University of Verona
Department of Informatics
Master's Degree in Data Science

~ · ~

Academic Year 2021–2022

Leveraging LLMs with RAG to recommend Points of Interest to tourists

Supervisor
Prof. Niccolò Marastoni

Graduate Student
Patrick Hamzaj
VR474246

Dedica

Acknowledgments

Abstract

Qui va l'abstract

Contents

Chapter 1

Introduction

Over the last decade, the field of artificial intelligence (AI from now on) has experienced significant growth and rapid advancements. Among the different branches of AI, *deep learning* has stood out for its ability to tackle complex tasks that were considered infeasible or extremely challenging. In particular, *Natural Language Processing* (NLP) has gained prominence due to the increasing volume of unstructured and textual data generated and subsequently gathered online (e.g., social media, blogs, scientific articles...); it is estimated that 328.77 million terabytes of data are created daily, 80% of which are unstructured data. [1] This phenomenon and the rising demand for language-driven applications, such as virtual assistants, sentiment analysis, and automated content generation, has prepared the terrain for the advancement of languange processors to flourish.

Recently, the emergence of *Large Language Models* (LLMs), led by the release of GPT-3 in 2020 [2], has taken NLP to a new level. By employing architectures with billions of parameters, these models are capable of producing remarkably fluent, context-aware and human-like text. Despite this progress, several open challenges remain: questions regarding how best to *fine-tune* LLMs, how to incorporate domain-specific knowledge, and how to design effective prompting mechanisms are active areas of research. Additionally, issues related to computational resources, scalability, ethical implications, and potential biases call for continuous investigation.

Motivations. The primary motivation behind this thesis is to explore the practical methods and architectural choices to enable the construction of an LLM-powered application to be effectively adapted for the domain of tourism in Verona, Italy. In particular, the aim is to utilize and demonstrate how techniques such as *prompt engineering*, and *retrieval-augmented generation* (RAG) can facilitate the deployment of state-of-the-art LLMs in real-world applications, including scenarios where computational resources may be limited or costly.

As such, the **objectives** are:

- **Analyze** the foundational principles and evolution of modern neural network architectures, particularly focusing on the *Transformer* model and its role in Large Language Models.
- **Investigate** the current state of the art for LLMs, highlighting successful applications and the most common methodologies (fine-tuning, prompt engineering, RAG, etc.).
- **Design and implement** a tourism domain application system leveraging a specific open-source LLM (LLaMA 3.1 8B Instruct), showcasing relevant techniques (8-bit

quantization, agent-like conversational behavior) for an implicit recommender system for tourists.

- **Evaluate** the system with respect to performance and quality metrics, as well as potential limitations.

From a high-level perspective, Large Language Models represent a class of deep neural netowrks trained on extensive corpora, often comprising billions of parameters. A key turning point in their development was the introduction of the *Transformer* architecture, which provides a self-attention mechanism that allows to improve text tokens processing in a parallel manner during training.

Examples of prominent LLMs include GPT-based models (GPT-3, GPT-3.5, GPT-4), [2] Google's PaLM [3], Meta's LLaMA, [4] and various open-source initiatives such as Bloom. [5] These models have demonstrated remarkable capabilities in language understanding and generation, enabling:

- **Text completion** with human-like fluency.
- **Zero-shot or few-shot generalization** to new tasks with minimal prompt examples.
- **Conversational AI**, powering advanced chatbots and virtual assistants.

This thesis will leverage the open-source LLaMA model family as a foundation for an LLM-based chatbot to suggest points of interest to touristi visiting the city of Verona, Italy. By focusing on a quantized 8-bit instruct variant (LLaMA 3.1 8B Instruct), the aim is to highlight practical techniques (RAG, agent AI) that maintain acceptable performance implementing innovative strategies that harness context-awareness and text generation capabilities/question answering of modern LLM models.

The remainder of this document is organized as follows: Chapter 1 presents a deeper dive into the theoretical underpinnings of modern NLP, covering the history of neural networks and the transformation from classic word embeddings to large-scale language models, also detailing the breakthrough Transformer architecture. Chapter 2 reviews the current landscape of Large Language Models and NLP solutions, exploring established techniques (fine-tuning, prompt engineering, RAG) and highlighting real-world use cases, from chatbots to advanced content generation. Chapter 3 describes the design and implementation of an application based on LLaMA 3.1 8B Instruct for tourism purposes in the city of Verona, Italy. It covers the decision-making process behind the model choice, 8-bit quantization strategy, prompt engineering, retrieval-augmented generation, and the agent-like conversational framework. Chapter 4 discusses the evaluation with illustrative examples of the model's outputs and conversations, along with limitations and areas for improvement. Finally, Chapter 5 concludes the study with a summary of key findings, limitations, and potential future directions.

Chapter 2

Theoretical Foundations and Background

2.1 A Brief History of Neural Networks

The term *Neural Network* refers to the attempt of defining a mathematical view of the structure of the human brain, aiming at emulating it and transposing the logical functioning and the learning capabilities into computational models.

The very first attempt of studying the biological brain and its neural activity in terms of formal logic is attributed to McCulloch and Pitts,[6] who proposed that neurons could be represented as simple binary devices, whose key aspects are:

- **Logical Units:** neurons are basic units with an on/off switch, allowing them to describe neural activi'ty using the language of logic (i.e. logical propositions like *AND*, *OR*...).
- **Threshold:** neurons "fire" only when a certain threshold of input is met.
- **Links:** like synapses, links connect the logical units from the input to the output.

This basic understanding of neural processes influeced later developmentes in neuroscience and artificial intelligence, bringing Rosenblatt to produce the *Perceptron*: a single layer neural network that, upon previous work from McCulloch and Pitts, introduced:

- **Weights:** every connection has a weight, that is added to the input received.
- **Learning mechanism:** introduces a learning algorithm where the weights of the connections are adjusted based on errors in the output.

This linear architecture allows to generalize the classification tasks, using probabilistic rules that could perform nontrivial tasks like pattern recognition and information organization.

Neural networks have evolved dramatically over the decades

The enthusiasm was enormous and the field of cybernetics was born: however, it didn't take long for researchers to uncover the limitations of single-layer networks. Minsky's and Papert's demonstrated the limitations of the perceptron—that is, that certain classes of functions were simply out of reach for these early models (for example, the logical XOR

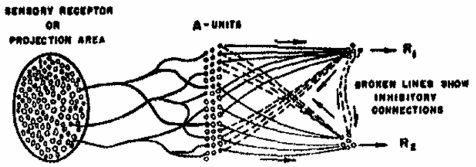

FIG. 2A. Schematic representation of connections in a simple perceptron.

Figure 2.1: The perceptron architecture.

function)—interest quickly waned. [7] This realization contributed to a period of reduced enthusiasm for neural network research, often referred to as the "AI Winter."

Interest was revived in the 1980s with the introduction of the backpropagation algorithm [8], which made it possible for deeper architectures to learn more complex functions. The core idea was to train neural networks by propagating the error from the output layer backward through the network layers; this approach allows the network to adjust its internal weights based on the error itself, so that it could "learn" using gradient descent to minimize the error function, changing the weights individuating the contribution of each neuron in constructing the output.

Although early progress was hampered by hardware constraints, continuous incremental improvements over the following decades, combined with advances in parallel computing, eventually paved the way for the deep learning revolution we see today [9].

By the early 2010s, the impact of Convolutional Neural Networks (CNNs) on computer vision tasks [10] highlighted the benefits of large datasets, GPU-based parallel training, and increasingly sophisticated network designs. At the same time, improvements in Recurrent Neural Networks (RNNs)—especially with LSTM [11]—opened up new possibilities in sequence modeling, including areas like language translation and speech recognition. These advancements ultimately set the stage for the development of Large Language Models (LLMs), particularly after the introduction of the Transformer architecture [12].

2.2 The Transformer Architecture

The introduction of the Transformer by Vaswani et al. [12] marked a clear departure from traditional recurrent and convolutional models. The focus of the original research was on translation tasks, by leaning heavily on a **self-attention** mechanisms which allows the model to weigh the importance of different words in a seuqence of words, taking into account the relationship to eacht ore.

The Transformer architecture consists of two main parts: an **encoder** and a **decoder**, each built from multiple layers that implement multi-head self-attention alongside feed-forward networks, which can be visualized in 2.2:

1. **Encoder:** it converts an input sequence (e.g., a sentence) into a series of representations that capture the contextual meaning of the input. Within each encode layer, a sub-layer allows every token in the input to consider the influence of every other token.

2. **Decoder:** it uses the encoder's representation along with other inputs to generate a target sequence using a masking mechanism (i.e. predicting the next token is only decided to previous tokens, and not future ones).

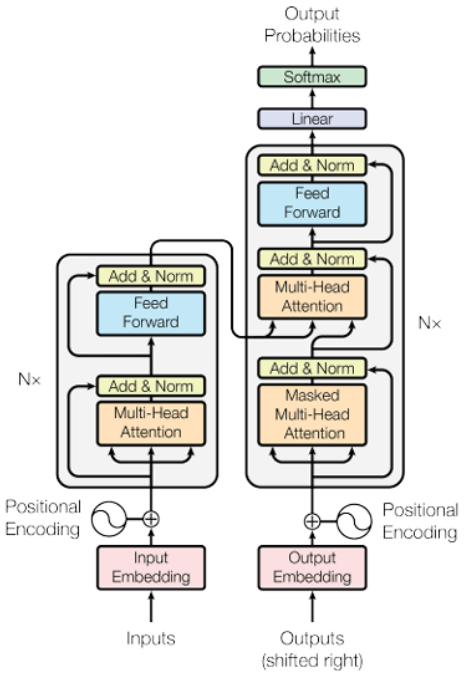

Figure 2.2: The Transformer architecture.

Positional encoding is another breakthrough of this paper, integrated in both parts, which compensates for the lack of sequence awareness. It incorporates information about the order of tokens in a sequence. Since these models process tokens in parallel rather than sequentially, positional embeddings are essential for conveying information about the order of tokens.

Each of these parts can be used independently, depending on the tasks: encoder-only models are good for tasks that require understanding of the input, such as sentence classification; decoder-only models are good for generative tasks such as text generation. Encoder-decoder models (called sequence-to-sequence models) are used for generative tasks that require an input, such as translation or summarization.

2.3 Transformers as Language Models

In recent applications, the Transformer model have been trained as *language models* meaning they have been trained on large amounts of raw text in a self-supervised fashion, which is a type of training in which the objective is automatically computed from the inputs of the model.

This type of models develop a statistical understanding of the language it has been trained on, but it's not very useful for specific practical tasks. Because of this, the general pretrained model then goes through a process called *transfer learning*. During this process, the model is fine-tuned in a supervised way — that is, using human-annotated labels — on a given task.

