il supporto e l'aiuto in ogni momento e in ogni cosa. A mio fratello per sopportarmi quando lavoro da casa.

Alla mia famiglia, per tutto.