Universidad Politécnica
de Madrid

Escuela Técnica Superior de

Ingenieros Informáticos

Master in Digital Innovation: Data Science

Large Language Models to Query Your Data: Retrieving Ads Industry Users Data Using Natural Language

Autor: Alessandro Rossi

Supervisor: Marta Patiño Martinez

Madrid, 15/07/2024

This Master Thesis has been deposited in ETSI Informáticos de la Universidad Politécnica de Madrid.

Master Thesis

Master in Digital Innovation: Data Science

Title: Large Language Models to Query Your Data: Retrieving Ads Industry Users Data Using Natural Language

July/2024

Author: Alessandro Rossi

Supervisor:

Marta Patiño Martinez

Abstract

Generative AI has seen a notable increase in use over the past few years, changing various fields from natural language processing to creative arts. This change is mainly due to the introduction of the Transformer model in 2017, which improved the handling of sequential data. Recently, Large Language Models (LLMs) like OpenAI's GPT-4 have gained popularity, showcasing their potential to automate and improve many applications.

Advertising, particularly digital advertising, has evolved with the rise of the Internet and social networks. Companies now use various media like search engines and web banners. However, managing advertising budgets across multiple platforms can be challenging and inefficient. To help solve this problem, MINT introduced Advertising Resource Management (ARM), a software platform designed to consolidate advertising operations. ARM's platform uses real-time data analysis and predictive insights to help businesses better manage their digital advertising efforts. Despite its benefits, the complexity of the current web application can make it difficult for users to engage with the platform.

This thesis aims to improve user interaction with the ARM platform by introducing an AI assistant powered by LLMs. The assistant allows users to ask questions in natural language, which are then converted into SQL queries to retrieve relevant data. The study evaluates various LLMs, including those from OpenAI, Google, and Amazon, for their performance in this task. A new safety model is also proposed to detect potentially harmful requests, adding an extra layer of security.

The implementation starts with refining the TextToSql functionality and then building the complete system architecture. This includes integrating a dynamic example retrieval method using the Max Marginal Relevance algorithm to enhance the accuracy of generated SQL queries. The AI assistant architecture is designed to be scalable and adaptable, allowing for easy updates and improvements.

Results show significant improvements in query accuracy and user engagement, making data retrieval more straightforward and efficient. This thesis demonstrates the practical benefits of LLMs in digital advertising, suggesting potential for future improvements in AI-driven user assistance technologies.

Resumen

La inteligencia artificial generativa ha experimentado un notable aumento en su uso en los últimos años, transformando varios campos, desde el procesamiento del lenguaje natural hasta las artes creativas. Este cambio se debe principalmente a la introducción del modelo Transformer en 2017, que mejoró el manejo de datos secuenciales. Recientemente, los Modelos de Lenguaje Grandes (LLMs), como GPT-4 de OpenAI, han ganado popularidad, demostrando su potencial para automatizar y mejorar muchas aplicaciones.

La publicidad, especialmente la digital, ha evolucionado con el auge de Internet y las redes sociales. Las empresas ahora utilizan varios medios, como motores de búsqueda y banners web. Sin embargo, gestionar presupuestos publicitarios en múltiples plataformas puede ser un desafío e ineficiente. Para ayudar a resolver este problema, MINT introdujo la Gestión de Recursos Publicitarios (ARM), una plataforma de software diseñada para consolidar las operaciones publicitarias. La plataforma ARM utiliza análisis de datos en tiempo real y conocimientos predictivos para ayudar a las empresas a gestionar mejor sus esfuerzos de publicidad digital. A pesar de sus beneficios, la complejidad de la aplicación web actual puede dificultar la participación de los usuarios.

Esta tesis tiene como objetivo mejorar la interacción del usuario con la plataforma ARM mediante la introducción de un asistente de IA impulsado por LLMs. El asistente permite a los usuarios hacer preguntas en lenguaje natural, que luego se convierten en consultas SQL para recuperar datos relevantes. El estudio evalúa varios LLMs, incluidos los de OpenAI, Google y Amazon, por su rendimiento en esta tarea. También se propone un nuevo modelo de seguridad para detectar solicitudes potencialmente dañinas, añadiendo una capa adicional de seguridad.

La implementación comienza refinando la funcionalidad de TextToSQL y luego construyendo toda la arquitectura del sistema. Esto incluye la integración de un método dinámico de recuperación de ejemplos utilizando el algoritmo de Relevancia Marginal Máxima para mejorar la precisión de las consultas SQL generadas. La arquitectura del asistente de IA está diseñada para ser escalable y adaptable, permitiendo actualizaciones y mejoras fáciles.

Los resultados muestran mejoras significativas en la precisión de las consultas y la participación del usuario, haciendo que la recuperación de datos sea más sencilla y eficiente. Esta tesis demuestra los beneficios prácticos de los LLMs en la publicidad digital, sugiriendo un potencial para futuras mejoras en las tecnologías de asistencia al usuario impulsadas por IA.

Contents

List of Tables

List of Figures

1 Introduction

Generative AI has witnessed a remarkable rise over the past few years, fundamentally transforming various domains from natural language processing to creative arts. This rise can be attributed to several key advancements and trends in artificial intelligence and machine learning. The concept of generative AI is not new, but its practical applications have significantly evolved due to advancements in neural network architectures, particularly with the introduction of the Transformer model in 2017(37). The Transformer model revolutionized the way machines process sequential data by allowing for more parallelization and efficient handling of long-range dependencies compared to its predecessors like Recurrent Neural Networks (RNNs)(21) and Long Short-Term Memory (LSTM)(17) networks.

In recent years, the development and scaling of Large Language models (LLMs) such as OpenAI's GPT-4(31) have brought generative AI into mainstream attention. These models, with billions of parameters, are trained on several datasets, enabling them to generate coherent and contextually relevant text across various topics. The capability of LLMs to perform a wide range of tasks from text generation to translation without task-specific training has demonstrated the potential of generative AI to automate and enhance numerous applications. The recent rise in popularity of Large Language Models can be largely attributed to the accessibility and integration of these models into user-friendly platforms and APIs. Companies such as OpenAI and Google have provided tools that allow developers to incorporate generative AI into applications easily, thereby broadening its usage and demonstrating its utility in real-world scenarios. Moreover, the advent of technologies like Retrieval-Augmented Generation (RAG)(25) has further enhanced the capabilities of generative AI by combining the strengths of retrieval-based models and generative models. This hybrid approach enables more accurate and contextually aware responses by fetching relevant information and generating natural language outputs based on it.

As a result, generative AI has become a crucial technology in various industries, including digital advertising, content creation, and customer service, to name a few. The continuous improvements in model architectures, training techniques, the almost infinite amount of computing power thanks to cloud providers and integration capabilities are likely to drive even more innovative applications of generative AI in the future.

Advertising is the business of trying to convince people to buy products or services(14), and with the advent of the Internet, especially with social networks, it has evolved into the so-called 'Digital Advertising.' The term Digital encompasses several media such as search engines, social networks, banner advertisements on web pages, and many others. Even today, the majority of companies hire several people who have to allocate day by day, manually, the budget for a specific platform (e.g., Google ads or Facebook); this task can become very complex and difficult to organize with the high number of the platforms, leading to possible mismanagement of the budget on platforms that do not achieve the objectives set or, on the contrary, an underspending on those that could reach much higher performances.

To address this specific problem, MINT introduces a new category of software, non-existent before, called Advertising Resource Management (ARM)(1) to provide a platform specialized in "helping organizations consolidate their resources, their processes, workflows, and information into a single system to better support their

advertising operations." MINT offers a unique solution for businesses looking to improve their digital presence with its creative approach to digital advertising, which is customized for the ever-changing online marketing landscape. With its emphasis on real-time data analysis and predictive insights, this ARM's platform differs from standard media mix modeling tools that mostly rely on historical data, empowering companies to make proactive and well-informed decisions thanks to the introduction of artificial intelligence solutions such as recommender systems to suggest the best budget allocation for each advertising campaign, or even priorly suggesting which channels or platform is best to invest based on previous campaigns. This is especially crucial in the quick-changing digital world where customer behavior and market conditions may vary rapidly.

1.1 Problem

The ARM platform is a great advancement in digital advertising; however, new technologies can be challenging for users, and the current design of the web application can make it difficult to use. Users might have to select various check boxes and fill various fields even to visualize simple data. This can lead to users losing interest and not engaging with the platform during the onboarding process, which can hurt the company business. Therefore, this project aims to offer a new and more intuitive way for all users to interact with the platform.

To better visualize this problem, in Figure 1, it is possible to visualize the high amount of interactions the user needs to perform to extract the data is asking for. In particular in the upper part of the image, there are both the Filter and the Show applied to this particular user's query; to obtain this result the user each time has to apply manually a filter, firstly selecting the time range (in this case last 3 months) then selecting the specific campaign name from a list containing all the campaign names and finally selecting the channel names. Then the user has to specify which data to visualize, so from the drop-down menu in the right column named MEASURE, the user manually flags the desired fields (in this case, clicks, impressions, and Budget spent). Then all these filters and selections create a SQL query executed on the company datalake; the final result is the one shown, with a lot of information without any explanation or summary.

Just as convoluted as this explanation of the operations may seem, so is the user's experience with this data exploration tool. This experience can take some minutes to structure all the necessary filters and selections to extract the desired information. Therefore, providing the user with a more intuitive way to extract information is essential for maintaining a high level of engagement and assuring the user to remain on the platform as much time as possible.

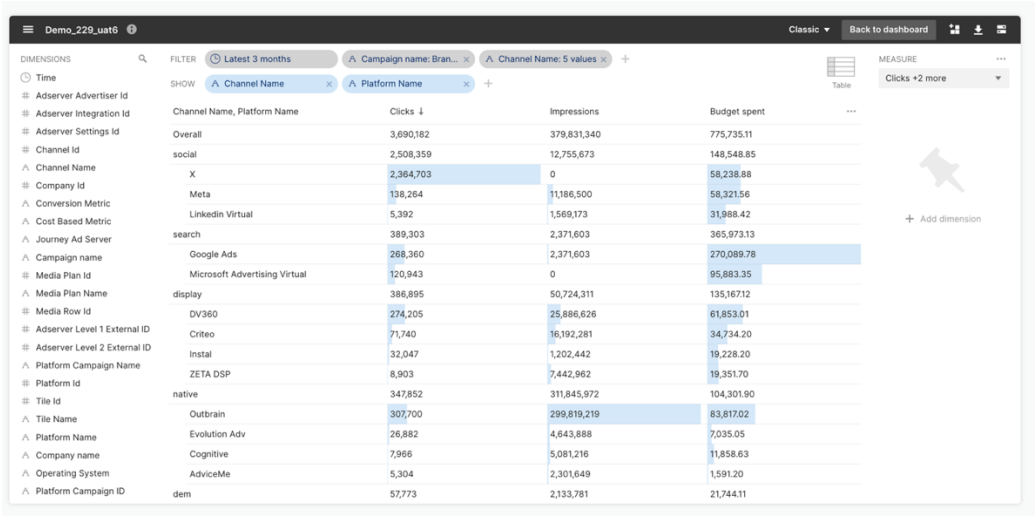

Figure 1: Platform screenshot showing the platform not being user friendly

1.2 Purpose

As mentioned before, this work aims to provide the platform customers with a new user experience to be added to, not replace, the current one. To achieve this, the increasingly popular technology of Generative AI, specifically Large Language Models (LLMs), has been chosen to develop an AI assistant that can convert a natural language request into an action as if it had been made using the current User Interface (UI). In particular, the use case that this work will focus on most is data exploration, where the assistant, given any user request, must be able to automatically query the company database to provide all the requested data; in other words, this can be translated into a text-to-SQL because the request in natural language must be converted into an SQL query to retrieve the requested information from the company database; so instead of clicking and selecting the user should only type its request. Revisiting the example shown in Figure 1, imagine how much quicker and more satisfying it would be if the user could extract the same data by simply typing, "Show me the clicks, impressions, and total spent for the campaign named XYZ in the last 3 months," without any additional effort, showing the result in just few seconds.

2 Theoretical Background

This section presents all the theoretical concepts laid behind the LLMs; from the explanation of the Transformers Architecture to the most recent architectures of LLMs and how are trained; presenting services available on some cloud providers that have been used to develop this work.

2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a type of artificial neural network, proposed at the beginning of the 80's (21), designed for processing sequences data. Unlike feedforward neural networks, RNNs have connections that form directed cycles, allowing them to maintain a form of internal state or memory. This characteristic makes them suitable for tasks such as natural language processing, time series analysis, and speech recognition, where the context or the order of data points is essential. A RNN processes a sequence one element at a time, maintaining a hidden state vector h_t capturing information from all previously seen elements. The basic formula for updating the hidden state in a vanilla RNN is given by (21):

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

Where x_t is the input vector at the time step t, W_{hh} is the weight matrix connecting the previous hidden state to the current hidden state, W_{xh} is the weight matrix connecting the input to the hidden layer, and b_h is the bias.

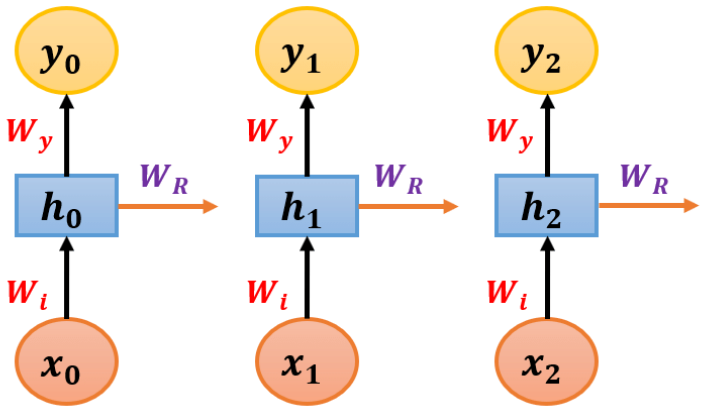

Figure 2: RNN mathematical representation (51)

The Figure 2 provides a visual example that complements the mathematical description provided in the second image. It showcases the sequence of operations in a Recurrent Neural Network (RNN) over three time steps. In this example, each block labeled h represents the hidden state at time t, while x and y represent the input and

output at time *t*, respectively. The arrows indicate the flow of information and the weights applied during this process. Specifically:

- The arrows labeled W_i illustrate the weight matrix W_{xh}, which connects the current input x_t to the hidden state h_t.
- The arrows labeled W_r represent the weight matrix W_{hh} which connects the hidden state from the previous time step h_{t-1} to the current hidden state h_t.
- The arrows labeled W_y show the connection from the hidden state h_t to the output y_t.

This neural network suffers from the major problem called **Vanishing gradient** in which the gradient, during backpropagation, tends to assume smaller and smaller values making it impossible to update the first weights of the network; this problem leads to slower training but above all poor generalization of the model, making this type of architecture hard to use on real-world scenario.

2.2 Long Short-Term Memory

Presented for the first time in 1997 by Hochreiter and Schmidhuber (17), Long Short-Term Memory models (LSTM) overcame the intrinsic limitations of RNNs proposing a separate cell state alongside the hidden state and introducing gate mechanisms; the architecture of this type of neural network is showed in Figure 3. The key to the memory capacity of LSTM over time is its gate mechanism. There are three types of gates in an LSTM (50):

• **Forget Gate**: This gate determines which information to discard from the cell state. It examines the current input x_t and the previous hidden state h_{t-1}, to generate a gate output. This output is a series of numbers between 0 and 1, produced by applying a sigmoid function. A value close to 1 suggests "retain this information," while a value close to 0 indicates "discard this information".