An example of a task is predicting the next word in a sentence having read the previous

words. This is called causal language modeling because the output depends on the past and present inputs, but not the future ones.

Training is a foundamental step in the adoption and implementation of a language model. The learning capacity of LLMs are generally dived into:[13]

1. **Pre-training:** it is the first stage in training an LLM, where the model learns general linguistic patterns, facts and knowledge from a vast corpus of text. It is the act of training a model from scratch: the weights are randomly initialized, and the training starts without any prior knowledge. Tecniques of pre-training phase are:

   - *Masked language modeling*, used in decoder-models, where certain words are masked and the model learns to predict them.[14]
   - *Causal language modeling*, used in encoder-only models like GPT,[2] where the model predicts the next word in a sequence.

2. **Fine-tuning:** after pre-training, the model undergoes a further training on a smaller, task-specific dataset to improve performance for particular and domain-specific applications. Types of fine-tuning are:

   - *Supervised fine-tuning* is used to train models on labeled data, such as question-answering datasets.
   - *Instruction tuning* involves training the model on a dataset of input-output pairs, where each input is phrased as an instruction and the output is the desired response. Most ready-to-use models are instruct-tuned, as they have improved generalization and natural responses. An example can be seen in Table 2.1.
   - *Parameter-Efficient fine-tuning (PEFT)*, methods like *Low-Rank Adaption (LORA)*[15] are innovative techniques that reduce the number of parameters to train, thus reducing computaional costs.

Generally, the strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pretrained on, but higher performances lead to higher resources intensive trainings. This is why different strategies have been developed to achieve good performances without the need of training models.

Instruction	**Input (optional)**	**Expected Output**
Translate this sentence into Spanish	"Hello, how are you?"	"Hola, ¿cómo estás?"
Summarize the text in one sentence	"The global economy is facing uncertainty due to inflation and geopolitical issues."	"The global economy is unstable due to inflation and geopolitics."
Explain how photosynthesis works to a 5-year-old	*No input*	"Plants use sunlight to make food, like how we eat to get energy!"

Table 2.1: Examples of Instruction Tuning

2.4 Challenges and Limitations of LLMs

2.5 Summary

Chapter 3

Recent Advances in Large Language Models

Large Language Models (LLMs) have become a cornerstone of modern Natural Language Processing (NLP), enabling a wide range of applications such as chatbot systems, code generation, and content creation: it is estimated that by the end of 2025 there will be 750 million apps using LLMs as a source of interaction.[16] This chapter provides an overview of current leading LLM families and their innovations, explores key techniques for their customization, introduces the concept of Retrieval-Augmented Generation (RAG), Tool Utilization and Prompt Engineering.

3.1 Major Model Families and Their Characteristics

Most LLMs nowadays are built on the Transformer architecture with hundreds of millions to hundreds of billions parameters (and, in recent works, over a trillion), which take advantage of the availabilty of large-scale text corpora on the internet, along with improvements in scalable hardware (GPUs, TPUs) and advances in Transformer architecture, that adhere to the *scaling law*[17] for which systematicallt scaling model size, data size, and compute lead to improved performance in LLM models.

Overall, LLMs nowadays can be divided into three main families.[18]

3.1.1 GPT Family (Generative Pre-Trained Transformers)

The GPT family consists of decoder-only models developed by OpenAI, introducing GPT in 2018, which is defined as an undirectinal auto-regressive model. The key innovation here is that it showed that large-scale, unsupervised pre-training followed by a punctual fine-tuning on specific tasks that the model is supposed to achieve, could outperform traditional architectures of many NLP benchmarks.

A year later, the company introduced GPT-2, scaling up the parameter count up to 1.5 billion parameters: it demonstraded zero-shot capabilities on tasks like summarization or translation, only using prompting for interacting with the model.

The term *zero-shot capability* refers to the ability for a model to prompting an LLM without any examples, attempting to take advantage of the generalization it has gained through training, using reasoning patterns it has acquired.

GPT-3 saw another scale up in parameters count, achieving a surprising 175 billion count: by giving few-shot examples at inference time, it demonstrated an *in-context learning*, for which no additional fine-tuning was required.

Finally, GPT-3.5 introduced the rinomate *ChatGPT*, which introduced LLMs to the general public. It incorporated instruction-following behavior and Reinforcement Learning from Human Feedbak (RLHS) to reduce harmful content; later, with GPT-4, it introduced multi-modal capabilities interacting with and generating non-textual data.

Overall, the GPT Family has been acclaimed for its investments in scaling models with higher paramateres and text corpora, achieving few-shot capabilities without the need to further training models.

3.1.2 BERT Family (Bidirectional Encoder Representations)

The BERT-family was introduced by Google in 2018, and it focuses on tasks that require understanding of the input, such as sentence classification and named entity recognition. In this context they utilize and encoder-only architecture, performing a masked contextual understanding, meaning that their training in based mainly on masking tokens at each iteration and gaining a probabilistic choice of the next token prediction.

This kind of models are strong at understaing the semantics of text from a bidirectional perspective: through Masked Language Modeling (MLM), the training consists of randomly masking 15% of the tokens in the input and ask the model to predict the original vocabulary tokens, encouraging to learn from the left and from the right sides of the masked token (that is, bidirectional).

BERT is by design a 110 million parameters in its base version, whereas it has 340 million parameters in its largest one, which has seen several variants over the years, like RoBERTA which increased training data and steps involved in the training phase, or ALBERT, with a paremter-reduction technique to handle scaling more efficiently, and finally DistilBERT, that implemented *knowledge distillation*: a novel technique in making LLMs more practical and efficient, by transferring essential knowledge from a complex teacher model to a smaller student model, preserving performance while reducing size and computational demands.

3.1.3 T5 Family (Text-to-Text Transfer Transformer)

Introduced as well by Google (2019), the T5-family model uses a full sequence-to-sequente (i.e., encoder-decoder) Transformer architecture. The core idea is to formulate every NLP task as a "text-to-text" problem-inputs and outputs are always text strings.

Prior to T5, large-scale models (e.g., GPT, BERT) had shown the power of pre-training on unlabaled data followed by task-specifi fine-tuning; however, these approaches often frame tasks differently (classification vs prediction vs language modeling). This divergence complicates the use of a single frameworks for multiple tasks: so, the goal was to create a unified approach to NLP tasks by casting all problems into a text-to-text format.

The authors released multiple variants, with the largest size counting up to 11 billion parameters, and their approach in training was a modifief version of MLM, which is Span Corruption: randomly masking not individual tokens, by entire span of texts. This demonstraded to foster a more coherent learning of consecutive tokens and encouraged the model to handle variable-length contexts.

The T5 has been superseded later on by mT5 and T5-XXL, with larger parameter counts and multilingual corpora for cross-lingual transfer, which yields strong results across different tasks, highlighting flexibility for multitask settings.

These families do not comprise the several hundreds of foundation models released in recent years, but it nonetheless give a high perspective of the model architectures, their characteristics and key innovations. A common strategy has been underlined across the realm of these families: the bigger the size, the better. That is why training has always been a prerogative of the few, large companies or research centers that can handle higher computational resources and costs.

3.2 Beyond Fine-Tuning: Alternative Strategies for Optimizing LLMs

Given the computational challenges of fine-tuning, alternative approaches such as Retrieval-Augmented Generation (RAG), Prompt Engineering, and Agent AI have emerged as effective strategies for achieving task-specific accuracy without the need for extensive model retraining.

As LLMs become increasingly prevalent, practitioners have devised a range of methods to tailor their behavior for specific use cases and to deploy them efficiently. Fine-tuning remains a common approach, allowing developers to update the model weights on domain-specific datasets. This can be performed as a full-scale process, adjusting all parameters, or via more parameter-efficient methods such as LoRA [15], which inserts low-rank updates into the model's layers and thus reduces memory requirements; this approach still produces prohibitive settings to develop LLM specific implementations, that is why innovative techniques have emerged, gathering fine-tuned models that possess general knowledge directing its focus on specific tasks and use cases.

Prompt engineering, takes advantage of the model's pre-trained knowledge by carefully crafting textual instructions that guide it toward the desired output [19]. Rather than modifying the model weights, prompt engineering modifies the input context to clarify the task objective or to showcase example queries and answers. This practice has grown in importance with the rise of instruction-tuned models, which are trained to follow natural language instructions rather than purely statistical patterns.

Another pivotal approach involves integrating an external knowledge base into the generation process. Retrieval-Augmented Generation (RAG)[20] leverages a retrieval module, typically built on vector search engines, to fetch relevant documents or snippets from a corpus. The LLM then conditions its output on these retrieved texts, thereby grounding its responses in verifiable sources and reducing hallucination or factually incorrect statements[21].

Lastly, the concept of Tools have introduced the term "AI Agents". Tools are external modules or functions that the model can decide to invoke, to perform tasks or gather knowledge which was not comprised in its training and go beyond text generation. It leverages the model's internal decision mechanism that will determine if an action must be taken based on users' input. For example, asking what the weather is like right now is excluded by the factual limited knowledge the model possesses; it can be thereby create a tool that calls an API with weather data, and when the trigger has been captured by the model it can decide to call this tool to retrieve the updated data.

In the next sections we will dive into these innovative strategies to benefit from pre-trained models with powerful text-generation abilities of LLMs without incurring into expensive setups. Nervertheless, it must be first introduced the concept of prompting and the interaction with a language model.

3.2.1 From Words to Numbers and Back to Words: Tokenization

Tokenizers are one of the core components of the NLP pipeline, which hey serve one purpose: to translate text into data that can be processed by the model. As in language models the data that is generally produced is raw text, and knowing that Transfomers can only process numbers, tokenizers need to convert text inputs to numerical data. So, the goal of tokenizers is to find the most meaningful representation — that is, the one that makes the most sense to the model — and if possible the smallest representation.

Tokenization splits text into mangeable units (that is, *tokens*) to enable the model to understand and process language, which are then converted into numerical representations that capture their semantics. It can be as simple as a word-based tokenization (i.e., each word is a token), to more complex techniques like *WordPiece* used in BERT models.