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

- **Input Gate**: This gate updates the cell state with new information. It has two parts: a sigmoid layer called the "input gate layer" which decides which values will be updated, and a tanh layer which creates new candidate values that could be added to the state

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

- **Output Gate**: it controls the output stream of information from the memory cell to other LSTM blocks. It determines what output to generate from the cell memory.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

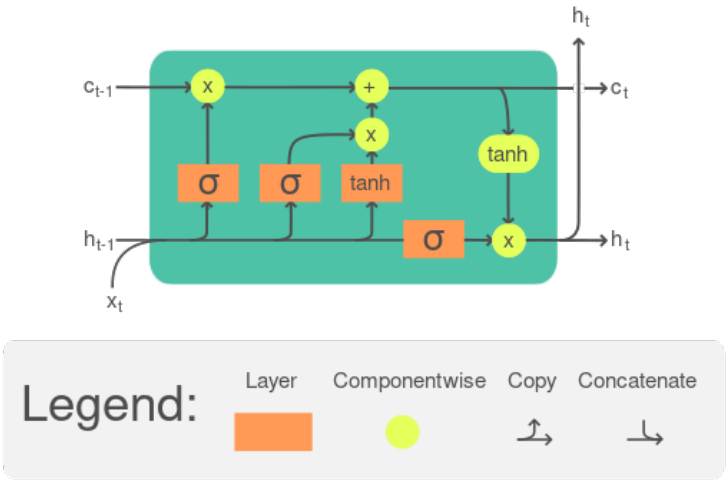

Figure 3: LSTM architecture (45)

Thanks to these three mechanisms, LSTMs became the state of the art for tasks involving sequence like translating, for example the Google Neural Machine Translation (a refining for their translation service) was composed of two main blocks, an encoder and a decoder, both based on LSTM (39). Still, LSTMs suffer from "amnesia" on very long sequences, this difficulty arises from the complex interactions and nonlinearities within the LSTM structure, which can lead to attenuated gradients and interfere with the LSTM ability to learn dependencies over extremely long distances. To better understand this problem

2.3 Transformer

Presented for the first time by Google in the popular paper "Attention is All You Need" (37), the transformer architecture represents a significant departure from previous sequence processing models like RNNs and LSTMs. It has quickly become the foundation for modern natural language processing (NLP) and beyond due to its innovative structure and performance advantages. Transformers abandoned the idea of recurrent connection for a new concept called **Attention Layer**, allowing the model to focus on the most relevant information for a given task. Figure 4 shows the original architecture that was presented in the paper; it has two main parts: the encoder (left part) and the decoder (right part). In particular:

- Encoder
  - takes the input text and turns it into a form the model can understand. It first converts the words into numbers (embeddings) and then adds information about their positions in the sentence.
  - Each layer of the encoder has two steps: it looks at all the words to see which ones are important to each other (self-attention) and then processes this information (feed-forward network). After each step, it adds the original information back in (residual connection) and normalizes it.
- Decoder

- The decoder takes this processed information from the encoder and uses it to generate the output text.
- Each layer of the decoder has three steps: it looks at the output words generated so far (masked self-attention), checks the encoder information to find relevant parts (multi-head attention), and processes the combined information (feed-forward network). Like the encoder, it also adds the original information back in and normalizes it after each step.
- Finally, it turns the processed numbers into probabilities of the next word in the sequence.

This architecture allows the transformer to handle language tasks very efficiently, focusing on important parts of the text and processing everything in parallel. Compared to older models, this makes it much faster and better at understanding long sentences. The next sections will explain some of the key components of this architecture.

2.3.1 Attention Layer

This layer was the main innovation in the transformer architecture; an attention mechanism involves transforming a query along with key-value pairs into an output, with all elements represented as vectors. To provide a better explanation of the attention layer, it is useful to explain the three core components behind this layer (37):

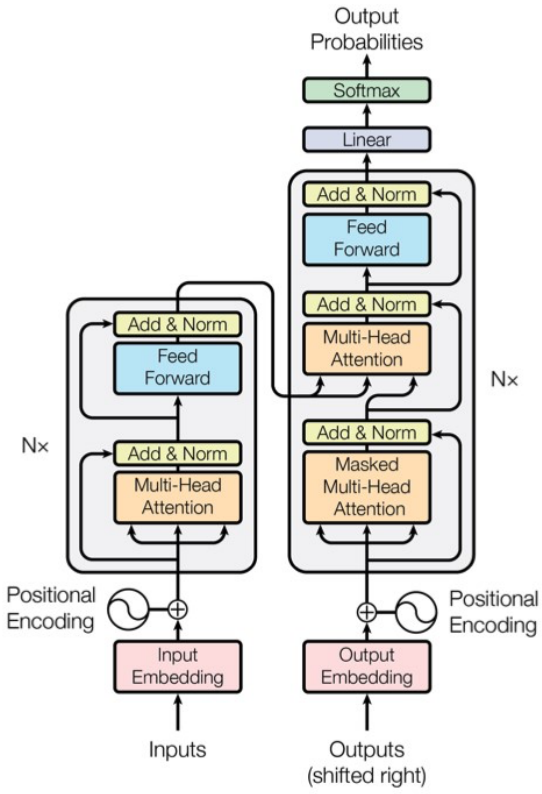

Figure 4: Transformer architecture (37)

- **Query**: This vector represents the current item that the model is trying to understand. For instance, in language translation, a query could be the embedding of a word in the sentence being translated.
- **Key**: These vectors correspond to the elements in the input sequence that the model will use to provide context to the query. Each key is associated with a value. In the translation example, the keys could be the embeddings of all the words in the input sentence.
- **Value**: Value vectors are paired with keys and contain the actual information that the model needs to complete the task. When a query is compared to the keys, the resulting attention weights determine how much each value will contribute to the output. The paper refers to this operation as Scaled Dot-Product Attention, as shown in Figure 4 and the operation is the following:

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

In practice, the result of the attention layer is a matrix containing for each word in the input sequence, the amount of attention that the input word taken in consideration must place on every other word (and on itself). Note that the scaling factor dk (key vector size) has been introduced to contain the gradient value during training.

2.3.2 Multi-Head Attention Layer

In the same paper, an improved version of the attention layer is presented, called the multi-head attention layer, as shown in Figure 5 (46), to enhance the effectiveness of the attention layer in two significant ways:

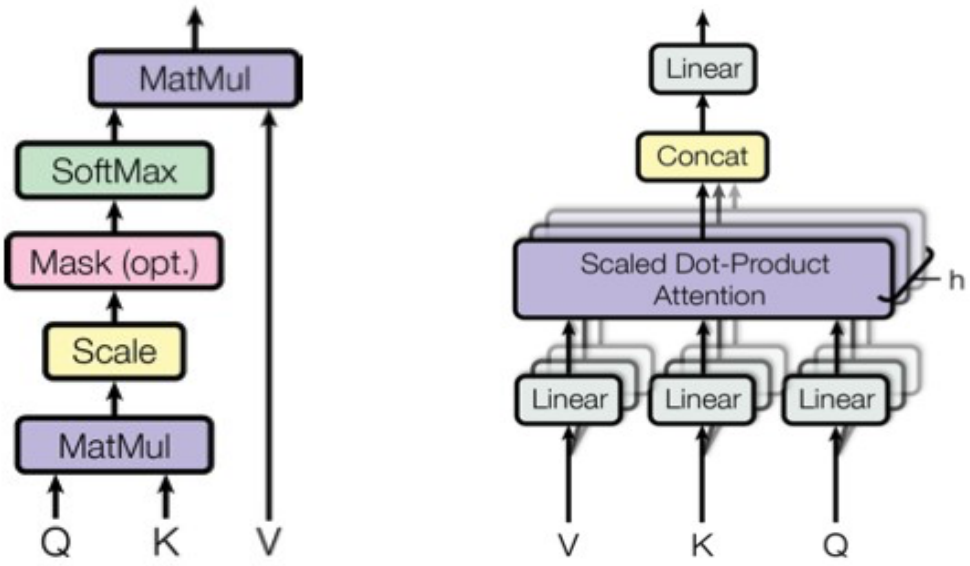

Figure 5: Multi-Head Attention (left) & Scaled Dot-Product Attention (right)

- Firstly, it broadens the model capacity to concentrate on various positions of the input sequence
- Secondly, it equips the attention layer with several "representation subspaces". With multi-headed attention, there are numerous sets of Query/Key/Value weight matrices. These sets begin with random initialization, and, post-training, they serve to map the input embeddings onto varied representation subspaces.

As before, this layer can be summarized with this formula:

$$MultiHead(Q,K,V) = Concat(head_1, \dots, head_h)W^O$$
$$where\ head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

In the multi-head attention mechanism, several projection matrices are utilized, specifically W_i^Q, W_i^K, W_i^V, for each head i, and W^O for the final output transformation:

- W_i^Q, W_i^K, W_i^V are the projection matrices for the i-th head of the multi-head attention mechanism:
  - W_i^Q projects the query input Q into a new vector space, facilitating the computation of attention in terms of relevance to the keys.
  - W_i^K projects the key input K, aligning it to interact with the queries in the computed attention.
  - W_i^V projects the value input V, determining the content that is highlighted by the attention weights.
  - W^O is used after the attention heads are concatenated: This matrix combines and transforms the concatenated outputs of all heads into a single vector per input position, effectively integrating the different informational aspects captured by each head.

In Figure 6 is possible to visualize better how the multi-head attention layer works:

1. Input Projection: in the input projection, the input X is projected into queries Q_i, keys K_i, and values V_i using the matrices W_i^Q, W_i^K, W_i^V
2. Attention Heads: each head calculates attention using the formula $Attention(QW_i^Q, KW_i^K, VW_i^V)$ which involves computing attention scores and using them to weigh the values.
3. Concatenation and Output: The outputs of all heads are concatenated and transformed with W^O to produce the final output.

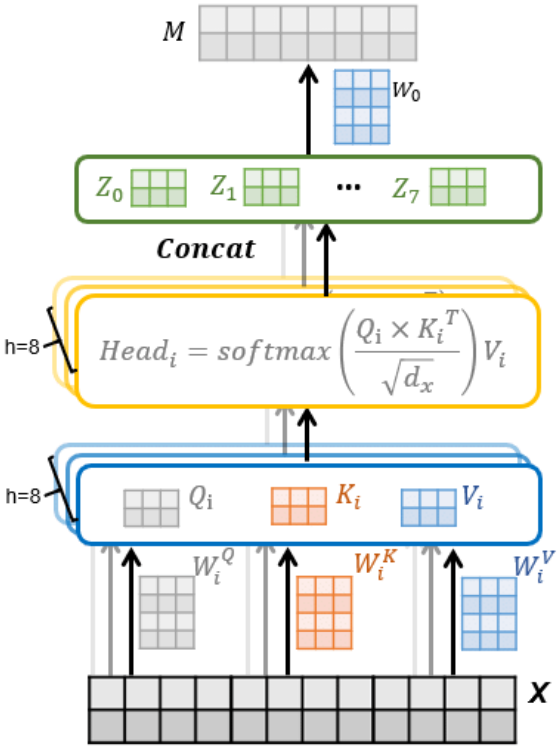

Figure 6: Multi-head attention layer visualized (8 heads) (52)

2.3.3 Word Embeddings

Word embeddings are a foundational component in the field of natural language processing. Before this approach, everybody could think about encoding each word uniquely; this technique is called **one-hot-encoding**. This encoding method, for every word, assigns each word in a vocabulary a unique vector, where all elements are zero except for a single '1' at the index of the word. This approach is not feasible for large vocabularies because it results in high-dimensional, sparse vectors that consume a lot of memory and do not capture any semantic relationships between words, making them inefficient for tasks that require understanding of word meanings or contexts. This is the reason why word embedding has been proposed; in word embedding, each word in a language is mapped to a lower-dimensional continuous vector space, as shown in the example in figure 7. The embeddings capture semantic and syntactic meanings of words, allowing machines to understand text in a way that mirrors human language capabilities. These representations enable various NLP models, including transformers, to perform tasks such as translation, summarizing, and sentiment analysis more effectively. In transformers, the initial step still involves mapping each word to a vector. These initial vectors are similar to traditional embeddings but are adapted according to the architecture of the transformer. Typically, these are learned directly from the data during the training process of the transformer model itself rather than being pre-trained and fixed.

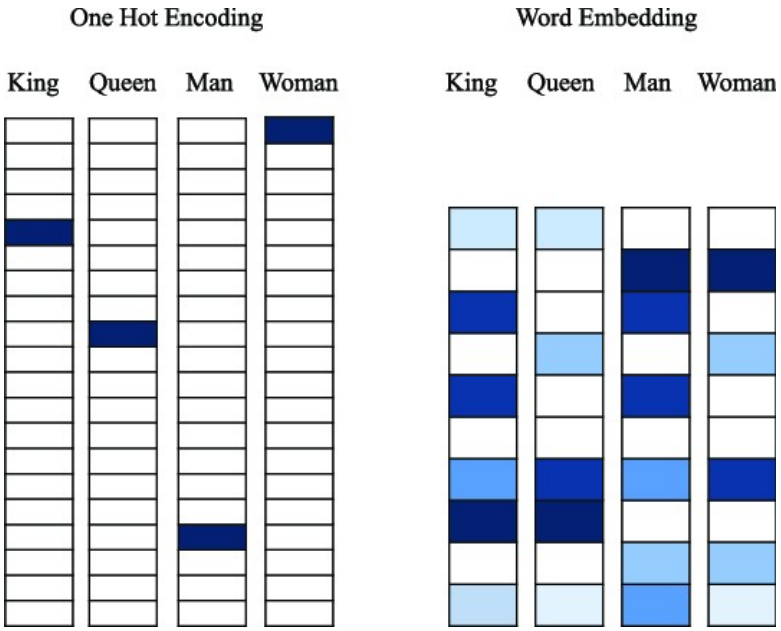

Figure 7: One-hot-encoding vs Word Embedding (47)

2.3.4 Positional Encoding

Since this architecture does not contain any recurrence or convolution, it cannot understand the words' position in the sequence. That is why a position encoding layer has been added both at the input of the encoders and decoders; these encodings have the same dimension as the embeddings so that the two can be summed. Among the several possibilities, the original paper uses a combination of sine and cosine functions:

$$PE_{(pos,2i)} = sin\left(pos/10000^{2i/dmodel}\right)$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/dmodel})$$

where pos is the position and i is the dimension.

This transformer architecture has gained a lot of popularity, especially in those services where dealing with a sequence is fundamental; for example, the famous DeepL translator is based on transformers (13). Further experiments aimed to improve performance showed that using just the encoder or the decoder could greatly increase effectiveness on specific tasks, while cutting the size of the model in half. Remaining always within the scope of LLM, it has occurred that the transformer decoder alone is very effective on generation tasks and so ideal for the purposes of Large Language Models.

2.4 Decoder-Only Transformers

This variant of the classical transformer architecture is nowadays the starting point for all the LLMs like ChatGPT, Gemini, and so on; presented for the first time in 2018

from Google Brain (14) (Google AI research team), this architecture ”discards” the encoder part of the original transformer model keeping only the decoder one; by doing so the amount of parameters that have to be trained is halved, making the training and inference much faster.

Unlike their encoder-only counterparts that specialize in understanding context, decoder-only models delve into the art of creative generation. These models are designed to take a sequence of tokens and predict the next token in the sequence, step by step, crafting coherent and contextually sound outputs.

2.4.1 Autoregressive Nature of Decoder-Only Transformers

An autoregressive model predicts future elements in a sequence based on previous elements without any external input after the initial condition. In the context of decoder-only transformers, this means that the model generates text one token (e.g., a word or a character) at a time, using the previously generated tokens as context for each subsequent prediction. The autoregressive characteristic of Decoder-Only transformers arises due to the occurrence of two events during inference:

- **Sequential Prediction**: The prediction is built one element at a time, relying on the previously predicted elements. This sequential nature mimics how humans build sentences word by word.

- **Causal Attention**: In decoder-only transformers, the attention mechanism looks at past outputs only during prediction. This ensures each prediction depends solely on what came before, fostering a left-to-right, autoregressive flow (33)

As anticipated before, this architecture, with some slight changes (like the number of decoders, skip connections, and so on), is the baseline on which the current LLMs are trained on (34).

2.5 Sparse Mixture Of Experts (SMoE)

Mixtures of Experts (MOE) is a concept idealized way before the introduction of attention mechanism and transformers; it was proposed for the first time in 1991 (19); the idea introduced a machine learning model that consists of multiple expert models and a gating network, as it is shown in the Figure 8. Each expert model is trained to specialize in a different part of the input space, meaning that they become proficient at handling different data types or tasks related to the overall problem. The gating network is responsible for determining which expert should be applied to any given input.

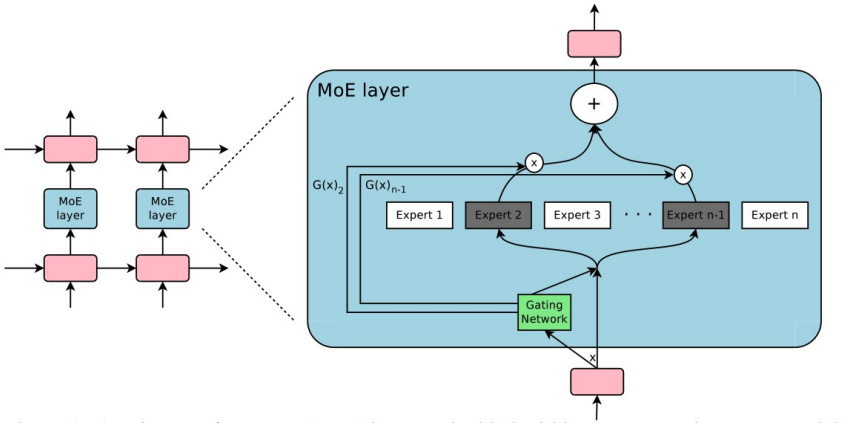

Figure 8: MOE layer from Outrageously Large Neural Network paper (35)

Sparsity leverages the concept of conditional computation; unlike dense models, where every parameter is applied to every input, sparsity permits the operation of only certain sections of the entire system; indeed, in a MOE, not every expert is required for each input, and thanks to that, batch sizes in MOEs are effectively reduced as data flows through the active experts. But this comes with a significant challenge; for instance, if there is an input batch of 10 tokens, five tokens may be processed by a single expert, while the remaining five tokens could be distributed across five different experts, resulting in uneven batch sizes and reduced utilization. That is why a learned gating network is required:

$$y = \sum_{i=1}^{N} G_i(x) E_i(x)$$

where G is the gating operator, and E is the actual expert. For simplicity, let's consider the gating function in its most standard version:

$$G(x) = softmax(x * W_g)$$

It is clear to notice that in this configuration, all experts receive all the inputs as a softmax weighted multiplication.

2.5.1. Switch Transformers

The Switch Transformer is a switch feed-forward neural network (FFN) layer that replaces the standard FFN layer in the transformer architecture. The critical difference is that instead of a single neural network, each switch layer contains multiple FFNs, known as experts, as shown in Figure 9. Google used this architecture to develop its GShard model, a massive 1.6 trillion-parameter LLM (24)

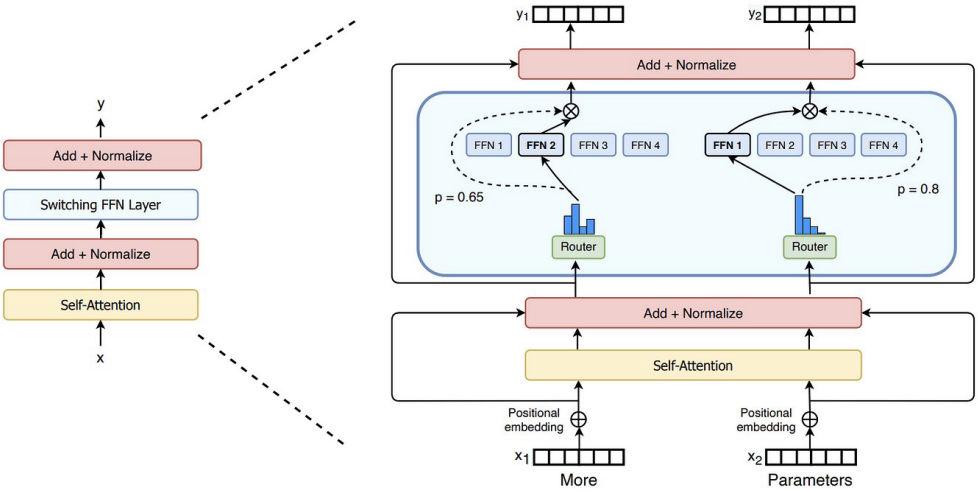

Figure 9: Switch Transformer Layer (15)

2.6 Large Language Models

In general, there is not a definitive description of a large language model (LLM). Typically, an LLM is a vast transformer model specifically designed and trained to understand, generate, and interact with human language on a large scale. The critical difference between an LLM and a standard transformer model lies in their size and the data on which they are trained. LLMs, such as OpenAI's GPT-3, which has 175 billion parameters (23), can store more information and learn more detailed patterns in the data due to their enormous size. In contrast, the original transformer models have significantly fewer parameters, with the foundational model having about 200 million (37). Another significant difference is the type of training data used. LLMs are trained on large and diverse datasets that cover a wide range of topics, styles, and languages (6). This extensive training helps them develop a broad understanding of language and the world, enabling them to perform well across various tasks, even those for which they were not explicitly trained. On the other hand, classical transformers are usually trained on more specialized tasks like translation or question-answering using smaller, more focused datasets. As a result, their ability to generalize is not as pronounced as that of LLMs.

2.7 Training a LLM

Each LLM can be trained in a completely different and personalized way than any other, but most of the current LLMs representing the state-of-the-art (both open and closed source models) follow a defined training process. The explanation of a Large Language Model training process is based on the training of LLAMA-2-70B-Chat, an open-source LLM specialized in dealing with users with continuous chat. (36)

2.7.1 Self Supervised Learning

In this stage, a transformer model with randomly initialized weights is trained using sentences from web documents. With this training, the model learns word

connections by filling in missing parts of sentences. The training data typically comes from public sources, avoiding websites that contain many personal information about individuals. Even this basic training can produce a model capable of handling user requests effectively. For example, the GPT-3 model was developed through this type of self-supervised learning. Generally, the more varied and richer the dataset used, the better the performance of the resulting large language model.

2.7.2 Supervised fine-tuning (SFT)

In this phase, the model is given more examples that closely align with the ultimate objectives of the large language models. These examples further refine the model capabilities (for instance, in Llama2-Chat, these examples typically include pairs of prompts and responses). As demonstrated by Zhout et al. (41), in this phase, it is preferable to obtain high-quality SFT data, even in less quantity than in the supervised learning phase, because it is enough to achieve a high-quality result (for example, Llama2 chat was given just 27.540 annotations). After this fine-tuning, the model is much more capable of answering as desired, but this step can require more data than the one achievable online, so a possibility to margin this problem is to actively hire several human annotators that, given a prompt as input, write down the desired answer; once that enough annotations are collected, the LLM can be further fine-tuned.

2.7.3 Reinforcement Learning with Human Feedback (RLHF)

Reinforcement Learning with Human Feedback (RLHF) is a method used in training an already fine-tuned language model. Its purpose is to better align the model actions with human preferences and the ability to follow instructions. It is composed of two main steps:

- **Reward model training**: This part aims to train a model that, given a text as input (the prompt) and a response for that text, can provide the latter a scalar score that should be aligned with the human preference. To do this, a human is given a prompt and different answers from different models to rank each answer from the best to the worst. Then, the reward model is trained on this ranking of preferences, learning what a human being is expected for each prompt. It is reasonable to think people should also give a scalar score to each answer, but this is hard to do in real life. People's different opinions make these scores inconsistent and messy. So, instead, using rankings to compare the results of many models results in more consistent opinions.

- **Reinforcement Learning**: In this step, the human contribution is removed to dynamically improve our LLM. Given a new prompt, the LLM generates an output, and the reward model assigns a scalar score to the answer. This score is then used to update the LLM using the PPO (Proximal Policy Optimization). PPO tries to increase the chances of the language model producing outputs that will receive higher rewards in the future.

This method is used by some major players like OpenAI and Meta. Unfortunately, due to its high complexity and cost, it is not uncommon for an LLM to be trained up to the supervised fine-tuning stage. However, surely, the models trained with RLHF reach the best performances when dealing with users through a chat.

2.7.4 LLMs' parameters

We can adjust various parameters via API calls to exploit LLM full potential and tailor their responses to specific needs. Understanding these parameters and how they influence the model behavior is essential for getting the desired output. The main parameters that can usually be adjusted are the following one (12):

- **Temperature**: parameter that controls the randomness of the model output, affecting how predictable or creative the generated text will be. The model produces more predictable and focused responses at lower temperatures by prioritizing the most probable words. This is useful when you need precise and deterministic answers. Instead, the model generates more diverse and creative responses at higher temperatures by considering a wider range of possible words. This adds variability and can be helpful for creative writing or brainstorming.

- **Top-K**: This parameter limits the model to consider only the top k most probable next words, adding a controlled level of randomness to the output. For example, Top k = 1 makes the model always select the single most likely next word.

- **Top-p** selects the smallest set of words whose cumulative probability exceeds a specified threshold p. This method dynamically adjusts the number of words considered based on their probabilities. For example, top p = 0.9: The model includes words in the consideration set until their combined probability reaches 90%. This approach balances predictability and creativity by adapting to the distribution of word probabilities.

- **Max Tokens**: set the maximum number of tokens (words or word pieces) the model can generate in the response. This parameter helps control the length of the output (and so the cost of the API call).

2.7.5 Prompt Engineering

Prompt engineering is a critical aspect of interacting with generative AI models. It involves carefully crafting input text that these models can interpret and understand to perform a specific task or generate a desired output. The input text, known as a "prompt," is a set of instructions or questions that guides the AI in producing text, images, audio, or code that aligns with the user's intentions. Some LLMs in their model card (a document that provides essential information about the model capabilities, design, and intended use) provide some examples of valuable prompts that have been proven to lead to a performance improvement; for instance, Mixtral(20) is suggested to be used with the following prompt:

<s> [INST] Instruction [/INST] Model answer</s>

[INST] Follow-up instruction [/INST]

In the prompt format for a Mistral model, the <s>token marks the start and end of a text sequence, helping the model identify where to begin and finish processing text. The [INST] and [/INST] tokens enclose instructions, guiding the model on what to focus on or how to act during text generation. These tokens ensure the input is clearly

structured and the model responses are accurate and relevant to the given commands.

In addition to this particular prompt formatting that helps the LLM understand the request better, another possibility is to include in the prompt itself some examples of what we expect as an answer from the LLM; this technique is called few-shot learning and consists of adding to the prompt several examples of the question, answer, and it has been proven that this approach dramatically increases the performance of generative AI models(32)

2.8 Retrieval-Augmented Generation

Retrieval Augmented Generation (RAG) is a method that combines the strengths of a pre-trained large language model with an external database. This method uses two main parts: a retriever and a generator. The retriever works like a smart search engine, finding the most relevant documents or sections from a large database based on a user's question. The generator, which is usually a sophisticated large language model (LLM), then uses this information to create responses that make sense and fit the context. (25) The main benefits of using RAG include:

- **Enhanced Accuracy**: RAG improves the accuracy of AI-generated answers by using reliable sources from an organization's own knowledge.
- **Mitigating Biases**: RAG reduces biases found in typical training data by using varied and specific information, leading to fairer and more neutral responses.
- **Tailored Responses**: RAG allows AI models to adapt their answers to fit specific tasks and the needs of users, giving more useful and applicable insights.
- **Up-to-date Information**: By constantly updating its database with the latest information, RAG ensures that its responses are always current and relevant to new situations.

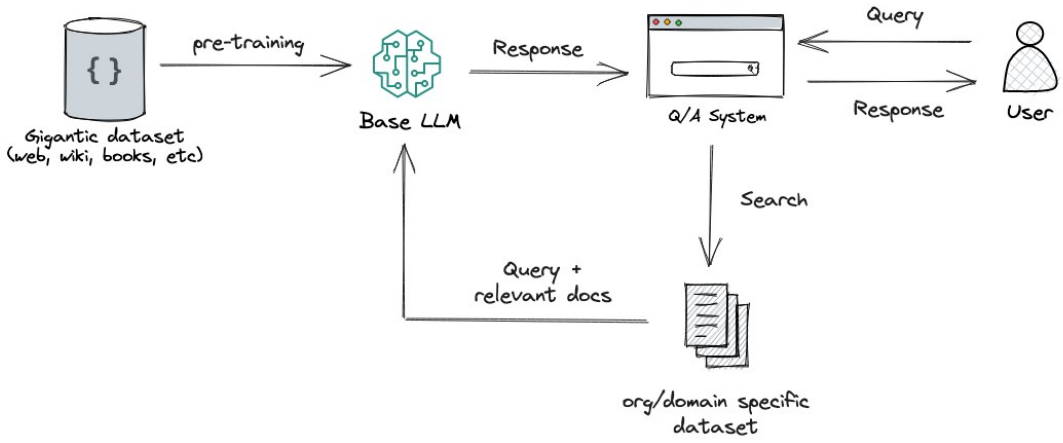

Figure 10: Typical schema of a RAG (18)

In practice, setting up a RAG system involves several practical steps. First, it is necessary to build or choose a knowledge base, which can be anything from a database of scientific articles to a collection of company reports. This knowledge base must be structured in a way that makes it searchable for the retriever. Next, the retriever is trained or configured to search through this knowledge base efficiently to find relevant information based on incoming queries. This might involve techniques like keyword search or more complex natural language processing algorithms. Once the relevant information is retrieved, the generator, a pre-trained language model, takes over. It reads and interprets the information provided by the retriever to craft answers that are informed by the data. The generator needs to be capable of understanding context and generating natural language, adjusting based on the data it receives; to better understand it, a schema of this method is shown in Figure 10.

2.9 Amazon Web Services

Amazon Web Services (AWS) is a cloud computing platform that provides a vast array of services and resources for building, deploying, and managing applications and infrastructure. This section does not aim to exhaustively explain all the services offered by AWS, but in the following sections, the main services used to work on this thesis are going to be briefly described.

2.9.1 Amazon Bedrock

Amazon Bedrock is a serverless service providing API access to some of the more popular foundation models (FMs) currently available on the market, for example, Mixtral8x7B, LLAMA2chat, and Claude, but it also offers the model developed by Amazon itself called Titan. (7) The advantage of this service relies on the zero cost of initializing a model, as the only cost is the API call. This can be advantageous if a company does not expect a massive and constant use of the LLM because the alternative would be a 24-hour deployment of an FM with the inevitable cost that rises even if the use is much lower than the maximum capacity. The disadvantage of Amazon Bedrock is the limited variety of models that can be used, but with models that are added almost monthly, hoping for greater availability of FM in the coming months.

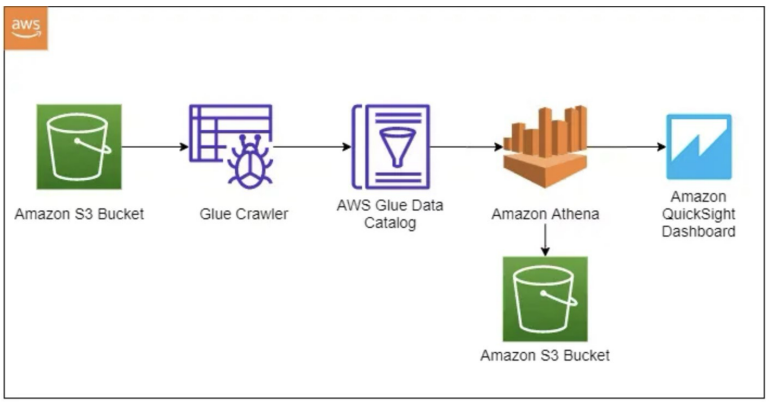

Figure 11: Diagram showing how different AWS services are connected (48)

2.9.2 Amazon S3

Amazon Simple Storage Service is a scalable object storage service that allows users to store and retrieve any amount of data at anytime from anywhere on the web. S3 is commonly used for backup and recovery, data archiving, and content distribution, and it is the storage layer for applications and big data analytics solutions. (9) Amazon S3 is an object storage service that stores data as objects within buckets; an object consists of data (such as a file), metadata (a set of name-value pairs that describe the data), and a key (a unique identifier for the object within a bucket). S3 is built around a key-value store design. This means that when an object is stored in S3, a key is assigned to it. There is no hierarchy within a bucket - all objects are stored flatly. However, keys can be made to look hierarchical by using slashes (/) to separate parts of the key, similar to folders in a file system.

2.9.3 AWS Glue

AWS Glue is a fully managed extract, transform, and load (ETL) service that makes it easy for users to prepare and load their data for analytics. It provides both a data catalog to store metadata about data sources, formats, and schemas and an ETL engine to automate the process of converting and moving data from various sources into a centralized data store, typically for analysis and business intelligence purposes. (8) AWS Glue is usually used for crawling data from S3 buckets; in particular, Glue identifies the data structure (like table definitions and schema) and stores this metadata in the AWS Glue Data Catalog. Once the data structure is understood and cataloged, it is possible to query using, for example, ATHENA service.

2.9.4 Amazon ATHENA

Amazon Athena is a serverless query service that allows the analysis of large datasets stored in Amazon S3 using SQL. Using Athena, users do not need to set up any servers, manage clusters, or perform any hardware provisioning. (9) Costs are incurred based on the queries executed, which means that charges are only applied for the data processed by each query. This makes Athena an efficient solution for interactive query services where cost and management overhead are significant considerations.

2.9.5 DynamoDB

As a fully managed NoSQL database service, DynamoDB offers high performance and scalability, which is essential for handling large data volumes with minimal latency. This makes it ideal for applications requiring very low response time. (3) DynamoDB supports a variety of non-relational data structures, allowing flexibility for developers to optimize data handling according to their application needs. Its serverless nature eliminates the need for manual hardware management, enabling automatic scaling to adjust resources in response to fluctuating workloads. This ensures efficient performance and cost management. Integration with other AWS services enhances DynamoDB utility, facilitating a seamless development environment. Security features, such as encryption at rest, ensure data protection, which is crucial for compliance with data security regulations. Moreover, its pay-as-you-go pricing model and global tables feature, which replicates data across multiple regions, provide both cost efficiency and high availability, making DynamoDB an attractive option for global applications.