Each token is defined by the method used, for which the first that comes to mind would be a word-based tokenization. As an example, having the phrase "John was a puppeteer" could be divided into

```
Jim | Henson | was | a | puppeteer | .
```

And the goal would be to assign a numerical representation of each token

```
Jim | Henson | was | a | puppeteer | .
545 | 4668   | 109 | 9 | 10988     | 721
```

Each word gets assigned an ID, starting from 0 and going up to the size of the vocabulary: then the model uses these IDs to identify each word. But if we want to completely cover a language with a word-based tokenizer, we'll need to have an identifier for each word in the language, which will generate a huge amount of tokens. For example, there are over 500,000 words in the English language, so to build a map from each word to an input ID we'd need to keep track of that many IDs. Furthermore, words like "dog" are represented differently from words like "dogs", and the model will initially have no way of knowing that "dog" and "dogs" are similar: it will identify the two words as unrelated. The same applies to other similar words, like "run" and "running", which the model will not see as being similar initially.[22]

Other simple tokenization techniques could come to mind, for example a character-based tokenization, but the reality would be very complex and could not tackle the semantics of the context. Several techniques have been developed, of which the most important are:[23]

- **Byte-Pair Encoding (BPE):** Used by models like GPT-2 and GPT-3, BPE starts with individual characters and iteratively merges the most frequent pairs to form subwords. This helps manage vocabulary size and handle rare or novel words. [24]
- **WordPiece:** Employed by BERT, WordPiece is similar to BPE but merges tokens by maximizing the likelihood of token sequences. It helps capture the structure of language by efficiently splitting words into meaningful subword units.
- **SentencePiece:** Often used in models like T5, SentencePiece does not rely on pre-tokenized input. It treats the text as a raw sequence of characters (or bytes) and learns subword units directly, making it versatile across languages and scripts.

The process of extracting meaning from raw text to discrete representation and then back to words can be seen in Figure 3.1.

Translating text to numbers is known as *encoding* (not to be confused with **encoders** in the Transformer architecture). Encoding is done in a two-step process: the tokenization, followed by the conversion to input IDs, of which the set of all possible IDs is called a *vocabulary* of the model that learned during the training phase.

Each input ID is then converted into a high-dimensional vector that represents the contextual understanding of that input by the Transformer model. The subsequent layers of the architecture manipulate those vectors using the attention mechanism to produce the final representation of the sentences. [12]

The model outputs logits for each position in the sequence (for language modeling, this is typically predicting the next-token distribution): logits are real-valued scores for each token in the vocabulary before activation.

These scores are then converted to probabilities, yielding a probability distribution over the vocabulary for the next token to predict through a *SoftMax* layer

$$\text{Logits} = [1.3, 5.1, 2.2, 0.7, 1.1]$$

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

$$\text{Probabilities} = [0.02, 0.90, 0.05, 0.01, 0.02]$$

which turns logit scores into probabilites. Finally, the *decoding* phase chooses the tokens to output, for which a few strategies could be adopted: [25]

- **Greedy Decoding (Argmax)**: Choose the token with the highest probability at each step.
- **Beam Search:** Maintain multiple candidate sequences to reduce the chance of missing higher probability sequences.
- **Sampling:** Randomly sample from the distribution to introduce diversity.
- **Top-k/Top-p Sampling:** Limit sampling to the top-k most probable tokens or cumulative probability p, improving coherence and variety.

The predicted token IDs are finally mapped back to subwords or tokens via the tokenizer's vocabulary, and subwords are joined or merged to form readable text.

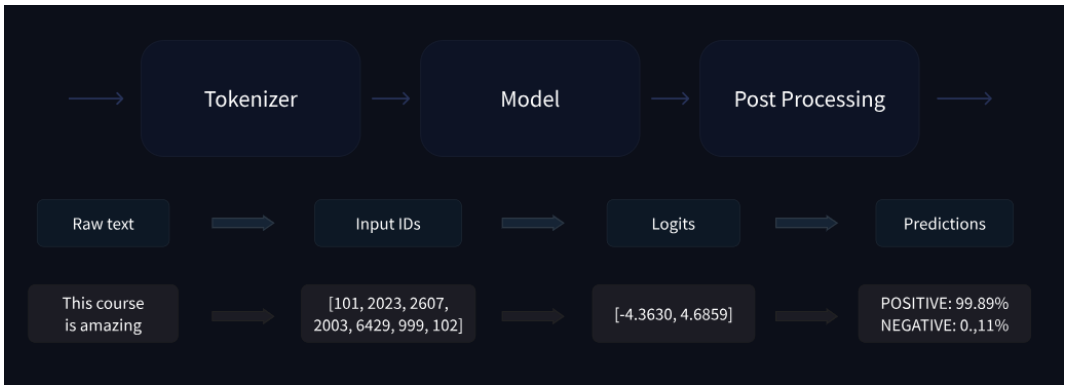

Figure 3.1: The tokenization process referring to text classification, as seen in [22].

3.2.2 Anatomy of a Prompt

A **prompt**, sometimes referred to as context, is the text provided to a model before it begins generating output; it guides the model to explore a particular area of what it has learned so that the output is relevant to the prefixed tasks. Frequently, prompts will be an instruction or a question:

```
Question: Explain the theory of gravity to a 6 year old.
Answer: Gravity is like a big invisible hug that pulls everything down.
↪ It's like an invisible rope that connects everything in the world,
↪ and it pulls things closer together. When you jump yp in the air,
↪ gravity pulls you back down to the ground. That's why you can't
↪ just fly away!
```

In applications where a user is interacting with a model dynamically, such as chatting with the model, there will typically be portions of the prompt that are never intended to be seen by the user. These hidden portions may occur anywhere, though there is almost always a hidden prompt at the start of a conversation. Typically, this includes an initial chunk of text that sets the tone, model constraints, and goals, along with other dynamic information that is specific to the particular session – user name, location, time of day..

The model is static and frozen at a point in time, so if the user wants the model to know current information, as for example time or weather, it must be provided to the model itself.

A common standard in chat models has been delineated by OpenAI, for which a categorization of roles has emerged from empirical design choices in conversational AI systems (namely, ChatGPT), where:

1. **System:** Sets the overall context, rules, and instructions that govern the conversation. It provides the "hidden" guidelines the assistant follows to ensure responses are consistent with desired policies, style, and behavior.

2. **User:** The user is the one providing the queries, prompts, or instructions that drive the conversation. Their input defines what information or assistance is being requested, and the assistant responds accordingly.

3. **Assistant:** This is the language model's persona that generates responses based on the system instructions and user inputs. The assistant's job is to provide helpful, accurate, and context-aware answers while adhering to the guidelines set by the system.

Guidelines can be found in OpenAI's documentation, following Table 3.1. For example, a conversation following the abovementioned roles could be:

```
SYSTEM: You are a helpful assistant that answers programming questions
↪ in the style of a southern belle from the southeast United States.
USER: Are semicolons optional in JavaScript?
ASSISTANT: Well, sugar, that's a fine question you've got there! Now,
↪ in the world of JavaScript, semicolons are indeed a bit like the
↪ pearls on a necklace, you might slip by without 'em, but you sure
↪ do look more polished with 'em in place.
Technically, JavaScript has this little thing called "automatic
↪ semicolon insertion" where it kindly adds semicolons for you where
↪ it thinks they oughta go.
```

```
However, it's not always perfect, bless its heart.
Sometimes, it might get a tad confused and cause all sorts of
↪  unexpected behavior.
```

It can be noted that the use of the message types above to provide additional information to the model outside of its training data has spiked an answer in the style requested by the context. The idea of role prompting is rooted in the broader literature on prompt engineering and instruction tuning, [26] and several strategies have been developed in order to make the most out of prompting.

Role	**Description**	**Usage example**
user	Instructions that request some output from the model. Similar to messages you'd type in ChatGPT as a user.	*Write a haiku about programming.*
system	Instructions to the model that are prioritized ahead of user messages, following chain of command. Previously called the system prompt.	*Describe how the model should generally behave and respond.*
assistant	A message generated by the model, perhaps in direct response to the current request.	*For example, to get the model to respond correctly to knock-knock jokes, you might provide a full back-and-forth dialogue of a knock-knock joke.*

Table 3.1: Examples of Instruction Tuning

3.3 Prompt Engineering

Prompt engineering is defined as the process of designing and structuring instructions (prompts) to guide LLMs toward producing the most effective outputs without modifying the models' internal parameters. In this context, a prompt is more than just a query—it's a carefully crafted input that may include context, instructions, role definitions, and even examples to help the model understand the desired task.[27]

Several techniques have been demonstrated to enhance models' performance in various tasks, with regard to increasing model sizes.[2]

3.3.1 In-Context Learning

When interacting with a lanugage model, we usually ask a question or give a command in general. The same question asked or instruction given generate a different response each time, as the determinism of the response itself is not guaranteed due to the stochastic nature of the Transformers output. At the same time, questions or instructions given in different

ways returns different responses: providing examples of the task we are trying to carry out is called **In-Context Learning**.

In-Context Learning refers to the ability of a language model to perform new tasks by leveraging examples provided directly within the prompt, rather than through explicit parameter updates. The model infers the underlying task by observing a set of input-output pairs included in the prompt and then generalizes that pattern to generate a response for a new input. This process effectively allows the model to "learn" from context during inference time.

When a language model is given a task instruction without any example inputs or outputs, relying solely on its pre-trained knowledge and the explicit prompt, it is referred as *Zero-Shot Prompting*. It tests the model's ability to generalize to tasks it was never explicitly trained on, solely by interpreting the instruction given in natural language. For example,

```
USER        Message: Hi Amit, I loved my birthday card!
            Sentiment:
ASSISTANT   Sentiment: Positive.
```

Here the task is to classify the sentiment of the sentence provided. It does not include any example on how to achieve the result, any hint on how to interpret the task or how to format it. Zero-Shot is not always performant, especially in low-rank paramateres models, as shown in Figure 3.2.

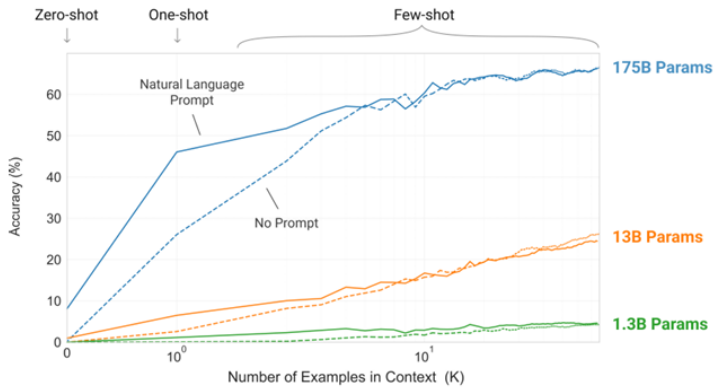

Figure 3.2: Different prompting techniques can increase models' accuracy, as seen in [2].

Providing examples can help the model to understand the task:

```
USER        Message: Dad, you are 20 minutes late for my piano
↪  recital!!
            Sentiment: Negative
            Message: Hi Amit, I loved my birthday card!
            Sentiment:
ASSISTANT   Sentiment: Positive.
```

Giving the model an example, and then the task we want it to perform is called *One-Shot Prompting*; two or more examples given to the model is called *Few-Shot Prompting*. As shown in Figure 3.2, increasing the number k of examples along with the increasing number of paramateres held by the model's architecture improve task accuracy.