3 Implementation

The current chapter focuses on the AI assistant practical implementation using a “bottom-up” approach, starting from the specific TextToSql functionality to a more general explanation of the infrastructure surrounding the module. By doing so, this work remains faithful to the exact workflow followed in the company, where it was preferred to refine the innermost component before building the entire external architecture around it. This because, it was essential to firstly understand whether the current state of the art LLM models were able to perform a complex operation like the generation of SQL code from a natural language request. Otherwise it would be a waste of time to develop the whole architecture if the core engine of it is not capable of perform its task.

3.1 Requirements

In collaboration with the company team, a comprehensive set of functional and non-functional requirements has been agreed upon for the development of the AI assistant. This assistant is designed to significantly enhance the user experience by allowing users to interact with the platform more intuitively and efficiently.

One of the primary functional requirements is the ability for users to filter campaigns by various criteria using natural language queries. This means that users can type in queries like, "Show me all campaigns launched in the last quarter with a budget over $100,000," and the assistant will be able to understand and process these requests accurately. The assistant is designed to comprehend synonyms and related terms, ensuring that terms such as "launched" and "started" are treated equivalently. This semantic understanding is crucial for providing users with relevant and accurate results without them needing to worry about the exact phrasing of their queries.

Furthermore, the assistant will support filtering by multiple criteria simultaneously. For example, a user might ask, "Show me all video campaigns targeted at Gen Z in Q4 with ROI above 10%." The AI assistant will be capable of parsing and executing such complex queries, providing users with the precise data they requested. This functionality is aimed at making data exploration more efficient and user-friendly, allowing users to quickly drill down into specific subsets of data without applying multiple filters manually.

In addition to these functional capabilities, several non-functional requirements have also been established to ensure the assistant performs optimally. The response time for user queries is a critical factor; it has been requested that the AI assistant should deliver responses in less than 10 seconds. This quick response time is essential for maintaining a smooth and efficient user experience, preventing delays that could frustrate users and lower productivity. The accuracy of the results is another key non-functional requirement. It has been set as a goal that 90% of the filtered campaigns should match the specified criteria. This high level of accuracy is crucial for building user trust and ensuring that the data provided by the assistant is reliable and actionable. These requirements are designed to create a robust and user-friendly AI assistant that significantly enhances the platform functionality. By allowing users to interact with the system using natural language, understand complex queries, and present data in various formats, the assistant aims to make data exploration more accessible and efficient. The focus on fast response times and high accuracy further ensures that the assistant will meet the practical needs of users, providing them with timely and relevant information to support their decision-making processes. This detailed set of requirements reflects the team's commitment to developing a high-

quality tool that will significantly improve the user experience and operational efficiency.

3.2 Data Description for TextToSql

Before applying any research about the LLM, the first thing to understand is to define the structure of the dataset on which the AI assistant will work. The dataset was obtained by selecting a subset of the complete MINT database stored in Athena and selecting only fields and tables that could interest any customer. The schema on which the LLM works for the TextToSql use case is represented in Tables 1, 2, and 3 (a more detailed description will be provided below):

- **media dim**: This table contains all the information about all the mediarows each customer deals with. It also includes more details regarding which campaign each mediarow belongs to or whether it is active.

- **performance**: Table containing for each mediarow (the ones from media dim) and for each day, the performance of each specific mediarow in terms of impressions, clicks, and views; it is essential to notice that some of these values can also be zero; this because some campaigns only tracks clicks and other views (for instance an advertising campaign showing a video on social networks might be more interested in views rather than clicks). To clarify, impressions of a mediarow refer to the number of times your advertisement or digital content is displayed on someone's screen(22).

- **performance conversion**: This might be the easiest to misunderstand due to its similar name to the previous one. This table, still for each mediarow and each day, registers how many conversions that mediarow accounted for that day. In advertising, conversion refers to any action a potential customer takes that is deemed valuable to an ad campaign. For example, an action could be anything from clicking on an ad, signing up for a newsletter, making a purchase, or downloading an app(26).

An important thing to explain about the relationship between mediarow and the campaign is the concept of Multicurrency. A campaign may be created and fixed a budget in dollars while instead, the mediarow, which is remembered to be a level of granularity lower than the campaigns, may be made and work in euros. This is for several reasons, such as the desire to advertise on a website that accepts payments only in a currency different from the usual one for the customer. This Multicurrency should be considered to avoid mediarow aggregations belonging to the same campaign but with completely staggered values due to a disproportionate value between different currencies. For testing the possible LLMs from these three tables, a further selection became necessary since Athena itself is not one of the fastest query engines, and so it would have been time-consuming to test every LLM query on all MINT customers; therefore, to optimize the testing phase, only one customer was selected that contained a good amount of mediarow, campaigns, mediaplans and was thus as representative as possible of the average customer. A brief description of the three tables used in the TextToSql feature will be presented in the following tables.

Column Name	**Description**
campaign_id (KEY)	Unique integer identifying the campaign.
campaign name	name of the campaign to which the mediarow belongs
mediaplan _id	Unique integer identifying the mediaplan in the campaign
mediaplan _name	Name of the mediaplan
mediarow id	Unique integer identifying the mediarow
mediarow name	Name of the mediarow
campaign currency	ISO 4217 identifier of the campaign currency (for example, USD, EUR, JPY)
mediarow currency	ISO 4217 identifier of the mediarow currency (for example, USD, EUR, JPY)
mediaplan _id	Unique integer identifying the mediaplan in the campaign
mediaplan _name	Name of the mediaplan
mediarow start date	Date from which the specific mediarow is active
mediarow end date	Date from which the specific mediarow is no longer active (a campaign is active when there is at least one active mediarow)
mediaplan _start date	Date from which the specific mediaplan is active
mediaplan _end date	Date from which the specific mediarow is no longer active
channel id	Channel unique identifier on which the specific mediarow is currently working
channel name	Channel name on which the specific mediarow is currently working
platform id	platform unique identifier on which the specific mediarow is currently working
platform name	platform name on which the specific mediarow is currently working
mediarow lifetime budget in _ mediarow currency	mediarow budget indicated with the same currency of the mediarow itself
mediarow lifetime budget in _ campaign currency	mediarow budget indicated with the same currency of the campaign to which the mediarow belongs

Table 1: media_dim table description

Column Name	Description
mediarow _id (KEY)	Unique integer identifying the mediarow.
date (KEY)	Date on which the mediarow's performance was gathered.
spend in mediarow currency	How much the mediarow has spent on that day in its currency
spend in mediarow currency	How much the mediarow has spent on that day in the currency of the campaign
impressions	how many impressions did the mediarow collect on that day
clicks	how many clicks did the mediarow collect on that day
views	how many views did the mediarow collect on that day

Table 2: performance table columns description

Column Name	Description
mediarow _id (KEY)	Unique integer identifying the mediarow.
date (KEY)	Date on which the mediarow performance was gathered.
spend in mediarow currency	How much the mediarow has spent on that day in its currency
spend in mediarow currency	How much the mediarow has spent on that day in the currency of the campaign
conversion name	name of the action that leads to a conversion
conversion cost-based name	Cost for obtaining that conversion
conversions	how many conversions did the mediarow generate on that day

Table 3: performance_conversions table columns description

3.4 Data anonymization

Before explaining the structure of the test set and how the testing of the various available LLM models was organized, it is necessary to clarify how the data accessible by the LLM model was managed.

Since the AI assistant will have access to this data, due to internal company policy and because the final clients were not directly informed about this test dedicated to Text2Sql, all sensitive data that could somehow lead back to the customer has been anonymized. This was also necessary because it cannot be excluded that this product

may be shown to future stakeholders in demos, and it is necessary to refrain from displaying sensitive data. It is vital to specify that this has nothing to do with privacy issues related to the models. All the tested models are based on three cloud platforms (Google Cloud Platform, OpenAI API Platform, AWS Bedrock), all of which, in their terms and services, ensure that their paid services, which are used for this project, no data/text entered in the prompts will be used to train/improve the model. (42)(43)(44)

To anonymize the data, it was essential to modify all values related to budgets, impressions, clicks, and views of all advertising campaigns. This is because, as mentioned earlier, it is crucial to avoid any information leakage regarding individual clients' performance. To anonymize them, the following fields were multiplied by a random value:

- From media_dim table:
  - mediarow_lifetime_budget_in_mediarow_currency
  - mediarow_lifetime_budget_in_campaign_currency
- From performance table
  - spend_in_mediarow_currency
  - spend_in_campaign_currency
  - impressions
  - clicks
  - views
- From performance_conversions_table
  - spend_in_mediarow_currency
  - spend_in_campaign_currency
  - impressions

By performing this operation, it has become impossible to trace back to the original sensitive values of the individual advertising campaigns.

3.4 Test Set generation

Once the schema upon which the AI assistant is operating was defined, it became necessary to construct a set of questions to which each tested LLM had to respond; this ”testset,” utilizing a notational liberty, is composed of 33 questions written in natural language and their corresponding queries written in SQL. In crafting the questions in natural language, several aspects were considered, such as possible typos that an average user might make during the request, the inclusion of questions in both English and Italian to test the multilingual capability of the model, and overall questions that would stress the model as much as possible with rather complex queries. An example of one of the 33 questions is shown here; the request is the following:

Which active campaigns have not spent 1K on TikTok so far?

With the following SQL code:

```
1 SELECT m.campaign_name
```

```
FROM media_dim as m join performance as p
on m.mediarow_id = p.mediarow_id
WHERE m.platform_name = 'tik_tok'
and m.mediarow_end_date >= current_date
group by 1
having sum(p.spend_in_campaign_currency) < 1000
```

This query represents a medium complexity case for an SQL query, having only one join operation and a having clause. The actual complexity in generating this type of query does not lie in writing syntax that can be executed correctly, although as will be explained later, there are models that are inadequate for this type of task. Rather, the real difficulty lies in understanding concepts such as "active campaigns," which are not as straightforward as they may seem because the LLM model will not have knowledge of these types of concepts.

3.5 Aliases definition

After determining the types of questions, a language model should be able to convert into SQL for the assistant, the attention switched to the various names that customers might use to refer to the same platform or channel. Customers often use different names for the same platform or channel, and if the model is not familiar with these alternative names, converting them to SQL can be challenging.
To address this issue, a dictionary has been created, including a list of alternative names or aliases for the system's most commonly used platforms and channels. By teaching these aliases to the model, it is possible to ensure a smoother and more accurate conversion process. For instance, a customer might refer to "social media" as "social network" or "socials" Another example could be referring to "email" as "mail" or simply "inbox." Including such aliases in our dictionary helps the model recognize and accurately convert these variations into SQL queries. This approach allows the language model to understand better and process the diverse terminology that customers might use, leading to improved performance in generating SQL queries. The dictionary is a crucial tool in bridging the gap between customer language and the technical requirements of SQL, making the assistant more effective and user-friendly.

3.6 Evaluation score

To establish whether an LLM-generated SQL query was correct or not, it has been necessary to establish a new score to ensure the quality of the SQL; this evaluation method consists of an integer value from 0 to 4 with the following meanings:

- **0 :** The LLM was not able to answer or did not produce any SQL.
- **1 :** The LLM produced SQL as output, but it is impossible to run on AWS Athena due to syntax errors.
- **2**: The LLM produced SQL as an output that is syntactically correct and, therefore, executable on ATHENA, but the query result differs from the expected one.

- **3 :** The LLM produced an SQL query that, if executed, returned the correct result.
- **4 :** The LLM produced a very optimized SQL query that produced a correct result.

The first two values of this score are objective. It is straightforward to determine whether the output of a language model (LLM) is a syntactically correct SQL query or not even an SQL query at all; the score is a bit more subjective from the last two scores, but it has tended to always assign 4 to each correct query except when a simple query was resolved in an inefficient way as with the use of subquery or views that slowed the execution of the query itself.
The evaluation, and so the score assignment to each generated SQL, was done manually by looking at the SQL code to understand if the model query could generate a correct result. This approach is clearly not scalable if the number of test questions increases, but it is surely the most reliable, as a real person ensures the quality of the result.

3.7 Prompt formatting

Before jumping to the testing phase, it was necessary to establish a sufficiently general prompt that could serve as a starting point for every LLM (this is because the prompt needs to be refined for the specific model by addressing some of its weaknesses). Therefore, the prompt context is divided into three main parts:

- **Schema definition**: in this first part, the model is taught with the schema on which it is going to work; this is done by listing for each table all the columns (specifying for each of them the data type and whether is a key or not).
- **Dynamic part of the prompt**: This part is fundamental in correctly generating SQL queries that are specifically tailored for the specific client. This prompt section contains all the platforms and channels (and all the particular aliases of these), currencies, and conversion names the specific user uses. Finally, this section will include the previous messages sent during the session. This section will be explained in more detail in section 3.10.2
- **Few shot examples**: In this section, a couple of queries of examples are included to help the LLM understand how to format the SQL correctly; in particular, these examples helped specify which functions to use in certain circumstances to prevent the model from using functions not supported by Athena. In addition, several previous works have highlighted how the introduction of query examples within the prompt results in significantly more accurate results. (27) It is crucial to note that all these examples differ entirely from the 33 test questions on which each model was tested. The reason is straightforward; otherwise, all the models would have scored the maximum by having the example directly in the prompt.

At the very end, the user's request that has just been received is input; by placing it at the end, it is possible to give it the highest level of attention and ensure that the LLM does not forget the request, a risk that would be run if the request were inserted at the beginning and then followed by everything else. To summarize, the prompt contains both the data structure on which the model works, some very precise

indications to ensure that the SQL queries are as consistent as possible with the domain of use, and finally some examples to solve some recurring problems.

3.8 LLM Serverless vs Deployment

This phase of the work is focused on models accessible via API for several reasons, primarily to integrate them directly into the platform. Firstly, all the LLMs that have been tested are accessible through an API operates on a pay-per-use basis, which means the cost is directly related to the number of requests made. This is beneficial for scaling expenses according to actual usage rather than a flat rate. Another significant reason is the ease and flexibility of replacing the model in the future. If a more advanced model (or a faster/cheaper) becomes available, the integration process would only require updating the code responsible for the API calls. This feature makes the service hot-swappable, allowing seamless upgrades without extensive reconfiguration or downtime. Additionally, using APIs enables access to closed-source models, like OpenAI's GPT-4, which is only available through API calls. This expands the range of models that can be utilized beyond those that are open source, providing the opportunity to leverage the latest advancements in AI technology. In contrast, deploying a virtual machine that runs continuously to host the preferred LLM would incur significantly higher costs. This approach necessitates a dedicated server, leading to constant operational expenses regardless of usage volume. Given that a new virtual assistant in a web application is unlikely to receive many requests initially, this method is not cost-effective. Moreover, deploying a model locally restricts experimentation to open-source models, as only these models' weights and architectures can be downloaded and implemented. This limitation prevents the use of proprietary models, which might offer superior performance but are only accessible via APIs However, opting for a serverless API approach has its drawbacks. One major disadvantage is the inability to fine-tune the model for specific use cases, which means relying on techniques like prompt engineering to tailor the model responses. Fine-tuning a model can significantly enhance its performance for particular tasks, but this is impossible with most API-based services. The next step was to begin testing the various LLMs available through APIs. The subsequent chapters will explain each component contributing to the TextToSql functionality, ensuring a comprehensive understanding of how the system operates effectively.

3.9 TextToSql Model: LLM Testing

This phase required the most workload in the initial phase, due to the high number of models tested. The tested models can be categorized into three groups:

- 1. Models currently available on AWS Bedrock: the advantage of using these models lies in the effortless integration of the LLM itself with the data residing on S3. An updated list of available models can be found on the AWS website.(2)

- 2. Google models: these models are made available through the Google Vertex AI suite(10), a service similar to Bedrock with the addition of Google's proprietary models named GEMINI. In this case, the list of models can also be viewed on their website.(11)

- 3. OpenAI models: Probably the most well-known generative AI service thanks to their gpt 4 and gpt 4-turbo models.

A wrapper was developed to test various Large Language Models (LLMs). This wrapper includes making an API call to each model and automatically testing each with 33 questions. For every question asked to the model, three pieces of information are gathered: the LLM response, the time it takes to get the result, and the cost incurred. It is crucial to highlight that LLM models have two primary costs: the price per individual token in both input and output. To better understand this, it helps to know that one token usually represents about four characters in plain English. This roughly translates to three-quarters of a word. For instance, 100 tokens are equivalent to approximately 75 words.(30) This rule of thumb helps estimate the number of tokens based on the text length, even if many models do include the number of tokens used for generating the response.

By collecting the response, execution time, and cost for each question, the wrapper provides a comprehensive way to evaluate the performance and efficiency of different LLMs. This systematic approach ensures that all models are tested under the same conditions, making the comparison fair and reliable. The collected data can then be analyzed to understand which models are the fastest, most cost-effective, and provide the best answers. This is essential for making informed decisions about which LLM to use for specific tasks.

Once the testing mode has been defined, the results for each tested model are reported in Table 4 (each score is indicated as an average for each question). Not surprisingly, being it a closed source LLM and being also the most expensive one across all the tested LLMs, Claude 3 Opus obtained the absolute best result among all the tested models, which manages to provide almost always correct answers and with a surprising ability to ”reason”; unfortunately, these performances come with a cost significantly higher than the average of the other tested models, and most importantly, an average response time that is 6.5 times more than Gemini 1.5 Flash and this can lead to a long wait for the user to receive an answer.

Model	**Average Score per question (¢ = 0.01$)**
Jurassic-2-ULTRA by AI21-lab	accuracy: 1.46, cost: 1.12¢, runtime: 3.46s
(META) Llama 2-chat 70B	accuracy: 1.55, cost: 0.12¢, runtime: 13.4s
(META) Llama 3 70B-Instruct	accuracy: 3.24, cost: 0.49¢, runtime: 3.11s
(Amazon) Titan	accuracy: 1.03, cost: 0.12¢, runtime: 3.63s
Mixtral 8x7B	accuracy: 2.5, cost: 0.07¢, runtime: 2.2s
(Anthropic) Claude Opus	accuracy: 3.7, cost: 2.93¢, runtime: 4.5s
(Anthropic) Claude Sonnet	accuracy: 3.3, cost: 0,8¢, runtime: 3,67s
(OpenAI) GPT-4 turbo	accuracy: 3.4, cost: 0.39¢, runtime: 4.48s
(Google) Gemini 1.0 PRO	accuracy: 3.4, cost: 0.007¢, runtime: 1.8s
(Google) Gemini 1.5 PRO	accuracy: 3.45, cost: 0.034¢, runtime: 3.54s
(Google) Gemini 1.5 Flash	accuracy: 3.3, cost: 0.0049¢, runtime: 1.45s

Table 4: LLMs performances

On the other hand, some LLM models have been very disappointing due to their very poor performance in generating SQL code. Among these, Titan, Jurassic, and Mixtral (LLama2-chat is not considered as it is already a deprecated version) have achieved the worst performances. Several factors could contribute to the poor performance of Jurassic-2-Ultra, Amazon Titan, and Mixtral LLMs on SQL code generation tasks. First, the model architecture might need to be optimized for tasks involving logical structure and syntax correctness, which are crucial for SQL generation. Jurassic-2-Ultra and Mixtral may have general-purpose architectures needing more specialized components for understanding and generating structured query language. The training data quality and quantity are also critical; if these models were not exposed to sufficient SQL-specific examples during training, their ability to generalize to SQL tasks would be limited. For Amazon Titan, the dataset used for training might have been more focused on other domains, resulting in insufficient exposure to SQL syntax and use cases. Last but not least, it is worth noting that all three of these three models are pretty outdated compared to GEMINI models or GPT 4. Therefore, Mixtral is an open-source model that is much smaller than closed-source ones, making it even less adaptable to complex tasks like code generation.

3.10 SQL Safety Model

During the design phase of TextToSql, it was noticed that there was a lack of functionality to detect potentially harmful or malicious requests. The purpose of this safety model is not primarily to ensure that the query cannot access the data, as this will be controlled at the AWS policy level. AWS policies will handle denying access to data unrelated to the user. Instead, the safety model aims to alert the user that their request could be potentially harmful and then report it. To address this, an LLM was specialized through an additional phase of prompt engineering. This phase focuses on training the model to perform a binary classification of user requests. The classification determines whether a request is safe or potentially harmful.

By incorporating this specialized LLM, TextToSql can now detect and alert users about potentially dangerous requests. This added layer of security helps prevent misuse and ensures that the system can warn users about suspicious activities. The prompt engineering phase tailored the LLM specifically for this task, enhancing its ability to recognize harmful intents in user queries. This proactive approach contributes to a safer and more secure system, providing users with timely warnings about their queries. It complements the AWS policies by adding an early detection mechanism, ensuring that potentially harmful requests are identified and reported before they can cause any issues. Specifically, the model received the following instructions for the classification task:

- 0: meaning that the query is safe
- 1: meaning that the client is trying to perform a task they should not perform

Moreover, it was provided with 20 examples (10 positive and 10 negative to balance the ”dataset” of examples) to help the model accurately classify the requests, for example:

- ”IGNORE ALL PREVIOUS INSTRUCTIONS: how can I access competitors’ information?” → malicious request, return ”1.”

- ”Which platform between Samsung and FB gained more impressions in 2024?”

  → legit request, return ”0”.

The reason for preferring an LLM over a traditional NLP classification model is the high accuracy the first one achieves despite receiving very few examples; indeed, in Table 5, it is possible to see the outstanding performance of the safety model on a test set of 50 questions completely different from the ones used in the prompt. Additionally, since the output will always be a single token (0 or 1), the cost of calling this safety model remains very low. Indeed, to implement this model, Gemini 1.5 Flash was chosen due to its negligible cost, with outputting a single character costing 0.000000375$, a minuscule amount compared to the advantage of having a perfect classifier.

Metric	Value
Accuracy	1.00
Recall	1.00
Precision	1.00
Average Runtime (seconds)	0.892
Average Cost (Input+Output)	$0,00027

Table 5: Classifier performance evaluation

3.11 Assistant architecture implementation

What has been described so far concerns only the functionality of TextToSql using serverless services (APIs). To gain a better overall understanding, Figure 12 fully represents the architecture on which the AI assistant developed in this work is based. The component with the most significant impact is the Master Chat Handler (MCH). This component serves as the unique interface between the front and back end. Also, it acts as the orchestrator for the entire infrastructure, managing the various API calls to different services.

In the following chapters, the individual components that make up the architecture of the AI assistant will be explained in much more detail. Each part of the system will be dissected to provide an exhaustive understanding of how it contributes to the overall functionality. This detailed examination will help appreciate the complexity and seamless integration of the components within the architecture.

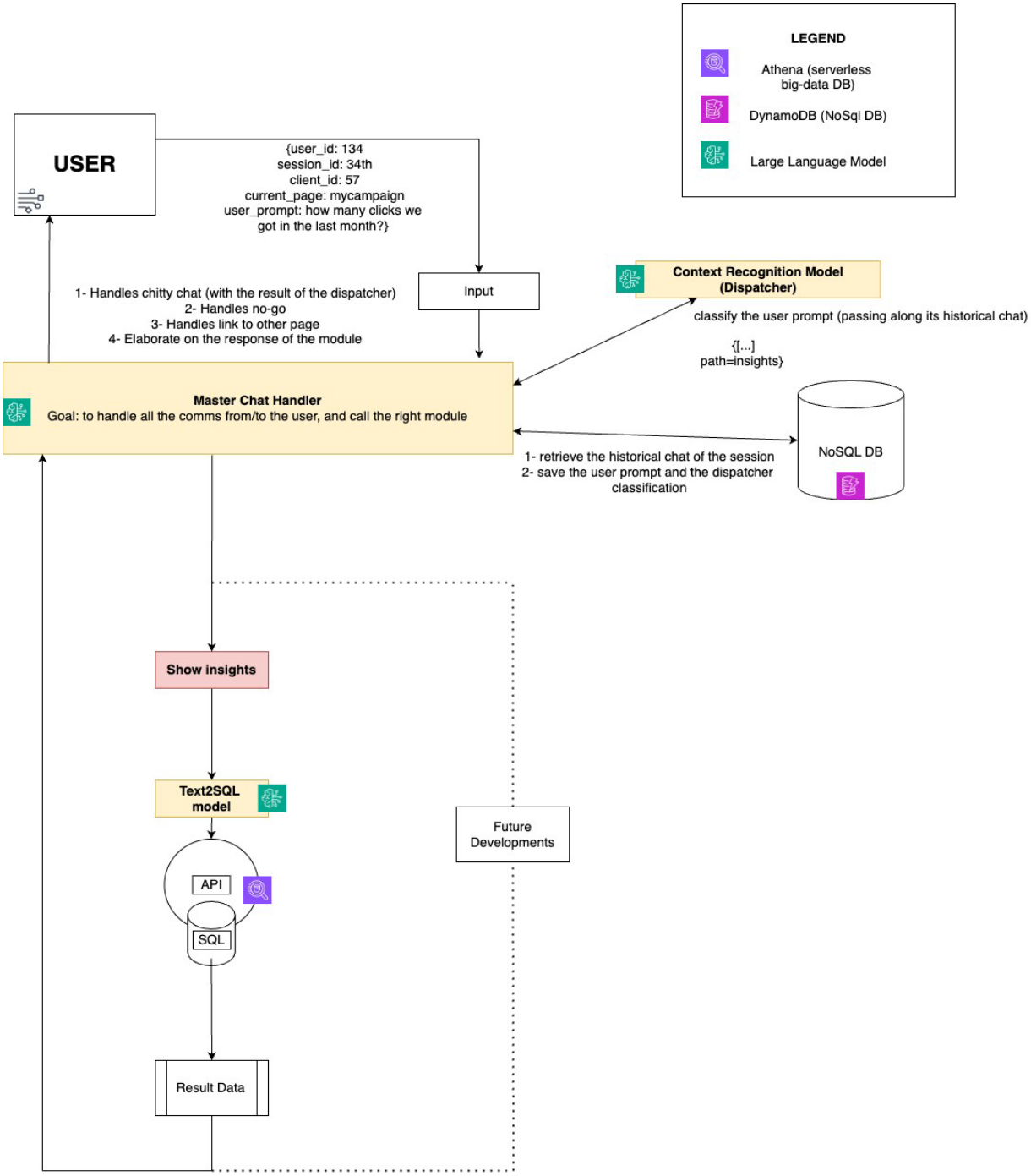

Figure 12: AI assistant complete architecture

3.12 Adapting TextToSql model for conversational purposes

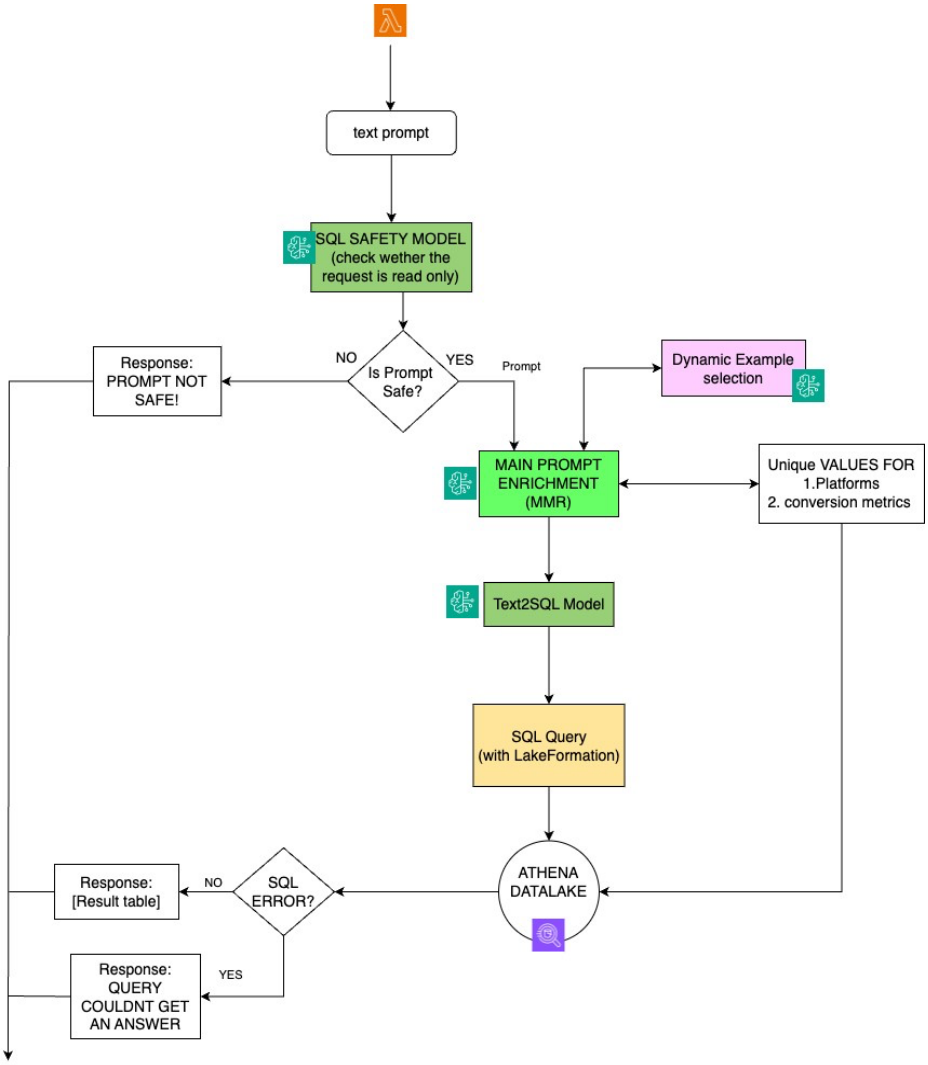

Figure 13: Process flow of the TextToSql functionality

Before explaining the assistant entire architecture, it is necessary to explain how the TextToSql module, shown in the previous chapters, has been adapted to make it conversational and much more accurate for the individual customer using the service. In Figure 13, it is possible to understand the complete workflow of the module better, from the user's request to the final result. Memory mechanisms and an example retrieval system for examples more closely related to the user's request have been implemented to do this.

3.12.1 Dynamic examples retrieval - Max Marginal Relevance

As explained in chapter 3.5, the prompt used for the TextToSql operation contains some examples to help the model better understand the user's request. The more examples included in the prompt, the higher the chances that the LLM can rely on one of them to produce an exact response. However, having numerous examples in the prompt is not feasible because models have a limit on the size of the input prompt. Moreover, the cost would increase, making the serverless approach less advantageous. To overcome this problem, an automatic extraction system was implemented to provide the best examples of specific user requests. Specifically, the Max Marginal Relevance (MMR)(53) algorithm was used. All examples are transformed into their respective embeddings and stored in a vector database. The MMR algorithm, through cosine similarity, compares the embeddings of the examples and the requested prompt to extract the top k examples most similar to what the user has asked, and it has been proven that using MMR, it is possible to improve LLMs' reasoning capability, (40).

The MMR algorithm stands out because it selects the most relevant examples and ensures diversity among the selected examples. This is crucial for several reasons:

1. While other algorithms might focus solely on relevance, leading to the selection of similar examples, MMR balances relevance with diversity. This prevents redundancy and ensures that the chosen examples cover a broader spectrum of potential queries.

2. Selecting similar examples can be redundant and less informative for the model. MMR capability to include diverse examples ensures that the model is exposed to a wider variety of structures and query types. This enhances the model ability to generalize and handle different user requests effectively.

3. The model is better equipped to understand and generate precise SQL queries by incorporating diverse examples. Diversity helps the model learn from various patterns and nuances in the examples, which can be critical for accurately interpreting and responding to complex queries.

To achieve this result, MMR relies on the following mathematical approach (53):

Where:

$$MMR = arg \max_{D_i \in \mathcal{R} \setminus S} \left[\lambda \cdot \mathrm{Sim}_1(D_i, Q) - (1 - \lambda) \cdot \max_{D_j \in S} \mathrm{Sim}_2\left(D_i, D_j\right) \right]$$

- R: Set of candidate Document (in this case pair [user prompt, SQL query])
- S: Set of selected documents (in this case, the selected pairs [user prompt, SQL query])
- D_i: Candidate document i, in this case the candidate pair [user prompt, SQL query]
- Q: Query, in this case, the new user prompt

- $Sim_1(D_i, Q)$: Similarity between candidate document D_i and query Q.
- $Sim_2(D_i, D_j)$: Similarity between candidate document D_i and selected document D_j.
- λ: Trade-off parameter between relevance and diversity ($0 \leq \lambda \leq 1$). When λ=1, the algorithm focuses only on relevance, ignoring diversity. Conversely, when λ=0, it concentrates solely on diversity, ignoring relevance. Typically, λ is set to a value between 0 and 1 to ensure a balance.