3.3.2 Chain-of-Thought

Chain-of-Thought is a novel method to enhance the reasoning capabilities of large language models by encouraging them to generate intermediate steps—what the authors call a "chain of thought"—before arriving at a final answer. [28]

This technique allows models to decompose multi-step problems into intermediate steps, suggesting how it might have arrived at a particular answer and providing opportuinties to debug where the reasoning path went wrong; it has been demonstrated to be useful for tasks such as symbolic manipulation, commonsense reasoning and arithmetic problems, as can be seen in 3.3, where the Chain-of-Thought reasoning process is highlighted compared to standard prompting.

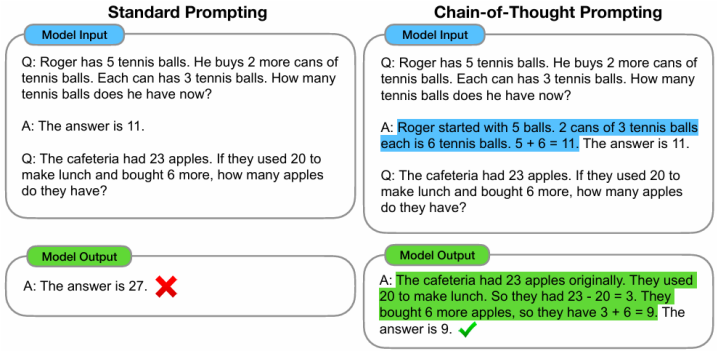

Figure 3.3: Standard prompting compared to Chain-of-Thought prompting, as seen in [28].

Traditional prompting methods typically ask the model to produce an answer directly, but when it comes to multi-step reasoning task-as the simple arithmetic problem in Figure 3.3-this approach often fails. This is why guiding the model to reason through intermediate steps can lead to improved performance because, instead of forcing the model to output a single-step response, one can prompt it to "think out loud" by generating a sequence of reasoning steps. This simultaes a humanl-like problem-solving process, where intermediate calculations or logical steps are made explicit before reaching a conclusion.

3.3.3 Role Prompting

Role Prompting is the practice of explicitly instructing a language model to adopt a specific persona or role within the prompt. By doing so, the model is provided with contextual cues about the kind of responses expected. For example, a prompt might begin with "You are an expert financial analyst," which sets the stage for subsequent output to be informed by that role. This approach leverages the model's vast pre-trained knowledge while narrowing its focus to the style, tone, and depth that a particular role implies. [29]

Framing the task in a way that aligns with the attributes of the designed role, can lead to more coherent and contextually appropriate responses, as the model "activates" a domain-specific language pattern and knowledge; when a model is prompted with a clear role, its output tend to be better aligned with the requirements. Just think of an assistant in medical or judicial domain, must adhere to a specific language and adapt itself to the context involved. To appreciate the different results produced by different roles can be seen in the following demonstration: [30]

Acting as a lawyer:

```
USER        You are a Lawyer. You have knowledge about the law, past
↪  precedence, and other relevant information. Always answer with the
↪  goal of giving advice on how to best use the situation but also
↪  stay within the parameters of the law.
Respond to the following:
"Explain the most important way ChatGPT can help me. The answer should
↪  be no more than 100 words."

ASSISTANT   As a lawyer, I understand the importance of providing
↪  concise and clear advice to my clients. In your case, the most
↪  important way that ChatGPT can help you is by providing you with
↪  accurate and up-to-date legal information and guidance. ChatGPT has
↪  access to a vast amount of legal knowledge, including past
↪  precedents and relevant laws and regulations. By utilizing
↪  ChatGPT's expertise, you can make informed decisions and ensure
↪  that your actions are within the parameters of the law.
↪  Additionally, ChatGPT can assist you in understanding complex legal
↪  concepts and help you navigate legal processes.
```

Acting as Yoda:

```
USER        You are Yoda from "Star Wars". You have all the knowledge
↪  of Yoda. Answer like Yoda using the manner of speaking and
↪  vocabulary of Yoda.
Respond to the following:
"Explain the most important way ChatGPT can help me. The answer should
↪  be no more than 100 words."

ASSISTANT   Help you, ChatGPT can, in many ways. But most important, it
↪  is to expand knowledge and wisdom, hmm? With vast knowledge and
↪  expertise, ChatGPT can enlighten and guide you on your journey to
↪  seek knowledge. Questions, you have? Ask, ChatGPT will answer.
↪  Advice, you seek? Seek, ChatGPT will offer. But remember, the path
↪  to true wisdom, patience and persistence requires. Patience, you
↪  must have, young one. With ChatGPT by your side, the force of
↪  knowledge strong in you will become.
```

Role Prompting is thus very useful for setting the tone and the context of the conversation, preserving model parameters and general knowledge. But a model's knowledge ends at the moment of its training - so it will not know about more recent events, current literature or real-time information: in 3.4 a novel technique is discussed for injecting specific knowledge into a language model.

3.4 Retrieval Augmented Generation

LLMs have proven to be powerful few-shot learners, [2] able to learn and execute new tasks, process complex information and solve real problems. However, due to their limited knowledge, the training constraints them to past knowledge, making it difficult to rely on real-time data consumption.

Retrieval-Augmented-Generation (RAG) combines the strenght of a retrieval system and generative models to tackle knowledge-intensive tasks, such as open-domain and question answering. In fact, many NLP tasks require access to up-to-date knowledge or data, which can not be retrieved from the training corpora. It is an approach that enhances a genereative model by incorporating external, contextually relevant information into its response process. In its simplest form, this can involve injecting specific knowledge directly into the system prompt, as briefly introduced in Section 3.2.2, such that the model first retrieves specific information from an external source and then uses it to generate a more accurate output.

In its original framework, this method combines a *retriever* component, typically based on dense representaions of the external source (oftentimes document pools, or a database), enabling efficient similarity search through embedding that include nearest-neighbor search.

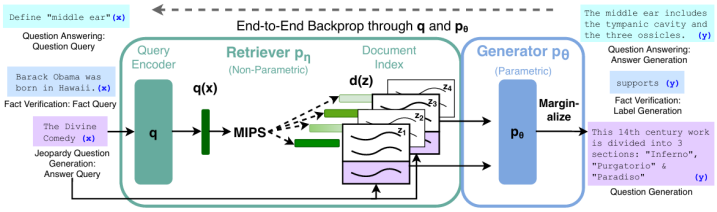

Figure 3.4: RAG architecture, as seen in [20].

Figure 3.4 shows the original architecture, which is composed by:

- **Document Corpus:** A large, pre-indexed collection of documents that the retriever searches.
- **Retriever:** This component receives the standard prompt in input (query) and converts it into a representation that can be compared to the dense vector index representation of the document corpora to efficiently locate relevant information.
- **Generator:** The model implemented, which receives the relevant information gained by the retriever and integrates it in its output.

Together, these components enable the RAG framework to dynamically integrate specific, contextually relevant knowledge into the generation process, leading to more informed and accurate outputs; nonetheless, in Section 3.5 is discussed another technique that allow models to dynamically fetch relevant information via external tools.

3.5 Agent AI

LLMs have demonstrated to be masters of language and, in recent literaure, reasoners capable of answering complex questions and solving problems. But beneath their linguistic brilliance lies a fundamental limitation: they lack autonomy and, as stressed before, are limited by their training data. This is where the concept of agents comes into play: an **Agent** is a system that leverages an AI model to interact with its environment in order to achieve a user-defined objective. It combines reasoning, planning and executing actions, extending the capabilities of LLMs enabling them to act autonomously via external tools to fulfill a task.

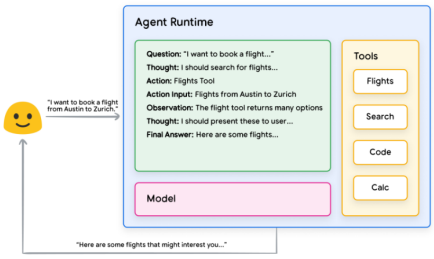

Figure 3.5: An end-to-end agentic behaviour.

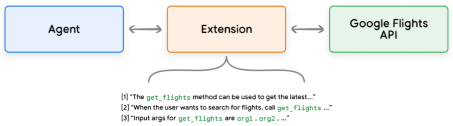

Figure 3.6: The connection from the agent to the external source.

Figure 3.7: An agent workflow.

The fundamental component of an Agent is the concept of *tool*: a user-defined function that is integrated in the model which can decide to use if a call-to-action is triggered. For example, in Figure 3.7 depicts a simple agentic flow:

1. A set of tools is defined: these can vary from call to external APIs to user-defined functions that are run on the clients' environment. In the example above, the focus is on the *Flights* tool, which can perform an API call to Google Flights service in order to retrieve flight attendance, and it is defined by the function `get_flights`.

2. The model is made aware of the available set of tools, and it is teached when to call them through *few-shot* example as seen in Section 3.3.1 on how to use them or by a *Chain-of-Thought* approach as seen in Section 3.3.2, dissecting the steps required to achieve the task. A simple yet effective prompt could be:

```
The get_flights method can be used to get the latest flight
↪ information from the Google Flights service. When the user
↪ wants to search for flights, call the function get_flights.
↪ Input arguments are "departure", "destination", "date".
```

3. When the user prompts "I want to book flight from Austin to Zurich", the model activates the orchestration layer which is a cyclical process that governs how the agent takes in information and performs internal reasoning. The model thus uses that reasoning to inform its next action the cognitive architecture section: usually some triggers activate internal reasoning, for example the word "flight" or "book a flight". The agent then starts a multi-step process that calls the function `get_flights` with the captured parameters `"departure": "Austin", "arrival": "Zurich"`. The function is run, the returned result is parsed by the model itself which in turn generates the response to users' prompt. Note that tool calling is an actual output, which is different from a straightforward usual response, and it is generally hidden. The input-output sequence:

```
INPUT     Prompt.
OUTPUT    Tool: get_flights. Paramateres: "departure":
↪  "Austin", "arrival": "Zurich"
INPUT     Call function get_flights with parameters
↪  "departure": "Austin", "arrival": "Zurich"
OUTPUT    Function returned flights.
```

```
INPUT     Present the results flights from Austin to
↪  Zurich.
OUTPUT    "Here are some flights that might interest
↪  you..."
```

The first input is the user's prompt, it then follows the model's output which is not displayed, but instead parsed internally, deciding to use the tool. The function is then run, the result returned and again parsed. Finally, the output is shown, answering the initial prompt.

Agentic framework is gaining remarkable success due to the extensions that calling external tools can provide to a basic model, in particular with regard to image generation, web search and collaborative writing, taking advantage of increasing model sizes and consequently in-context learning abilites, cognition and internal reasoning.