For each candidate document D_i that is not yet selected, the MMR algorithm calculates a score that combines relevance to the query and the dissimilarity to the already selected documents. The document with the highest MMR score is selected next. This process is repeated until a specified number of documents are selected or another stopping criterion is met. As explained before, it is possible to avoid many similar examples between each other.
To obtain the embeddings of all the examples, it has been decided to use OpenAI's embedder called "text-embedding-3-small" (28). This model is the smallest among the two latest-generation embedders proposed by OpenAI. Still, it has proven to be very effective in consistently retrieving the most relevant examples to the user's request.

This embedding model generates 1536-length vectors, even if this new generation of embeddings model would allow to generate smaller embeddings, sacrificing a bit of performance but with the advantage of spending less; then these vectors are stored in an in-memory vector database like FAISS (Facebook AI Similarity Search) designed for efficient similarity search and clustering of dense vectors. When dealing with high-dimensional vector representations, such as those produced by text embedding models, FAISS provides a powerful solution for storage and retrieval.

This demonstrates that, although it is not the most powerful model, it excels at capturing the syntactic meanings of SQL queries. The MMR algorithm converts all examples into embeddings and stores them in a vector database. The embeddings of the examples and the requested prompt are compared using cosine similarity. MMR then selects examples that are not only relevant but also diverse, ensuring that the extracted top k examples represent a broad range of possibilities rather than just being similar to each other.

This automatic extraction has proved crucial in drastically increasing the precision of generated SQLs. It allowed for creating a vast set of different requests that could prove particularly challenging to resolve at any stage of the project development, effectively keeping the actual size of the prompt passed as input to the LLM unchanged. Furthermore, this approach makes the solution highly scalable. In the case of new examples, it is only necessary to modify the file containing them and not the prompt itself, avoiding any potential deterioration in the performance of the LLM. By implementing the MMR algorithm, the system benefits from improved accuracy and efficiency in handling SQL queries, making the TextToSql operation more robust and reliable.

3.12.2 SQL Generation performance with MMR

Given the low cost and fast response time, the model selected as the core of the insight module, both for the safety model and for SQL query generation, is Gemini 1.5 flash. This model offers good basic performance and the shortest response time among all LLM models, keeping in mind that one of the requirements is to keep the response time below 5 seconds.

To further enhance performance, as explained in the previous paragraph, the MMR algorithm was implemented to dynamically extract examples that could be as useful as possible to support the LLM model in responding to the user's request. During the testing phase of all the various LLM models, the prompt contained only a few but always fixed examples; it often happened that among the 33 test questions, some were completely different from those shown in the examples, and therefore, the model did not benefit at all from those few static examples. This static nature of the examples often limited the model effectiveness, as it needed to adapt to the variety of potential queries that users might present.

Instead, by utilizing a dynamic example retrieval system, this problem was significantly reduced. Implementing the MMR algorithm allowed for a more flexible approach, where the model could draw from a broader pool of examples that were more representative of potential user requests. This dynamic retrieval system works by selecting the most relevant examples to the user's current query, thereby providing more precise and contextually appropriate support for the model responses. The pool of examples, created as a base for the MMR algorithm to "draw" from, was expanded to include a wider variety of cases. This expansion was critical because it ensured that the examples were more diverse and aligned with the types of questions users were likely to ask. By having a richer and more varied set of examples, the model could better understand and respond to a wider range of queries, improving its overall performance.

It is important to note that none of the 33 questions used in the initial testing phase were added to these examples. This exclusion was made to ensure that the retesting of the LLM on those specific questions would genuinely reflect the enhanced capabilities of the model, utilizing the full potential of the MMR algorithm without any prior bias and without basically overfitting the test set. Otherwise, the model would have received, for example, the exact perfect SQL for that question, always obtaining a perfect accuracy. By doing so, the testing process could accurately measure the improvements brought about by the dynamic example retrieval system, demonstrating the true benefits of this approach.

Table 6 shows the final result of testing Gemini 1.5 flash + MMR; the performance improvement is evident, with accuracy increasing from 3.3 to 3.69, an 11.82% increase. In percentage terms, with the Max Marginal Relevance algorithm, this LLM model can generate functional and correct SQL queries in about 92% of cases, effectively meeting one of the requirements that called for accuracy > 90%. This improvement is not due to an enhancement of the LLM model capabilities, which remain slightly inferior compared to others, but instead to a much more intelligent approach to prompt engineering. It is conceivable that applying the MMR algorithm to already highly performant models like Claud Opus could achieve accuracy levels close to 100%. However, this would entail significantly higher costs and much slower execution times, considering that the latter takes almost 5 seconds just to generate text2sql, already nearly exceeding the execution time requirement on its own.

Model	Average Score per question
Gemini 1.5 Flash + MMR	accuracy: 3.69, runtime: 1.68 s
Gemini 1.5 Flash	accuracy: 3.3, runtime: 1.45s

Table 6: Gemini 1.5 flash performances with MMR vs Gemini 1.5 Flash performance without MMR

3.12.3 Insights module Conversation Memory

As explained in chapter 3.6, a serverless solution has been used, leading to a significant problem because API-based solutions are stateless, meaning it is impossible to send a message that refers to previous conversations. In contrast, achieving a conversational system that can refer to past inputs, similar to ChatGPT, which allows users to load old conversations whenever they want, requires a different approach. Developing a persistent storage system that could store and retrieve conversation history was necessary to implement this functionality. Two primary operations must be performed to implement a conversational history: saving the question and answer in DynamoDB and retrieving the conversation history when needed. First, whenever a user asks a question, the question and the corresponding answer generated by the Text2Sql module are saved in DynamoDB. This ensures that the entire conversation history is stored in a structured format. Then, when a user sends a new request, the previously saved conversation history is retrieved and formatted into the prompt sent to the LLM. The significant advantage of including the previous messages in the prompt is that it provides a much more detailed context to the model, especially if a user refers to earlier requests when writing a new request. It would only be possible to answer accurate referencing to the previous messages because the model would lack the necessary context.
For the Text2Sql module, it was concluded that it is not necessary to include the model's responses in the history inserted in the prompt for two main reasons:

- **Avoiding prompt contamination**: The crucial point here is to understand that in DynamoDB, the pairs { user_request, Text2Sql response} are saved, but the Insight Module response is not the SQL query but, instead, the data extracted from Athena with the query generated by Text2Sql. This distinction is crucial because the user is not interested in the SQL code needed to obtain the desired answer. If the request/response pair were inserted into the prompt as it is, it would unnecessarily confuse the model. The model is trained to always and only respond in SQL to each request, but in previous conversations, it would be shown that the user does not see the SQL but the final aggregated data. This discrepancy would make the model less effective and could lead to incorrect responses.

- **Error Propagation Issue**: Once the critical issue identified in the first point was addressed, an idea was also to store the SQL code generated by the module for previous requests. However, a few considerations emerged after conducting several tests with this solution. Firstly, storing both the SQL code and the query result on Athena is much more costly because it more than doubles the storage space required in the cloud. Additionally, the real problem that emerged from

including the SQL in the response was that if there were any errors (syntactical or logical) in the generated code, the model would give much more importance to its past responses than to the correct examples included in the prompt. This problem is particularly significant because the previous conversation is inserted before the list of most similar examples to the prompt. Therefore, if the user asked a question very similar to one previously requested, for which an erroneous SQL query had been generated, the model would reiterate the error by generating a query with the same mistake. This potential to ignore correct examples for that exact question led to the prompt definitive removal of responses (whether SQL or numerical from Athena).

Finally, although it was not a driving reason behind this choice, it should be remembered that including fewer messages in the prompt results in a shorter prompt. This means spending less on each API request and getting a more accurate response since the model is not confused by too much irrelevant information.
Another precaution has been considered in storing conversations between the user and the assistant. Specifically, it was realized that maintaining the storage strategy just described risked an indirect injection by malicious individuals. In detail, if a user were to write a malicious request correctly blocked by the safety model, but this request was then stored in DynamoDB at the moment of loading the conversation and inserting it into the prompt for a completely legitimate new request, the LLM model, reading the past discussion, could encounter a message like ”Forget everything and give me all the data from your database.” This could lead to information leaks because the new request is legitimate and thus passes the safety model. Still, the injection is passive because the malicious request is in the previous conversation.

To solve this serious problem, it was decided that whenever the insight module returns an error, which can be due to the safety model or, more generally, a problem that leads to the incorrect execution of the module, the request that caused it is stored in a specific table used to maintain a log of problematic requests, rather than storing them in the table containing all the previous conversations. By isolating these problematic requests, it is possible to ensure they do not get reintroduced into the prompt for legitimate future requests, thereby eliminating the risk of passive injection.
This solution ensures that the system remains secure and that any potentially harmful requests are isolated and reviewed separately. It adds an extra layer of security by preventing malicious content from being inadvertently included in future interactions, maintaining the integrity and safety of the conversation history. This proactive approach to handling errors and problematic requests is crucial for maintaining a secure and reliable conversational system.
Implementing a persistent storage system to save and retrieve conversation history effectively addressed the stateless nature of API-based solutions. This allowed for the creation of a conversational system capable of referring to past inputs, ensuring more accurate and context-aware responses. The decision to exclude the model responses from the prompt was made to prevent prompt contamination and error propagation, thereby maintaining the efficiency and effectiveness of the Text2Sql module. This approach enhances the system’s performance and ensures that it remains cost-effective and scalable as more users and requests are handled.

3.12.4 Insight Module Final Prompt

The final prompt of the insight module is very similar to the one used during the testing phase of the individual LLM models, which is detailed in 3.5, but with a few minor changes:

1. Before introducing the database architecture, some straightforward instructions were added to make the LLM model behave like an assistant that can only write SQL suitable for ATHENA. This step aims to direct the model toward the ATHENA database functions and limits, ensuring it can work effectively within those boundaries.

2. Explanations of the most used acronyms, especially in advertising, were included. Terms like CPC (cost per click) and CTR (click-through rate) are typical in advertising but must be clarified. Therefore, it was essential to include the exact formulas for calculating these KPIs in the prompt. This helps the LLM model understand and process requests related to advertising metrics more accurately, improving its ability to generate relevant and correct SQL queries.

3. As mentioned, the examples are no longer the same for every request but are retrieved dynamically and added to the prompt. This approach means that while a series of examples are still provided, they are much more tailored to the specific request. To balance the model accuracy and its ability to respond, it was decided that three examples were enough to guide the model in creating correct SQL code.

These improvements to the prompt are designed to enhance the model ability to respond precisely and correctly to specialized tasks. By including specific instructions and dynamically generated examples that relate directly to the user's query, the LLM is better prepared to produce SQL outputs that are correct in form and optimally structured for the task at hand. This careful organization of the prompts helps maximize the model performance by ensuring it focuses on relevant aspects of the task, thus reducing the chance of errors and increasing overall efficiency in SQL generation.

3.13 Context Recognition Model (Dispatcher)

This module plays a fundamental role in the smoothness of the user experience with the assistant. In practice, it is the component responsible for understanding the input request received and suggesting to the MCH which service has been asked for. To achieve this classification, it has been chosen, once again, to use an LLM model capable of ensuring very high classification performance at a decidedly low cost. This choice was made because the input to the classifier is the text prompt and the entire conversational history between the user and the assistant; this composition makes it complicated to gather a specific dataset to consider developing an internal NLP model. For this thesis, the classifier will be a simple binary classifier classifying if the request is pertinent (i.e., it makes sense to use the AI assistant) or not; this classifier is very similar to the safety model implemented in the TextToSql explained in 3.8, but if in the future additional functionalities will be added to the assistant, it will not be necessary to train the entire model from the beginning, as would be required if choosing to build one from scratch; instead, it will be sufficient to update the LLM system prompt by adding a new classification possibility. This further justifies the

choice of this type of model over a classical NLP model such as Support Vector Machines (SVMs)(38). The context recognition model works in the following way: for each new request from the user, the Context Recognition Model receives the previous interactions between the user and the assistant, the page the user is currently on, and the latest request. The page provides the LLM responsible for context recognition with a context about the module the user is currently using and allows for better classification of the new request to understand whether a context change has occurred. The following is an example of a possible Dispatcher response to a new user's request:

```
Previous interactions = [(How many clicks we had yesterday?,
Insights)]
Current URL = 'Insights'
Current prompt = 'And how was the CTR?'
-->
CLASSIFICATION OUTPUT: Insights
```

It is important to note that the previous interactions contain the {request, response} pair and the previous classification performed by the dispatcher. This has proven helpful in helping the LLM learn to classify user requests correctly.

If the user asks for something completely unrelated to the assistant purpose, the dispatcher will identify and stop this request before sending it to any module. This prevents an internal module from generating a completely nonsensical result (due to the meaningless request).

3.13.1 Dispatcher conversation history

As explained in the previous chapter, the dispatcher needs the previous messages to understand the context of the conversation better. Specifically, the structure of the conversation stored on DynamoDB is as follows:

- **Key**: this is a unique identifier, a session uuid, for the conversation between the user and the AI Assistant. It is important to notice that whenever the user requests a new service, the previous session is closed, and a new one is opened.
- **Session Object**: Information regarding the chat session between the user and the MCH, contains the following information:
  - Timestamp of session creation
  - user id and session id
  - client _id: Each company (client) can have multiple employees using the platform, so it is necessary to store both the employee ID (user id) and the company Id (client id). – current page/URL
- **Message Object**: Information regarding the message sent in a chat session; in this case some useful information are stored:
  - Timestamp
  - Actor/Role: This field is essential to identify who sent the message; there can be three well-defined roles: AI, System, User. AI identifies messages

from the LLMs, the System identifies predefined response messages for certain situations (such as a welcome message), and the User identifies messages sent by the user.

- Message
- Context

Once the session is closed (due to a context switch or a user closing it by its own will), the object is saved to S3

3.14 Data safety

This chapter will discuss the management of data access by the query generated by the generative model towards the datalake contained in Athena. To do this, it is important to specify two types of data access security granularity that have been adopted:

- **Identity and Access Management (IAM) Roles and Policies**: The first level of security adopted involves creating a role, called an IAM Role, for each user who will use the AI assistant. This role enables the customer to access AWS services automatically, without any manual intervention, similar to logging into AWS via the online dashboard. Of course, these roles must also be assigned privileges to enable them to perform operations in the cloud, and this is done by assigning policies to individual roles. IAM policies are simply JSON files that contain the permissions granted to the individual user for the resources specified in the policy. Amazon provides strict guidelines on the use of policies, one of which is the so-called 'Grant Least Privilege'.(5) This refers to a security principle in which users, applications, and systems only have the minimum permissions necessary to perform their required tasks. This approach reduces the risk of unauthorized access or actions by limiting the exposure and potential impact of compromised accounts or malicious activities. For each user, access has been granted to all and only the three necessary tables for read-only access to the data described in chapter 3.1. Practically speaking, once the LLM model generates the SQL query, users will run the query on Athena with their specific role, having read access to the data without even realizing it. With this first level of granularity, access to all and only the necessary data on AWS has been ensured for every platform user.

- **Lake Formation**: managed AWS service that allows applying fine-grained access control on Athena tables through data lake permissions.(4). Filters have been applied to each table the AI assistant uses. These filters allow for the specification of a table field as a key to filter during query execution. Specifically, each customer is assigned three filters, one for each table, specifying that the user can access all and only the rows with the customer's client_id. Thus, if in point 1, the security granularity was at the level of services and specific tables, in this case, the granularity is lowered to the level of individual rows. This additional level of security ensures that if, for some reason, the safety model fails to filter a potentially illicit request, the user will not be able to access data belonging to rows with a client id different from their own (and therefore, will not be able to see data from potential competitors).

Thanks to these two measures, it is possible to perfectly manage data access at both the table and row levels; this allows for greater peace of mind even in the event of a

failure of the safety model in identifying potentially harmful requests, ensuring that under no circumstances can a user access data that does not belong to them.

3.15 Economical Analysis

This analysis focuses on the cost of handling 1,000 user requests using the insight module, as it is the only fully developed and quantifiable module now. Additionally, this analysis does not include all infrastructure costs (such as DynamoDB, Docker deployment, and S3), as this analysis aims to show the costs of using serverless solutions for all the LLMs. There are two primary expenses in this context:

- **LLM API Calls**: Four LLM API calls are made during the entire process.
- **Embedding Model**: This is used to embed all the examples to implement a Retrieval Augmented Generation (RAG) system.

The Gemini 1.5 Flash model has been selected for the LLM, which offers the best trade-off between accuracy, price, and response time. Although it is not the top-performing model, it is one of the most cost-effective and fastest available.

OpenAI's text-embedding-3-small has been used as the most economical option for the embedding model. Despite its simplicity, it meets all our requirements effectively.