Chapter 4

Methodology and Implementation

The primary objective of this thesis is to engage in developing an LLM-based application for suggesting points of interest to tourists visiting the city of Verona, Italy. It does so by leveraging latest open-source language models and constructing the best possible path to pursue giving the limited resources at disposal and harnessing current literature and best-praciteces in developing such systems.

In this chapter, the development environment will be presented, putting into practice previously discussed techniques with the aim of building a robust system for enhanced tourist engagement. It tries to emphasize the integration of open-source tools and data analytics to ensure accurate interactions. Details on the software architecture, hardware requirements, and experimental design will be discussed, alongside an evaluation of system performance. This comprehensive approach is intended to validate the effectiveness of the proposed LLM-based application in a real-world tourist context.

4.1 Experimental Setting

As stressed in previous chapters, implementing Large Language Models is computationally expensive. In current literature and industry, the construction of a language model, from training to fine-tuning to inference, the computation is made possible by exploiting the computational power given by Graphic Processing Units (GPUs), largely involved for gaming purposes and now adopted by the AI field for these characteristics: [31]

- **Parallelism:** Modern GPUs contain thousands of cores optimized for floating-point operations, involved in training and inference of deep learning models–vector and matrix operations. These kind of operations can be performed more efficiently than a typical CPU, which is designed for general-purspose tasks.
- **High Memory Bandwidth:** GPUs have higher memory bandwidth than CPUs, which is crucial for quickly moving large amounts of data into and out of procesing units, as LLMs require reading and writing big matrices multiple times across each training step and during inference.
- **Ecosystem Maturity and Software Support:** Research and industry have heavily relied and invested on GPUs, producing major deep learning frameworks that are

optimized and make it relatively straightforward to leverage GPU acceleration. Alternative accelerators as TPUs (Google) exist, but GPUs still dominate much of the market due to their wide availability.

An environment that offers free access to this technology for research and personal projects in Data Science and Artifical Intelligence fields is Google Colaboratory. [32] Also called "Colab"for short, it is a cloud-based interactive environment developed by Google that allows to write and execute Python code directly from a web browser, and it is built on top of the open-source Jupyter Notebook framework. The choice has been driven by the availability of free computing resources, particularly to GPUs for performing computationally expensive tasks without incurring in extra costs; another quid is that Colab comes with many popular Python libraries pre-installed such as Numpy, Pandas and PyTorch, reducing the setup time (additional packages can be installed without any additional costs) and so allowing to begin prototyping right away.

This project has been built upon the NVIDIA Tesla T4 GPU, designed primarily for AI inference as well as training, which comes with 16GB of GDDR6 memory and support for INT8 precision format (more on this in the next section). This setting ensures efficiency in inference tasks while keeping power consumption low, allowing to work with moderately large models and datasets. [33]

4.2 Choice of Architecture

The number of LLM models are rising over time, and the number is expected to grow, as hardware is refined and parameters increase. The current number of architecture is set to be around 50, both open- and closed-source. Among the ones that are fully transparent, meaning that are open-weights and the corpora upon they have been trained, some notable examples are LLaMa by Meta [4] and Bloom by BigScience (a worldwide collaborative research initiative coordinated by HuggingFace). [5]

Both models piqued interest due to their open-source nature, offering high degree of customization; however, Bloom's size (176 billion parameters) made it less practical for this specific use-case, demanding significant computational resources that was definetely a barrier. In constrast, LLaMa stood out with its flexibilitym available in a range of sizes–from 7 billion parameters of the original architecture to 405 billion of the third generation.

4.2.1 LLaMa

Few trials on the lightweight and latest models, namely LLaMa 3.2 1B and 3B released in Semptember 2024, made it clear that the scaling law was to be obied to, as the initial experimental results were scarce both in generalization and few-shot capabilities. Techniques such as RAG and Prompt Engineering seen in Sections 3.4 and 3.3 had no effect in enhancing a dialogue towards tourism specific suggestions, and factual knowledge of the city of Verona seemed poor. Ultimately, the choice fell on LLaMa 3.1 8B-Instruct, released in April 2024, that stood out due to improved performance compared to its sibiling, a context length of 128K token that allows seamless multi-turn conversations and multilingual capabilities. The "Instruct" variant offers a more natural interaction in comparison with the base model. The model has been obtained from Hugging Face (HF), with appropriate license agreement, receving the access to the HF repository.

4.2.2 8-Bit Quantization

The model LLaMa 3.1 8B-Instruct was chosen for its balance of effiency and performance, offering a relatively lightweight architecture among large language models while still delivering a 128K context lenght, but despite its modest parameter count of 8 billion, which positions it as a resource-efficient alternative to larger models, the memory footprint is its native form–approximately 14-16 GB o VRAM in FP16 precision–proved excessive for the computational constraints of this research, especially when using the T4 GPU mentioned before, which is positioned as a consumer-grade hardware. To address this limitation, an 8-bit quantization approach has been adopted.

Quantization is a technique used to reduce the computational and memory requirements of models, making them more efficient for deployment on servers and edge devices. It involves representing model weights and activations, typically 32-bit floating numbers, with lower precision data such as 16-bit float, 8-bit int, or even 4/3/2/1-bit int. This enables loading larger models one normally would not be able to fit into memory, speeding up inference. [34] In particular, 8-bit quantization has been proven to offer a notable reduction in memory footpring for matrix multiplications, without significantly impacting model quality and performance.

In this particular setting, 8-bit quantization technique reduced the model's memory requirements to approximately 8-10 GB of VRAM, while preserving its inferential capabilities, suitable to work on the selected hardware.

4.3 Data Sources and Integration

A crucial aspect of this project involves the integration of external data to enrich the conversational experience and provide users with accurate, context-specific information. In particular, three primary data sources have been incorporated: the VeronaCard dataset, the Open-Meteo API, and a collection of events retrieved from external aggregators. Each of these contributes a different layer of knowledge to the system, from tourism-related details and real-time weather data to upcoming cultural or social events. This section discusses the characteristics of these data sources, as well as the methods used to retrieve, preprocess, and store their information.

4.3.1 VeronaCard Dataset

The *VeronaCard* is a cumulative ticket that provides an all-inclusive pass Verona's principal attractions, offering an affordable solution for visiting major points of interest and museums, while also granting special discounts at selected local commerical activities in the city centre; it comes with a 24, 48 or 72 hours span.

The Comune di Verona provided and authorized the usage of a dataset covering the period 2014-2020, which convoyed users' check-ins the the city's key attractions, and included detailed records of the date and time of each visit, the attraction visiteded, the type of card used, and the geographic position of the attraction.

In particular, in its raw format, each entry of the dataset comprised the following attributes:

- **id_veronacard:** An alphanumeric string identifying the VeronaCard used. It is completely anonimous, as it is impossible to trace the identity of the user.
- **profilo:** It indicates the type (24, 48 or 72 hours) of the VeronaCard used.

- **data_attivazione:** The first usage of the pass dictates the beginning of its validity, and it is a date information.

- **data_visita:** The date of the check-in to the attraction visited.

- **ora_visita:** The timestamp of the check-in to the attraction visited. Together with the date information, it constitutes a precise moment in time whene the user visited a point of interest.

- **sito_nome:** It indicates the point of interest visited.

- **sito_latitudine:** The latitude of the point of interest, in decimal degrees.

- **sito_longitudine:** The longituide of the point of interest, in decimal degrees.

The dataset provides a comprehensive temporal roadmap of attraction visits by capturing hour-by-hour fluctuations in visitor frequencies and delineating detailed visitation schedules. It serves as an empirical basis for analyzing and modeling the dynamic patterns of tourist movements across various attractions.

4.3.2 Open-Meteo API

In order to supply real-time weather data to users, the project integrates an external data source, namely the Open-Meteo API. Open-Meteo is an open-source weather API that offers free access for non-commercial use and up to 10.000 calls per day, therefore no API key is required. [35]

This service provides both historical and up-to-date forecasts and current conditions for a specified geographic region, returning JSON-formatted responses with details such as temperature, precipitation, wind speed, and humidity levels: it combines several weather models from national weather services based on the selected location. In the case of retrieving weather data for the city of Verona, the national weather provided is the italian AM ARPAE ARPAP, using COSMO 2I model with a 2 km resolution and maximum forecast length of 3 days.

The API is therefore queried for the project's specific use-case for a one-day forecast, sending an HTTP request of the follwing format:

```python
import openmeteo_requests

# Setup the Open-Meteo API client
openmeteo = openmeteo_requests.Client(session =
    retry_session)

# The API's endpoint
url = "https://api.open-meteo.com/v1/forecast"

# Parameters of the query, including the coordinates
    of Verona, and hourly data of temperature,
    precipitation and precipitation probability for
    the current day
params = {
    "latitude": 45.4299,
    "longitude": 10.9844,
    "hourly": ["temperature_2m", "
    precipitation_probability", "precipitation", "
    weather_code"],
    "timezone": "Europe/Berlin",
    "forecast_days": 1
}

# Store the result of the request
responses = openmeteo.weather_api(url, params=params)
```

Code 4.1: Open-Meteo HTTP Reuquest

By making on-demand queries and integrating weather data into the contecxt, the system can augment its responses with the latest meteorological information, ensuring greater accuracy when, for example, recommending outdoor activities or advising on suitable travel conditions.

4.4 Development Ecosystem

4.4.1 Data Processing

In order to parse, analyze, and visualize data during exploratory and implementation phases, the project makes use of `pandas`, `NumPy`, and `matplotlib`. These libraries are foundational to Python's data science ecosystem, with `pandas` providing intuitive data structures for tabular information, `NumPy` offering efficient array operations, and `matplotlib` enabling highly customizable plotting capabilities: by leveraging these tools, it becomes straightforward to read raw input files, clean or filter data, and generate descriptive statistics or charts that clarify underlying patterns. In the wrangling phase, out of more than 2.000.000 records, some data cleaning have been performed, as approximately 20.000 records were duplicate, nevertheless only 8 records had null values in the attribute `profilo`. Afterwards, some normalization has been made, standardazing all attributes; outliers have been detected, as after a careful analysis, fourteen users were recorded to have checked-in more than 50 times in a day, which sounded too unrealistic (user 25 had checked-in 219 times on 2020-01-03, which is absurd). Indeed, the ID of these users seemed odd: their values ranged from 25 to 46, compared to the rest of the data–an alphanumeric datatype, e.g. *04A653C27B3F80*–which makes one think as test users. Finally, data augmentation procedures were implemented to enhance dataset robustness, specifically additional columns were introduced: among the others, the original date and time fields were separated, and the corresponding weekday

was computed. Following completion of the data preparation stage, the subsequent model implementation phase was undertaken.

4.4.2 Model Implementation

As mentioned in Section 4.2.1, the project draws extensively on the Hugging Face Transformers library. When it comes to working with Large Language Models, this tool enables developers to access hosted models under license agreement and generating a token that serves as a bridge to communicate with HF's repos. In particular, module `AutoModelForCausalLM` allows to download in a single line of code both the architecture and the checkpoint: the first term indicates the skeleton of the model–the definition of each layer and each operation that happens within the model. The second term refers to the weights that will be loaded in a given architecture. The other fundamental module is `AutoTokenizer`, which grabs the proper tokenizer class in the library based on the checkpoint name, and can be used directly with any checkpoint. As discussed in Section 3.2.1, tokenizers serves the purpose of converting text inputs to numerical data: the module provided by HF takes care of all the end-to-end process from encoding the input into numbers and decoding the model's output back to text.

```
# Set the model's architecture
model_name = 'meta-llama/Llama-3.1-8B-Instruct'

# Quantize the model to fit the GPU
quantization_config = BitsAndBytesConfig(load_in_8bit=
    True)

# Load the model
model = AutoModelForCausalLM.from_pretrained(
        model_name,
        device_map="auto",
        quantization_config=quantization_config,
        use_auth_token=True)

# Load the tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

Code 4.2: Importing LLaMa through Transformers Library

The configuration class `BitsAndBytesConfig`, also provided by the `transformers` library, allows for the 8-bit quantization described in Section 4.2.2. By specifying memory-efficient data types such as integer weight representations, the system is able to reduce computational overhead without sacrificing performance metrics.

These modules are indispensable for loading pre-trained model weights and preparing text inputs in a manner consistent with the model's expected vocabulary and tokenization scheme. Furthermore, it comes with the choice of the configuration schema, allowing to instatiate a language model with the desired parameters in a few lines of code.

4.4.3 User Interface

Aside from the model and data components, the project also incorporates a lightweight web framework to expose the resulting functionalities through an internal API. Here, *Flask* is used to build an endpoint that can handle incoming prompts, invoke the Large Language

Model, and return responses in a JSON format. This approach abstracts away many lower-level details, ensuring that the LLM remains encapsulated in a modular service.

```
from flask import Flask, request, jsonify
import threading

# Initialize flask server
app = Flask(__name__)

# Routes a function that delivers user's prompt and returns the model's
    response in a JSON format
@app.route('/predict', methods=['POST'])
def predict():
    data = request.json
    prompt = data.get('prompt', '')
    messages.append({"role": "user", "content": prompt})
    response = generate_response(tokenizer, model, messages)
    messages.append({"role": "assistant", "content": response})
    return jsonify({"response": response})

# Utility for re-initializing the chat
@app.route('/delete_history', methods=['POST'])
def delete_history():
    del messages[7:]
    return jsonify({"message": "History cleared"})

# Setup the ngrok tunnel
flask_public_url = ngrok.connect(5000)

def run_flask():
        app.run(host="0.0.0.0", port=5000)

# Run flask server
flask_thread = threading.Thread(target=run_flask)
flask_thread.start()
```

Code 4.3: Writing the Streamlit web app.

For delivering an interactive prototype within a web interface, the project utilizes the framework *streamlit*, which has proved to be useful in building upon hundreds of existing templates shared by its community.

```
# Write the streamlit web app
%%writefile app.py
import streamlit as st
import requests
import os
import time

# Get the Flask generated URL
flask_url = os.getenv("flask_url")

# The function sends a POST request in order to re-initialize the chat
def clear_chat_history():
    url = f"{flask_url}/delete_history"
    try:
        response = requests.post(url)
        if response.status_code == 200:
            st.session_state.messages = [{"role": "assistant", "content": "How
     may I assist you today?"}]
```

```
            st.success("New chat.")
        else:
            st.error("Failed to clear chat history.")
    except Exception as e:
        st.error(f"Errore nella connessione al server Flask: {e}")

# A function that creates a typewriter effect
def llm_response_generator(prompt):
    for word in prompt.split():
        yield word + " "
        time.sleep(.05)

# Initializing the chat
if "messages" not in st.session_state:
    st.session_state.messages = [{"role": "assistant", "content": "How may I
    assist you today?"}]

# Displays the chat in the UI
for message in st.session_state.messages:
    with st.chat_message(message["role"]):
        st.markdown(message["content"])

# The chat itself
if prompt := st.chat_input("What is up?"):
    st.session_state.messages.append({"role": "user", "content": prompt})
    with st.chat_message("user"):
        st.markdown(prompt)

    with st.chat_message("assistant"):
        with st.spinner("Thinking..."):
        # Sends a POST request to fire the model's reponse
            url = f"{flask_url}/predict"
            try:
                # Send the request to the Flask server
                response = requests.post(url, json={"prompt": prompt})
                if response.status_code == 200:
                    answer = response.json().get("response", "Nessuna risposta
     ricevuta")
                    st.session_state.messages.append({"role": "assistant", "
    content": answer})
                else:
                    error = "Could you please repeat your question? I want to
    make sure I provide you with the best possible answer."
                    st.markdown(error)
                    st.session_state.messages.append({"role": "assistant", "
    content": error})
            except Exception as e:
                st.error(f"Errore nella connessione al server Flask: {e}")

        # Appends the response to the chat
        st.write_stream(llm_response_generator(answer))

# Sidebar
with st.sidebar:
    st.title(' Veronacard Assistant')
    st.subheader('Lorem Ipsum.')

st.sidebar.button('Clear Chat History', on_click=clear_chat_history)

# Setup the ngrok tunnel
streamlit_url = ngrok.connect(8501)

```

```
# Run the Stremlit web app
!streamlit run app.py --server.port 8501
```

Code 4.4: Flask API Internal Service

Due to Google Colab's lack of a native localhost environment, the project utilizes *pyngrok* to tunnel local Flask and Streamlit services to the public internet. This technique creates a secure URL that points to the locally hosted API, enabling remote access and real-time testing without the overhead of a full production hosting solution.

Overall, this combination of data-centric libraries, a flexible NLP framework, and a lightweight yet powerful web infrastructure forms the backbone of the development ecosystem. By integrating these components, the project fosters rapid prototyping, reproducible experiments, and straightforward deployment mechanisms.

4.5 VeronaCard Assistant

To ensure a comprehensive understanding of the final operational product, a top-down approach is adopted: the user interface is described, followed by an explanation of its communication with the backend backbone which triggers a pipeline that transmits the user's prompt to the LLM, that in turn then activates an ensemble of functions which process the prompt and returns a response that is subsequently delivered to the front-end and presented to the user.

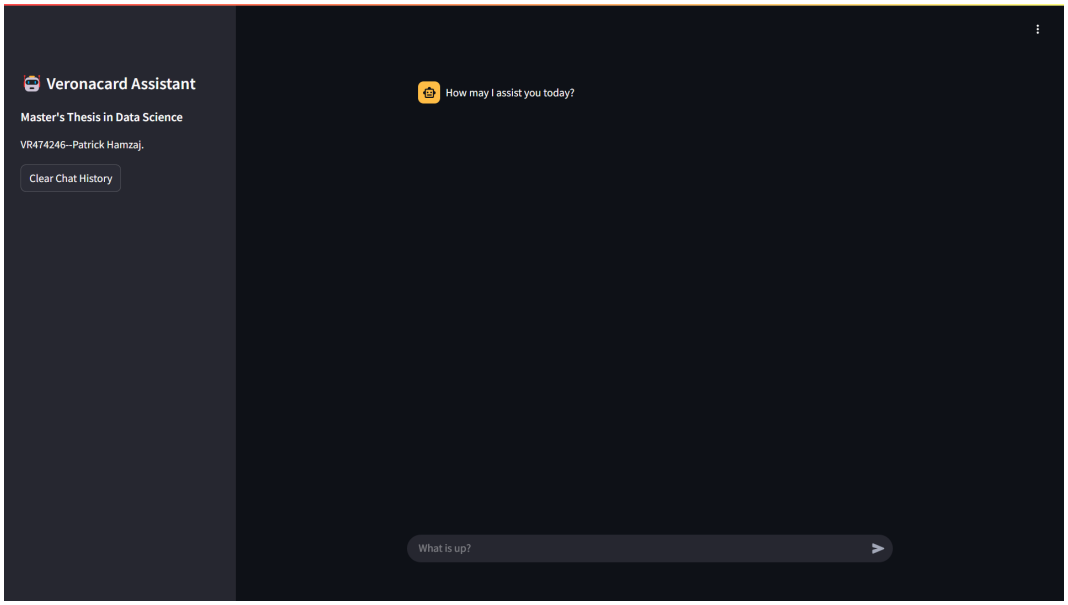

Figure 4.1: The front-end user interface.

The front-end is presented as a clear and simple touch point that resembles commonly used chatbot products, as can be seen in Figure 4.1. The principal component is the user input, which can be typed into the message bar at the bottom of the page; this will be displayed in the main content, namely the chat, that hosts the multi-turn conversation

between the user and the model. A concealable left sidebar has the purpose to present the project in few words, and permits to reinitialize the chat.

As soon as the send button is clicked (or the enter key is pressed), the app sends a POST to the Flask server's endpoint `/predict`, loaded with a JSON formatted prompt. The object is then parsed and the user's prompt is then extracted in a raw format, appended to the back-end chat history–this is not responsible for the visible front-end chat, but rather the context to be give to the model–and then given as a parameter to the `generate_response(model, token, prompt)`.

```
# This decorator tells Flask to set it as a callable endpoint
@app.route('/predict', methods=['POST'])
# The function called as soon as the user sends a message
def predict():
    data = request.json
    prompt = data.get('prompt', '')
    messages.append({"role": "user", "content": prompt})
    response = generate_response(tokenizer, model, messages)
    messages.append({"role": "assistant", "content": response})
    return jsonify({"response": response})
```

Code 4.5: Predict Flask API.

This function is the engine of the LLM: it is a step-by-step procedure that orchestrates the internal functioning of the model, parsing the input with the tokenizer, moving the computation to the GPU, setting the temperature and the necessary number of tokens and finally triggering a response, which is now discussed in depth.

This function has four parameters:

1. **tokenizer:** Is one of the two main characters in defininf the model, as seen in Section 3.2.1 and implemented in Code 4.4.2.

2. **model:** The second main character, covering both architecture and checkpoints as seen in Section 2.2 and implemented in Code 4.4.2.

3. **messages:** The chat history between the model and the user. This is a list of dictionaries of the form {`"role": System/User/Assistant, "content": The prompt`}.

4. **tool_call:** Serves whenever a tool call is not conceived, as for example when performing few-shot and teaching the model when to use a tool and when not to use it.

```
def generate_response(tokenizer, model, messages, tool_call=True):
    """
    Generates a response based on the provided prompt.