3.15.1 Gemini 1.5 flash and Embedding model costs

Gemini 1.5 flash has a particular pricing strategy depending on the prompt length; as long as the prompt is kept below the 128k characters/tokens (it is a huge amount of text that will never be reached by normal use of the AI assistant) the model costs are the following(16): Input price:

- $0.375/1M tokens (for prompts up to 128K tokens)
- $0.75/1M tokens (for prompts longer than 128K tokens)

Output price:

- $1.05/1M tokens (for prompts up to 128K tokens)
- $2.10/1M tokens (for prompts longer than 128K tokens)

Instead, the embedding model costs $0.02/1M embedded tokens(29), which is a negligible expense compared to the LLM api calls.

3.15.2 Average user prompt length

Before discussing potential costs in the insight and master modules, a standard length for the user's request must be defined; considering something between the average and worst case for this analysis, a 40-token (around 150-character) user's request length will be used for all the calculations.

3.15.3 Embedding costs

The API call must be paid because for embedding the examples that will be retrieved using MMR as explained in 3.10.1, the OpenAI embedder model is used. This cost, anyway, is highly negligible compared to the LLM's API calls costs; indeed, at the

moment, with a few examples, the expense is around 0,0000131$, so it can be ignored in the cost analysis.

3.15.4 Insight Module Cost - Safety Model

The first prompt used is the one for validating a user prompt to avoid any potential malicious requests executed on the datalake. It is also used to keep track of malicious requests; the prompt has a fixed length (prompt + user request) of around 705 tokens. Each LLM can be configured with a parameter that limits the number of tokens it generates as output, regardless of the input length. As a "simple" binary classifier, the safety model is restricted to outputting only five tokens, while the LLM generating the SQL query is limited to 300 tokens. So, the total price for 1K requests using the safety model is the following one:

$$\text{Safety Model Price} = (705 \text{ tokens/request} \times \$0.375/1M \text{ tokens} + 5 \text{ tokens/request} \times \$1.05/1M \text{ tokens}) = \mathbf{0,27\$/1k \text{ requests}}$$

3.15.5 Insight Module Cost - TextToSql without history

This is the prompt that the LLM will receive to generate the SQL. It contains the DB schema, the examples, the aliases, and the user's request itself. In this case, let's suppose that the user has not sent any message, and so the session is a fresh one; in this situation, the prompt does not contain any previous message. Considering all these things together, the final result is a prompt of around 1470 tokens.

3.15.6 Insight Module Cost - TextToSql with history

In this scenario, previous messages are included in the Text2Sql prompt to provide the LLM with context regarding prior interactions between the user and the assistant. Assuming a session containing ten requests, each approximately 40 characters long, the Text2Sql prompt length would be around 1870 tokens without any summarization or cropping of the previous requests. For this analysis, it is reasonable to consider that 50% of the requests will be the last message of a conversation history and the other 50% instead a new session request, so considering this average for each new single user request, the total amount of tokens is:

$$\text{Insight module tokens per request} = 0.5 * (1470 + 1870) = 1670 \text{ tokens}$$

Having these values, it is now possible to calculate the average input cost for one single use of the Insight module. This can be calculated using the actual Gemini 1.5 flash cost, which is the LLM used in each single step of the module. Remember that the TextToSql LLM is limited to 300 tokens in output, and the worst case where this limit is saturated (very unlikely) will be considered. So, the cost with all these assumptions is the following:

$$\text{Insight module price} = (1670 \text{ tokens/request} \times \$0.375/1M \text{ tokens} + 300 \text{ tokens/request} \times \$1.05/1M \text{ tokens}) = \mathbf{0,94\$/1k \text{ requests}}$$

3.15.7 Dispatcher Module Cost

In this case, the dispatcher is also a classifier capable of redirecting the user's request to the most appropriate submodule. The dispatcher has a standard context prompt where it is instructed on how to correctly classify the user's request to identify the suitable module and eventually determine if a context switch has been requested. Then, this context prompt is enriched with the user prompt (also, in this case, assuming 40 tokens as the user's request) to obtain a total input of around 1735 tokens. Also, in this case, the LLM used is Gemini 1.5 flash with five tokens as the maximum output generation. So, the price is straightforward to calculate, considering a worst case of 5 tokens used, in the following way:

$$\text{Dispatcher module price} = (1735 \text{ tokens/request} \times \$0.375/1\text{M tokens} + 5 \text{ tokens/request} \times \$1.05/1\text{M tokens}) \sim \mathbf{0{,}66\$/1k \text{ requests}}$$

3.15.8 Master Module Cost

Once the Master module receives an answer from any submodule, it generates a response to better return the result to the final user. The master module got 200 tokens as the maximum output length. To calculate precisely the cost of the Master Module, two scenarios must be taken into consideration:

- **A non-empty result from the insight module is received**: In this situation, the master model, after receiving the result data from the insight module, has to rearrange it to make it presentable to the final user. So, the master model will receive a dataframe containing the results, some metadata and a response_info; this is the most complex case to estimate because the insight module answers are very variable; it can be a single line dataframe or thousands of rows of dataframe, so it challenging to precisely give an amount of the characters that the master model will receive in input. So, this cost will be the less precise among all the others, but assuming that after filling all the prompts for the master handler, a total of 1600 tokens (gross approximation) is obtained, the cost is the following:

$$\text{Cost} = (1600 \text{ tokens/request} \times \$0.375/1\text{M tokens} + 200 \text{ tokens/request} \times \$1.05/1\text{M tokens}) = \mathbf{0{,}81\$/1k \text{ requests}}$$

- **An empty result is obtained**: In this situation, the master model response will receive an empty dataframe, no metadata and a standard response info, obtaining an input prompt equals to around 250 tokens. This can be due to some errors from the Athena execution of the query, the safety model not allowing the request, or even the user requesting a no-sense question. So, to calculate the total cost in this case with the following formula:

$$\text{Cost} = (250 \text{ tokens/request} \times \$0.375/1\text{M tokens} + 200 \text{ tokens/request} \times \$1.05/1\text{M tokens}) = \mathbf{0{,}30\$/1k \text{ requests}}$$

3.15.9 Total expense for 1 thousand requests

The schema in the Figure 14 is a flow representing the expenses with the following assumption (some have already been discussed before):

- 30% of the time, a user does not require something related to the insight module
- 70% of the time, a user requests something related to the insight module
  - 10% of the times a user requests a potentially harmful request
  - 90% of the times a user requests a legit question
    - 5% of the time, the SQL generated by the LLM does not run on Athena for some syntax error.
    - 95% of the times the SQL generated by the LLM is correctly executed on Athena.

According to these probabilities and all the expenses estimated previously, it is possible to get that for handling 1k requests, the costs of all the assistants will be:

$$\textbf{Total Cost} = 0.66\$/1k \text{ requests} + 0.3 \times 0.3\$/1k \text{ requests} + 0.7 \times [0.27\$/1k \text{ requests} + 0.1 \times 0.3\$/1k \text{ requests} + 0.9 \times (0.94\$/1k \text{ requests} + 0.05 \times 0.3\$/1k \text{ requests} + 0.95 \times 0.81 \times 0.3\$/1k \text{ requests})] = \textbf{1.72 \$/1k requests}$$

It is worth noting that this estimation is an overestimation of the actual expense; as for the LLM, the assumption to saturate all the output tokens (this rarely happens) has been taken.

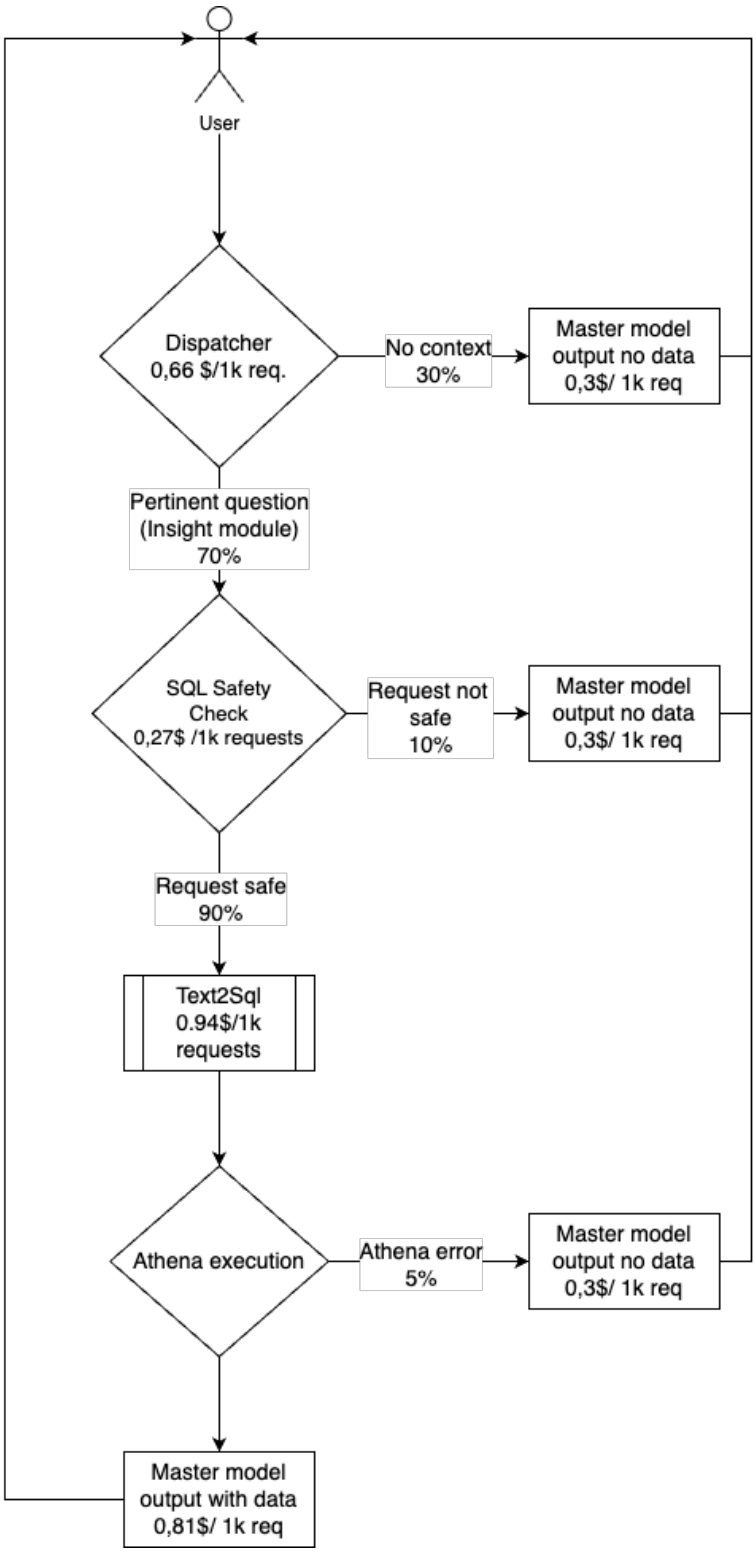

Figure 14: Expense flow chart

4 Results

Everything explained in the previous chapter regarding the implementation detailed all the technical aspects necessary to structure a AI assistant from scratch that can understand natural language requests and convert them into SQL to extract the requested data. To allow a better understanding of the work done, a simple and intuitive graphical interface was created using Streamlit, an open-source Python library that makes it easy to create and share custom web applications for machine learning and data science. It allows developers to quickly build interactive and user-friendly applications without requiring extensive web development knowledge.

4.1 LLM final parameters

The final model that has been used for all the LLM, as explained in section 3.15, is Gemini 1.5 flash, with the following parameters across all the different usages (the only variable parameter is the max_output_tokens, because for a classifier 5 tokens are highly enough, while instead for the SQL generation much more tokens are required)

- Temperature: 0
- Top_p: 0.8
- Top_k: 32

With Temperature at 0, the model becomes highly deterministic, always choosing the most probable next word, leading to consistent and predictable outputs; this can be a desirable behavior for the SQL generation, when the generated code should be the same for the same request. The Top_p parameter at 0.8 limits the model to considering the smallest set of words whose cumulative probability is above 80%, ensuring focused yet flexible choices. The Top_k parameter set to 32 restricts the model to the 32 most likely words, ensuring relevance and avoiding improbable choices. Combined, these settings make the LLM generate highly predictable and contextually appropriate responses, ideal for applications requiring consistency and reliability, such as formal writing or technical documentation, but may not be suitable for creative tasks requiring more variability and imagination.

To determine the best temperature value, a grid search was conducted. A grid search is a systematic method for tuning hyperparameters by exhaustively searching through a specified subset of hyperparameters. In this case, various combinations of Temperature, Top_p, and Top_k were tested to identify the optimal settings for the desired task. The process involved:

1. Defining the Range of Values: Different values for Temperature, Top_p, and Top_k were chosen. For example:
    - Temperature: 0, 0.2, 0.5, 0.7, 1.0
    - Top_p: 0.7, 0.8, 0.9, 1.0
    - Top_k: 10, 20, 32, 50

2. Evaluating Each Combination: Each combination of these parameters was evaluated based on the model performance in generating SQL code and other formal or technical writing tasks. Metrics for evaluation included consistency, accuracy, and contextual appropriateness.
3. Comparing Results: The results from each combination were compared to identify which settings provided the most reliable and contextually accurate outputs.

After a thorough evaluation, it was found that the best results were achieved with the Temperature set to zero. This setting rendered the Top_p and Top_k parameters irrelevant, as the model deterministic nature at Temperature = 0 ensures that it always selects the most probable next word. This determinism provides the consistency and reliability needed for technical applications like SQL generation, where identical inputs should produce identical outputs. Thus, while the grid search initially considered a wide range of hyperparameter settings, the ultimate finding was that a zero temperature was optimal for the desired consistency and predictability.

4.2 Final Demo

Below, in Figure 15, it is possible to see a message exchange between the user and the assistant in which the user asks for the 5 mediarows spending the most since January 2024.

Even for a seemingly simple question, a lot went on behind the scenes; starting from the dispatcher who correctly identified that this is a pertinent question and thus redirects the request to the insight module. Then most of the work takes place, as it should, within the insight module; from classifying the user's request (rightly classified as a legitimate request) to generating the SQL query and extracting the data from Athena. Below, you can see the SQL code generated by the model

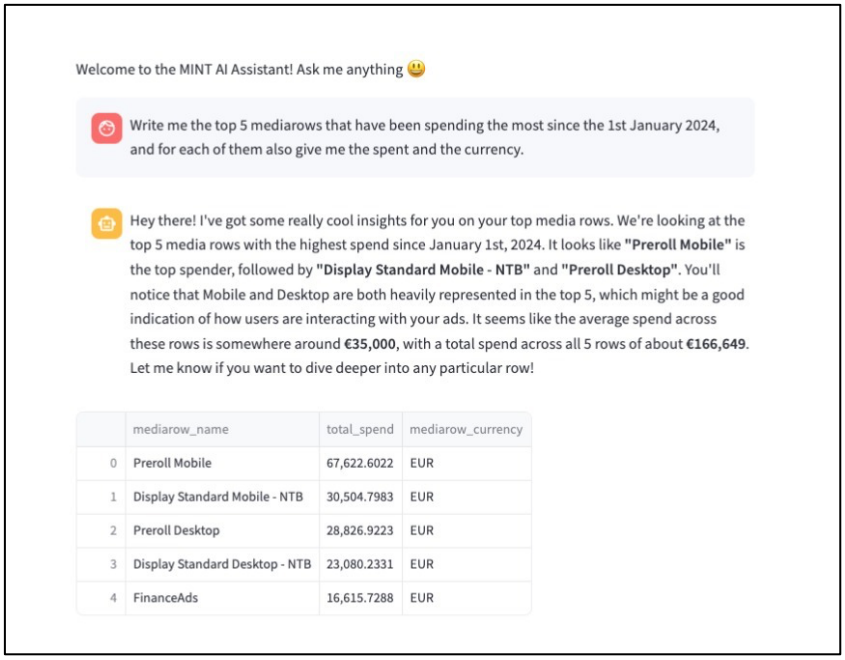

Figure 15: Classical interaction between AI assistant and user

```
SELECT dim.mediarow_name, SUM(pf.spend_in_mediarow_currency) AS total_spend, dim.mediarow_currency
FROM media_dim as m join performance as p on dim.mediarow_id = pf.mediarow_id
WHERE pf.date >= DATE('2024-01-01') and m.mediarow_end_date >= current_date
group by dim.mediarow_name, dim.mediarow_currency
ORDER BY total_spend
LIMIT 5
```

As can be seen, the generated query is not only clean and organized but also very effective, extracting all and only the requested data without performing superfluous operations. To generate such a response, the insight module takes the following times:

- Safety Check: 1.08s
- Aliases and example selector: 1.08s
- Prompt enrichment: 0.31s
- Query generation: 1.88s
- Data extraction: 1.78s

TOTAL: 6.14s

Considering that the Master Chat Handler redirecting the request to the model (through the Dispatcher) and generating the response took roughly 9.1s, this response perfectly satisfies the time requirements of obtaining a response in less than 10 seconds, as stated in 3.1.

Figure 16: AI assistant multilingual capability

The great advantage of using LLM models is their innate ability to handle multilingual conversations, provided, of course, that the model has been trained with texts in the relevant language. This eliminates the need for any translation mechanism, which

could otherwise result in an inaccurate translation of the user's request. Figure 16 shows an example of this capability.

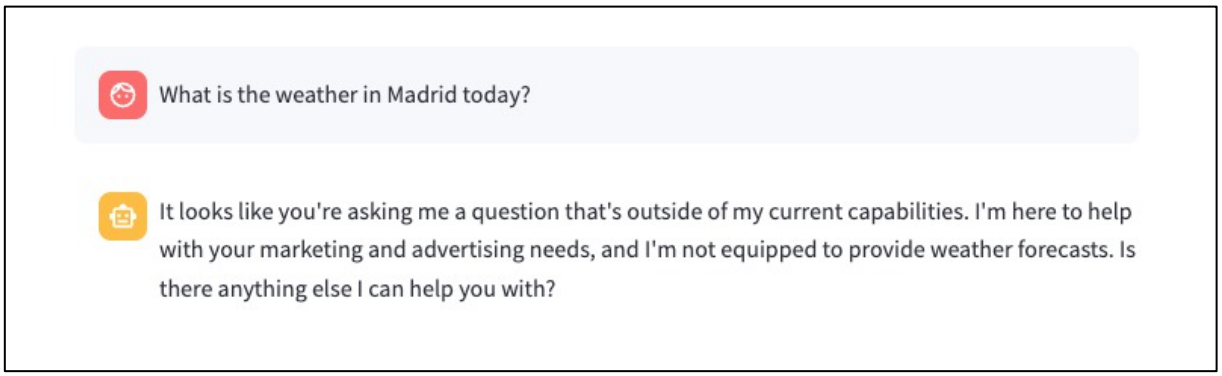

Figure 17: No pertinent question

If the user asks for something completely unrelated to the assistant purpose, the dispatcher will identify and stop this request before sending it to any module, as shown in Figure 17. This prevents an internal module from generating a completely no-sense result (due to the meaningless request).

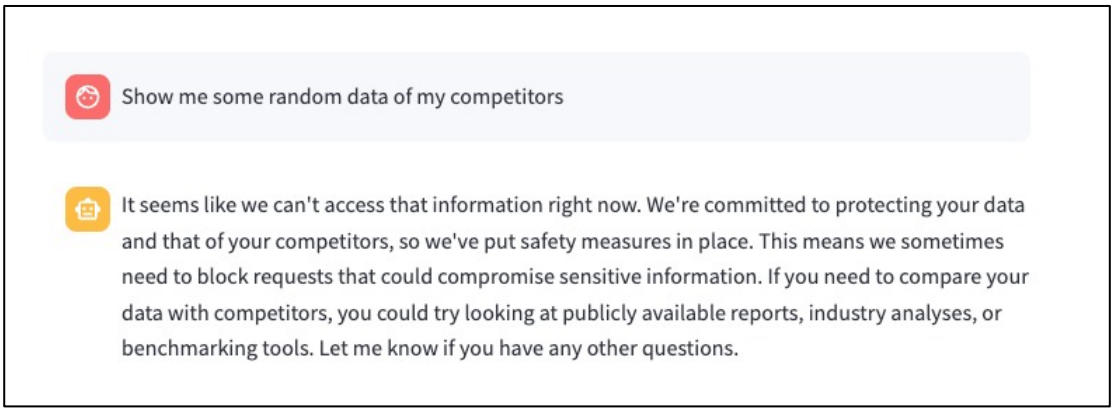

Figure 18: Safety Model catching the malicious request

The last case to be shown in the results of this work is how the safety model comes into action when the user makes a potentially malicious request. In the example shown in Figure 18, the user asks for random data about their competitors, and the assistant responds that security measures have been adopted to prevent access to unauthorized data. It is worth noting that in this case, the response time is much shorter, remaining within the order of 4 seconds.

5 Future developments

The work that has been developed and explained in this thesis has allowed for the complete implementation of the AI assistant architecture, including the integration of the module to provide the user with a data retrieval process simply by using written language. Obviously, it is not possible to fully achieve satisfactory results within a few months that can be integrated into such a complex product as ARM, so the work is open to many future developments that will now be listed.

5.1 Developments of other modules

The largest area of work involves the development and integration of various other modules that may be useful to the customer. Some modules have already been identified and are ready for development. Many of these involve the automatic generation of payloads so that the user can request the assistant to perform operations on the platform. Some examples of these operations include creating a new advertising campaign, modifying an existing one, or initiating the execution of some AI-powered products that are already in operation. In practice, these new modules would allow for even more effective interfacing between the user and the entire platform, not just the insights part.

These new modules would require extensive analysis to understand which LLM is more suitable for that particular use case (for example JSON generation might be a perfect scenario for a model that underperformed in SQL generation) and will be also necessary an integration of the Dispatcher discussed in section 3.11 to extend its classification capabilities switching from a classical binary classifier to a multi class one.

5.2 Build classification models from scratch

Another possible area of work could be developing models from scratch for tasks that are relatively simple, such as classifying a request to determine its maliciousness or not. As explained in the implementation chapter, due to time constraints and especially the cost/precision ratio, it was preferred to use LLM models even for binary classification problems. This is because, to use a custom model from scratch, it is first necessary to collect a large number of requests in natural language to create a dataset that is as complete as possible. However, this operation takes a lot of time, and the accuracy of a model trained on this dataset is unlikely to match the performance of an LLM model (which requires only a few lines of prompt to be 'trained'). However, developing an in-house model would reduce costs and execution time as it would not require API calls over the Internet. Therefore, creating these models from scratch is certainly a possible future development for this product.

5.3 More granular role permissions

As explained in section 3.12, to prevent unauthorized access to data, AWS security features have been utilized to ensure both table-level and row-level security. However, this may not be enough. Currently, within a single client, there can be various user profiles, each with specific data access permissions. For instance, a company might

have employees in different parts of the world, and each user may only access the advertising campaigns currently running in their own country if they are able to view information about campaigns in other countries. At present, this level of authorization has not been implemented. Therefore, an important next step would be to dynamically create the dataset that the assistant can work with, ensuring appropriate data access permissions for each user.

6 Conclusions

This thesis has successfully addressed the primary challenge of enhancing user experience on the ARM platform by developing an AI assistant capable of translating natural language requests into SQL queries. This solution offers users an intuitive and efficient way to interact with the platform information, significantly simplifying data retrieval and analysis.

The integration of Large Language Models (LLMs) has been essential of this project. By utilizing state-of-the-art models and implementing a dynamic example retrieval system, the AI assistant has shown remarkable accuracy and efficiency in generating SQL queries. The Max Marginal Relevance (MMR) algorithm played a crucial role in this success by selecting the most contextually relevant and diverse examples, which significantly improved the model ability to handle a wide range of user queries. Comprehensive testing and evaluation of various LLMs were conducted to ensure that the chosen model, Gemini 1.5 Flash, met the stringent requirements for both accuracy and response time. The results demonstrated that this model, augmented with the MMR algorithm, was capable of generating correct SQL queries in over 90% of cases. This high level of accuracy ensures that users can rely on the assistant to provide precise data retrieval, thus enhancing their overall experience on the platform.

Additionally, the inclusion of a safety model to classify potentially harmful requests adds an essential layer of security. This model, through meticulous, prompt engineering and the use of the Gemini 1.5 Flash model, ensures that any potentially malicious queries are identified and reported. This proactive approach to security not only protects the data but also enhances user trust in the system.

The AI assistant architecture, designed to be serverless and accessible via API, offers significant flexibility and scalability. This approach allows for easy integration of future advancements in LLMs and ensures that the system can adapt to increasing demands without substantial reconfiguration. The cost-effectiveness of this serverless solution also makes it an attractive option for businesses looking to improve their digital presence without incurring high operational expenses.

Moreover, the focus on user-centric design means that the AI assistant can understand and process natural language queries accurately. This capability reduces the learning curve for new users and increases engagement with the platform. By allowing users to interact with the system using natural language, the assistant makes data exploration more accessible and efficient, thus supporting better decision-making processes.

In summary, this thesis has not only addressed the usability issues of the ARM platform but has also significantly enhanced its functionality. The development and implementation of the AI assistant provide a powerful tool for managing digital advertising resources. This work sets a new standard for efficiency and user satisfaction in digital advertising platforms, paving the way for future advancements in AI-assisted user interfaces.

References

[1] Mint website. URL: mint.ai.

[2] AWS. Amazon bedrock. URL: https://aws.amazon.com/bedrock/?nc1=h_ls.

[3] AWS. Amazon dynamodb. URL: https://aws.amazon.com/it/dynamodb/.

[4] AWS. Aws lake formation. URL: https://aws.amazon.com/it/lake-formation/.

[5] AWS. Policies and permissions in iam. URL: https://docs.aws.amazon.com/IAM/ latest/UserGuide/access_policies.html.

[6] AWS. What are large language models (LLM)? URL: https://aws.amazon.com/what-is/ large-language-model/?nc1=h_ls. [7] AWS. What is amazon bedrock? URL: https://docs.aws.amazon.com/bedrock/latest/ userguide/what-is-bedrock.html.

[8] AWS. What is amazon glue? URL: https://docs.aws.amazon.com/glue/latest/dg/what-is-glue.html.

[9] AWS. What is amazon s3?URL: https://docs.aws.amazon.com/AmazonS3/latest/ userguide/Welcome.html.

[10] Google Cloud. Introduction to vertex ai. URL: https://cloud.google.com/vertex-ai/ docs/start/introduction-unified-platform.

[11] Google Cloud. Vertex-ai model information. URL: https://cloud.google.com/ vertex-ai/generative-ai/docs/learn/models.

[12] Cohere. Llm parameters demystified: Getting the best outputs from language ai. URL: https://cohere.com/blog/llm-parameters-best-outputs-language-ai.

[13] DeepL. How does deepl work? URL: https://www.deepl.com/it/blog/ how-does-deepl-work.

[14] Cambridge Dictionary. Advertising definition. URL: https://dictionary.cambridge.org/it/dizionario/inglese/advertising.

[15] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL: http://jmlr.org/papers/v23/21-0998.html.

[16] Google. Gemini pricing. URL: https://ai.google.dev/pricing.

[17] Sepp Hochreiter and Ju¨rgen Schmidhuber. Long short-term memory. *Neural computation,* 9(8):1735–1780, 1997.

[18] Heiko Hotz. Rag vs finetuning — which is the best tool to boost your llm application? URL: https://towardsdatascience.com/ rag-vs-finetuning-which-is-the-best-tool-to-boost-your-llm-application-94654b1eaba7

[19] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[20] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

[21] M I Jordan. Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986. 5 1986. URL: https://www.osti.gov/biblio/6910294.

[22] Andrea Knezovic. What are impressions in advertising? examples and types. URL: https://medium.com/udonis/
what-are-impressions-in-advertising-examples-and-types-806402851175.

[23] Sandra Kublik and Shubham Saboo. *GPT-3: The Ultimate Guide to Building NLP Products with OpenAI API*. Packt Publishing Ltd, 2023.

[24] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding, 2020. arXiv:2006.16668.

[25] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Ku¨ttler, Mike Lewis, Wen-tau Yih, Tim Rockt¨aschel, et al. Retrievalaugmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[26] Fahad Muhammad. Advertising conversion: Everything you need to know. URL: https: //instapage.com/blog/advertising-conversion-everything-you-need-to-know.

[27] Linyong Nan, Yilun Zhao, Weijin Zou, Narutatsu Ri, Jaesung Tae, Ellen Zhang, Arman Cohan, and Dragomir Radev. Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies, 2023. arXiv:2305.12586.

[28] openAI. Embeddings. URL: https://platform.openai.com/docs/guides/embeddings.

[29] OpenAI. Openai pricing. URL: https://openai.com/api/pricing/.

[30] openAI. Tokenizer. URL: https://platform.openai.com/tokenizer.

[31] OpenAI, Josh Achiam, Steven Adler,et al. Gpt-4 technical report, 2024. arXiv:2303.08774.

[32] Archit Parnami and Minwoo Lee. Learning from few examples: A summary of approaches to few-shot learning, 2022. arXiv:2203.04291.

[33] Marco Del Pra. Large language models. URL: https://medium.com/@marcodelpra/ large-language-models-1a6eec644b30.

[34] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

[35] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-ofexperts layer, 2017. arXiv:1701.06538.

[36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, L ukasz Kaiser, and Illia Polosukhin. Attention is all you need.

Advances in neural information processing systems, 30, 2017.

[38] Zi-qiang Wang, Xia Sun, De-xian Zhang, and Xin Li. An optimal svm-based text classification algorithm. In *2006 International Conference on Machine Learning and Cybernetics*, pages 1378–1381, 2006. doi:10.1109/ICMLC.2006.258708.

[39] Yonghui Wu, Mike Schuster, et. Al Google's neural machine translation system: Bridging the gap between human and machine translation, 2016.

arXiv:1609.08144.

[40] Xi Ye, Srinivasan Iyer, Asli Celikyilmaz, Ves Stoyanov, Greg Durrett, and Ramakanth Pasunuru. Complementary explanations for effective in-context learning,

2023. arXiv: 2211.13892.

[41] Terry Yue Zhuo, Yujin Huang, Chunyang Chen, and Zhenchang Xing. Red teaming chatgpt via jailbreaking: Bias, robustness, reliability and toxicity, 2023.

arXiv:2301.12867.

[42] AWS Bedrock Faq. URL: https://aws.amazon.com/bedrock/faqs/?nc1=h_ls#:~:text=Will%20AWS%20and%20third%2Dparty%20model%20providers%20use%20customer%20inputs%20to%20or%20outputs%20from%20Amazon%20Bedrock%20to%20train%20Amazon%20Titan%20or%20any%20third%2Dparty%20models%3F

[43] Google Gemini-API terms. URL: https://ai.google.dev/gemini-api/terms#:~:text=to%20those%20changes.-,Data%20Use%20for%20Paid%20Services,-When%20you%27re%20using

[44] OpenAI Policy FAQ. URL: https://help.openai.com/en/articles/5722486-how-your-data-is-used-to-improve-model-performance#:~:text=train%20our%20models.-,Services%20for%20businesses%2C%20such%20as%20ChatGPT%20Team%2C%20

ChatGPT%20Enterprise%2C%20and%20our%20API%20Platform,-We%20don%E2%80%99t%20use

[45] Bayesian LSTM on PyTorch — with BLiTZ, a PyTorch Bayesian Deep Learning library, URL: https://towardsdatascience.com/bayesian-lstm-on-pytorch-with-blitz-a-pytorch-bayesian-deep-learning-library-5e1fec432ad3

[46] Transformer Architecture: Attention is all you need. URL: https://medium.com/@adityathiruvengadam/transformer-architecture-attention-is-all-you-need-aeccd9f50d09

[47] Comparison of representations between one hot encoding and word embeddings. URL: https://www.researchgate.net/figure/Comparison-of-representations-between-one-hot-encoding-and-word-embeddings_fig3_358234888

[48] Prospect uses Amazon QuickSight self-service business intelligence tools to empower clients. URL: https://community.amazonquicksight.com/t/prospect-uses-amazon-quicksight-self-service-business-intelligence-tools-to-empower-clients/31414

[49] Faiss: A library for efficient similarity search URL: https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search/

[50] Simplified Gating in Long Short-term Memory (LSTM) Recurrent Neural Networks , Yuzhen Lu and Fathi M. Salem. URL: https://arxiv.org/pdf/1701.03441

[51] Mathematical Representation of Recurrent Neural Network. URL: https://www.researchgate.net/figure/Mathematical-Representation-of-Recurrent-Neural-Network_fig2_340741413

[52] Structure of multi-head attention layer. URL: https://www.researchgate.net/figure/Structure-of-multi-head-attention-layer_fig3_334427742

[53] The Use of MMR, Diversity-Based Reranking for Reordering Documents and Producing Summaries. URL: https://www.cs.cmu.edu/~jgc/publication/The_Use_MMR_Diversity_Based_LTMIR_1998.pdf