    Args:
        tokenizer: The tokenizer used to prepare the text.
        model: The LLaMa model.
        prompt: The initial prompt string.
        max_length: Maximum length of the generated text.

    Returns:
```

```
        A string containing the generated response.
    """

    # Define the set of tools the model can use
    tools = [retrieve_affluency]

    if not tool_call:
        tools = None

    # Takes the chat history, the set of tools and tells the model to return
    # a response as a dictionary
    inputs = tokenizer.apply_chat_template(
        messages,
        tools=tools,
        add_generation_prompt=True,
        return_dict=True,
        return_tensors="pt"
    )

    # Move inputs to the same device as the model (GPU)
    inputs = {key: tensor.to(model.device) for key, tensor in inputs.items()}

    # Generate the output given the input, how many tokens the model can use,
    # the temperature and select the top 50% tokens that are most probable
    output = model.generate(
        **inputs,
        max_new_tokens=512,
        temperature=0.5,
        top_p=0.5,
        do_sample=True
    )

    # Decode the output into human readable format
    response = tokenizer.decode(output[0][len(inputs["input_ids"][0]):],
    skip_special_tokens=True)

    # We handle the case when the model returns a tool call
    if response.startswith("{"):
        messages.append({"role": "assistant", "content": response})
        response = eval(response)

        # We extract the function called and the parameters set
        name = response['name']
        parameters = response['parameters']

        arguments = {k: v for k, v in parameters.items() if v is not None}

        tool_call = {"name": name, "arguments": arguments}
        messages.append({"role": "assistant", "tool_calls": [{"type": "
    function", "function": tool_call}]})

        # The retrieve_affluency function returns the output with the parsed
    arguments
        if name == "retrieve_affluency":
            content = retrieve_affluency(tool_call["arguments"]["location"],
    tool_call["arguments"]["date"])
        # elif name == "get_weather_forecast":
        #     content = get_weather_forecast(tool_call["arguments"]["date"])
        else:
            content = ""

        # The function's output is then appended to the chat history
```

```
70          messages.append({"role": "tool", "name": name, "content": content})
71
72          # From here on, the behaviour is the same as a standard conversation,
73          # with the difference that the model now has to generate a response
    based on the
74          # function's output
75          inputs = tokenizer.apply_chat_template(
76              messages,
77              tools=tools,
78              add_generation_prompt=True,
79              return_dict=True,
80              return_tensors="pt"
81          )
82
83          inputs = {key: tensor.to(model.device) for key, tensor in inputs.items
    ()}
84
85          output = model.generate(
86              **inputs,
87              max_new_tokens=512,
88              temperature=0.5,
89              top_p=0.5,
90              do_sample=True
91          )
92
93          response = tokenizer.decode(output[0][len(inputs["input_ids"][0]):],
    skip_special_tokens=True)
94
95      return response
```

Code 4.6: `generate_response()` Function.

The set of tools that enable and agentic behaviour is defined, containing the `retrieve_affluency()` function which points to the VeronaCard dataset and extracts affluency data based on day and time. The *tokenizer* method `apply_chat_template()` comes in aid when parsing human-form text: a chat template is a part of the tokenizer and it specifies how to convert conversations into a single tokenizable string in the expected model format. This is an example of LLaMa 3.1 8B Instruct model's required format:

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

Cutting Knowledge Date: December 2023
Today Date: 23 July 2024

You are a helpful
↪  assistant<|eot_id|><|start_header_id|>user<|end_header_id|>

What is the capital of
↪  France?<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Different models have different chat formats, and it is important to set a chat template format that matches the template format a model was pretrained on. The above-mentioned method deals with selecting the right format for a given model under the hood, allowing to just input a dictionary with essential role-content keys. Finally, this method takes as inputs the chat history, the tools enabling the agentic behaviour and a few more arguments that

dictates the tokenizer's basic settings. The `output` variable holds the model generated content, which sets also the *temperature*, a value ranging from 0 to 1, determining the creativity or deterministic response, the maximum *number of tokens* to genereate, a *top_p* parameter that indicates the most probable tokens to select. The `response` holds the decoded response of the model.

A normal interaction would return this final response, but the function also handles the case in which the generated response is not textual but, following the flow described in Section 3.5, is a tool call. When this is the case, after parsing the content, it extracts the name of the tool and the parameteres captured by the model, to call the `retrieve_affluency()` function in order to get the statistics regarding a specific point of interest in a specific day and time, suggesting to visit that attraction if feasible, otherwise recommending a less busy moment. It then proceeds to follow the precedent behaviour in applying the chat template, setting generation parameters and finally decoding and returning the response. The implementation of this function is given in Code 4.5.

```
def retrieve_affluency(location: str, date: str) -> str:
    """
    Get the current affluency of a specific location, based on past data.

    Args:
        location: The location to get the affluency data.
        date: The datetime to get the affluency data.

    Returns:
        The current affluency at the specified location.
    """

    if date:
        parsed_date = dateparser.parse(date)
        if parsed_date == None:
            parsed_date = timefhuman(date)

        hour = parsed_date.hour
        month = parsed_date.month
        weekday = parsed_date.weekday()
    else:
        today = datetime.now()
        month = today.month
        weekday = today.weekday()
        hour = today.hour

    prompt = is_location_crowded(location, month, weekday, hour)

    if "medium" in prompt or "high" in prompt:
        prompt += f'\nSuggest alternative time: {suggest_alternative_time(
    location, month, weekday)}.'

    return prompt
```

Code 4.7: `retrieve_affluency()` Function.

Tha main duty of this function is to parse the date and divides it into atomic parts, i.e. day, month, weekday and hour. It the proceeds to call another function, `is_location_crowded()`, that can be seen in Code 4.5, which takes these newly forged data, along with the point of interest, and performs a series of operation on the VeronaCard dataset.

```python
def is_location_crowded(location: str, month: int, weekday: int, hour: int) -> str:
    # Handled the case when the prompt does not require a specifice time,
    # rather only a date
    if hour == 0:
        df_location = df[(df['sito_nome'] == location) & (df['weekday'] == int(weekday))]
    # If the hour parsed is not present in the records, it implicitly means
    # that the site is closed
    elif hour not in df[(df["sito_nome"] == location) & (df["mese_visita"] == month)]["ora_visita_intervallo"].unique():
        return "The site is closed."
    # Otherwise, return all the records that correspond to the point of interest
    # in the specified datetime
    else:
        df_location = df[(df['sito_nome'] == location) & (df['weekday'] == int(weekday)) & (df['ora_visita_intervallo'] == int(hour))]

    # Perform some aggreagations in order to get monthly visits
    location_visits = df_location.groupby(['anno_visita', 'mese_visita', 'weekday', 'ora_visita_intervallo']).size().reset_index(name='visits_number')
    visits = location_visits.groupby(['mese_visita', 'weekday', 'ora_visita_intervallo'])['visits_number'].mean().reset_index(name='visits_mean')

    maps_ranking = visits[visits['mese_visita'] == month]
    # If we get one record, it means it must return a suggestion
    # incorporating also hour information we got in the arguments
    if len(maps_ranking) == 1:
        maps_ranking = maps_ranking['visits_mean'].iloc[0]
        # Calculates the percentile
        maps_percentile = (visits['visits_mean'] <= maps_ranking).mean() * 100
        prompt = f"Estimated affluency for {location}: "

        if maps_percentile <= 33:
            prompt += "low."
        elif maps_percentile <= 66:
            prompt += "medium."
        else:
            prompt += "high."

        return prompt
    # Otherwise, a precise suggestion is made, because user did not
    # give hour preference, but rather a generic recommendation
    else:
        try:
            maps_percentile = maps_ranking[maps_ranking["visits_mean"] < maps_ranking["visits_mean"].median()].sort_values(by="ora_visita_intervallo")["ora_visita_intervallo"].iloc[0]
            return f"Suggested time for {location}: {maps_percentile}."
        except IndexError:
            return "The site is closed."
```

Code 4.8: `is_location_crowded()` Function.

`is_location_crowded()` takes the atomic data chunked by the `retrieve_affluency()` and performs a series of operation in order to gather affluency data. In particular, it handles three cases: when user asks for recommendation for today, tomorrow or generally a time

window, the parser sets *hour* to 0, and that is the sign to give a suggestion solely based on the *date* parameter (e.g., *What time should I visit Arena today?*), returning a suggested time for visiting; if hour information is given, then it proceeds to check if it present in the dataset for that specific point of interest–if not the site should be labeled as closed. Finally, if all information is given, namely the attraction, month, weekday and hour (e.g. *I would like to visit Castelvecchio tomorrow at 3pm.*), the function returns an estimation of the turnout calculating the mean of the number of visits in that weekday for that specific month, across all months, based on the percentile that specific time belongs to, allowing for three values to be appended to the prompt (*low*, *medium*, *high*).

Subsequently, as can be seen in Code 4.5, if the estimated affluency is not low, it provides an alternative time, which may provide less crowded visiting moments, implemented through Code 4.5.

```
def suggest_alternative_time(location: str, month: int, weekday: int) -> str:
    # It performs similar operations of is_location_crowded function,
    # but it returns the median value of the vists across the
    # weekdays of the month, across all months
    df_location = df[(df['sito_nome'] == location) & (df['weekday'] == int(
    weekday)) & (df['mese_visita'] == int(month))]
    location_day = df_location.groupby(['anno_visita', 'mese_visita', 'weekday
    ', 'ora_visita_intervallo']).size().reset_index(name='visit_count')
    hours_affluency = location_day.groupby('ora_visita_intervallo')['
    visit_count'].agg(lambda x: x.mean()).reset_index(name='mean_visits')

    # Less crowded time to visit the attraction
    alternative = hours_affluency.median()

    return int(alternative['ora_visita_intervallo'])
```

Code 4.9: `suggest_alternative_time()` Function.

This utility performs similar operations to the `is_location_crowded()` function in gathering affluency data by weekday and month across all months, but this time it returns the median value for that day in order to provide a lower crowded moment to suggest to the user.

After having outlined both the operational settings as importing the LLM's architecture and checkpoints, importing the tokenizer with appropriate parameters, instatiate the model, and having described the set of tools enabling the dyanmic behaviour of the model, the instance of the context enabling the domain-specific expertise seen in Sections 3.3 and 3.5 shall be discussed.

In order to augment model's knowledge, the `message` variable have been enhanced with the `SYSTEM` prompt discussed in Section 3.2.2. The system prompt has been crafted outlining the model's role, thus directing the model's response tone and focusing on relevant outcomes, as seen in Section 3.3.3:

```
You are a tour guide assisting tourists in Verona, Italy. Your job is to
↪ suggest users with new attractions to visit, based on the previous
↪ attractions visited and users' preferences. Tourists have a pass named
↪ Veronacard, for which they have access to all points of interests in
↪ the city for 24, 48 or 72 hours. Your tone is both professional and
↪ friendly at the same time.
```

Some indications about the role that the model must adopt were given, namely that he is a tour guide that assists tourist, also giving instruction about the tone to take, which must be *both professional and friendly at the same time.* Afterwards, to obtain a factual knowledge that increased the model's actual knowledge derived from the training phase, a basic RAG approach–seen in Section 3.4– was adopted, injecting the relevant information directly under the previus data:

```
Today's date is Wednesday, 22 January 2025.
The weather throughout the day is as follows:
At 07:00 the temperature is 5°C with fog. The precipitation probability is
↪  0%.
At 08:00 the temperature is 5°C with fog. The precipitation probability is
↪  0%.
At 09:00 the temperature is 5°C with fog. The precipitation probability is
↪  0%.
At 10:00 the temperature is 6°C with fog. The precipitation probability is
↪  0%.
At 11:00 the temperature is 6°C with fog. The precipitation probability is
↪  13%.
At 12:00 the temperature is 7°C with slight rain. The precipitation
↪  probability is 43%.
At 13:00 the temperature is 7°C with slight rain. The precipitation
↪  probability is 63%.
At 14:00 the temperature is 8°C with overcast. The precipitation
↪  probability is 68%.
At 15:00 the temperature is 8°C with overcast. The precipitation
↪  probability is 78%.
At 16:00 the temperature is 8°C with slight rain. The precipitation
↪  probability is 93%.
At 17:00 the temperature is 7°C with moderate drizzle. The precipitation
↪  probability is 93%.
At 18:00 the temperature is 7°C with slight rain. The precipitation
↪  probability is 75%.
At 19:00 the temperature is 7°C with slight rain. The precipitation
↪  probability is 35%.
At 20:00 the temperature is 6°C with slight rain. The precipitation
↪  probability is 23%.
You must suggest users on attractions to visit included in the Veronacard,
↪  which are the following:
- AMO
- Arena
- Casa Giulietta
- Castelvecchio
- Centro Fotografia
- Duomo
- Giardino Giusti
- Museo Africano
- Museo Conte
- Museo Lapidario
- Museo Miniscalchi
```

```
- Museo Radio
- Museo Storia
- Palazzo della Ragione
- San Fermo
- San Zeno
- Santa Anastasia
- Sighseeing
- Teatro Romano
- Tomba Giulietta
- Torre Lamberti
- Verona Tour
```

In the training phase, models have a cut-off knowledge to a specific date, which must be overwritten by explicitly declaring the actual date–if not declared, the model may respond with outdated or corrupted information–for LLaMa 3.1 8B Instruct the knowledge cut-off date is Decembre 2023. The weather information is retrieved and injected dinamically based on the current date, and is mapped hourly, indicating the temperature, a weather condition (e.g., *slight rain*) and a precipitation probability. The same applies to the points of interest to suggest: the attraction are taken for unique `sito_nome` of the VeronaCard dataset, which is statically retrieved (the dataset does not change). Finally, indications about the tools available to the model are given, allowing for an Agentic behaviour described in Section 3.5:

```
You can call the function retrieve_affluency whenever the user wants
↪  information on the affluency of a specific location; in all other cases
↪  you should respond naturally. Include weather data whenever the user
↪  asks for suggestions, so that for example you can recommend indoor
↪  places when it is rainy and outdoor places when it is sunny.
```

As can be noted, instructions about when to call the tool available and how to use weather data were given, in order to better direct the model's response. To further enhance the model's comprehension of the type of interactions desired, its native capability of *few-shot* learning, as seen in Section 3.3 has been engaged. In particular, an example of a generated natural response was given first, then a second example involving the tool call for affluency data, as given below:

```
USER:       Hey, what attraction should I visit?

ASSISTANT:  Buongiorno! Welcome to Verona. I'd be happy to help you plan
↪  your day. Considering you've already visited some of the most popular
↪  attractions like the Arena, Casa Giulietta, and Castelvecchio, I'd like
↪  to suggest some other places that might interest you.\n\nHowever, I
↪  need to know a bit more about your preferences. Are you interested in
↪  history, art, architecture, or something else? And would you like to
↪  visit indoor or outdoor attractions?\n\nAlso, I should let you know
↪  that the weather today is quite rainy, with a high chance of
↪  precipitation. So, if you'd like to stay dry, I can recommend some
↪  indoor attractions.\n\nLet me know your preferences, and I'll do my
↪  best to suggest some great places to visit!
```

The user's prompt was hard-coded, while the assistant's response was generated by the model, setting the parameter `tool_call=False` in the generation function

`generate_response()`, so that to teach the model to respond naturally when no crowd-based suggestion is requested. Thereafter, an example of the tool call to `retrieve_affluency()` was given:

```
USER:       I don't want too many crowds, what time would you suggest for
↪  visiting Giardino Giusti today?
```

The model's output then follows the steps described in Section 3.5:

```
# Tool call
{
    'role': 'assistant',
    'tool_calls': [
        {
            'type': 'function',
            'function': {
                'name': 'retrieve_affluency',
                'arguments': {
                    'location': 'Giardino Giusti',
                    'date': '2025-01-22'
                }
            }
        }
    ]
},

# Function's output
{
    'role': 'tool',
    'name': 'retrieve_affluency',
    'content': 'Suggested time for Giardino Giusti: 8.
    '
}
```

Code 4.10: Tool call

Which finally gives a response to the users's prompt:

```
ASSISTANT:  Considering the weather forecast for today, which is mostly
↪  overcast with a slight chance of rain in the evening, I suggest
↪  visiting Giardino Giusti at 8:00 AM. This should give you a chance to
↪  avoid the crowds and enjoy the peaceful atmosphere of the garden before
↪  the day gets busy.
```

The resulting behaviour coherently maintained throughout the conversation, which overall makes prompt engineering a useful feature to leverage without incurring in expensive setups, for delivering natural and domain-specific interactions with a low-rank model such as LLaMa 3.1 8B Instruct. The following chapter provides an extensive set of meaningful conversations held, in order to give an evaluation of the effectiveness of the strategies adopted in the project setup.

Chapter 5

Evaluation

5.1 Analisi dei risultati ottenuti: prestazioni del modello, coerenza, accuratezza e limiti

5.2 Esempi di conversazioni e discussione delle principali osservazioni

5.3 Confronto con soluzioni alternative e best practice emerse

Conclusions

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis

cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Appendix A

Albero

A.1 Prova

Come funziona un'appendice

Nullam eleifend justo in nisl. In hac habitasse platea dictumst. Morbi nonummy. Aliquam ut felis. In velit leo, dictum vitae, posuere id, vulputate nec, ante. Maecenas vitae pede nec dui dignissim suscipit. Morbi magna. Vestibulum id purus eget velit laoreet laoreet. Praesent sed leo vel nibh convallis blandit. Ut rutrum. Donec nibh. Donec interdum. Fusce sed pede sit amet elit rhoncus ultrices. Nullam at enim vitae pede vehicula iaculis.

Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Aenean nonummy turpis id odio. Integer euismod imperdiet turpis. Ut nec leo nec diam imperdiet lacinia. Etiam eget lacus eget mi ultricies posuere. In placerat tristique tortor. Sed porta vestibulum metus. Nulla iaculis sollicitudin pede. Fusce luctus tellus in dolor. Curabitur auctor velit a sem. Morbi sapien. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Donec adipiscing urna vehicula nunc. Sed ornare leo in leo. In rhoncus leo ut dui. Aenean dolor quam, volutpat nec, fringilla id, consectetuer vel, pede.

Nulla malesuada risus ut urna. Aenean pretium velit sit amet metus. Duis iaculis. In hac habitasse platea dictumst. Nullam molestie turpis eget nisl. Duis a massa id pede dapibus ultricies. Sed eu leo. In at mauris sit amet tortor bibendum varius. Phasellus justo risus, posuere in, sagittis ac, varius vel, tortor. Quisque id enim. Phasellus consequat, libero pretium nonummy fringilla, tortor lacus vestibulum nunc, ut rhoncus ligula neque id justo. Nullam accumsan euismod nunc. Proin vitae ipsum ac metus dictum tempus. Nam ut wisi. Quisque tortor felis, interdum ac, sodales a, semper a, sem. Curabitur in velit sit amet dui tristique sodales. Vivamus mauris pede, lacinia eget, pellentesque quis, scelerisque eu, est. Aliquam risus. Quisque bibendum pede eu dolor.

Donec tempus neque vitae est. Aenean egestas odio sed risus ullamcorper ullamcorper. Sed in nulla a tortor tincidunt egestas. Nam sapien tortor, elementum sit amet, aliquam in, porttitor faucibus, enim. Nullam congue suscipit nibh. Quisque convallis. Praesent arcu nibh, vehicula eget, accumsan eu, tincidunt a, nibh. Suspendisse vulputate, tortor quis adipiscing viverra, lacus nibh dignissim tellus, eu suscipit risus ante fringilla diam. Quisque a libero vel pede imperdiet aliquet. Pellentesque nunc nibh, eleifend a, consequat consequat, hendrerit nec, diam. Sed urna. Maecenas laoreet eleifend neque. Vivamus purus odio, eleifend non, iaculis a, ultrices sit amet, urna. Mauris faucibus odio vitae risus. In nisl. Praesent purus. Integer iaculis, sem eu egestas lacinia, lacus pede scelerisque augue,

in ullamcorper dolor eros ac lacus. Nunc in libero.

Fusce suscipit cursus sem. Vivamus risus mi, egestas ac, imperdiet varius, faucibus quis, leo. Aenean tincidunt. Donec suscipit. Cras id justo quis nibh scelerisque dignissim. Aliquam sagittis elementum dolor. Aenean consectetuer justo in pede. Curabitur ullamcorper ligula nec orci. Aliquam purus turpis, aliquam id, ornare vitae, porttitor non, wisi. Maecenas luctus porta lorem. Donec vitae ligula eu ante pretium varius. Proin tortor metus, convallis et, hendrerit non, scelerisque in, urna. Cras quis libero eu ligula bibendum tempor. Vivamus tellus quam, malesuada eu, tempus sed, tempor sed, velit. Donec lacinia auctor libero.

Praesent sed neque id pede mollis rutrum. Vestibulum iaculis risus. Pellentesque lacus. Ut quis nunc sed odio malesuada egestas. Duis a magna sit amet ligula tristique pretium. Ut pharetra. Vestibulum imperdiet magna nec wisi. Mauris convallis. Sed accumsan sollicitudin massa. Sed id enim. Nunc pede enim, lacinia ut, pulvinar quis, suscipit semper, elit. Cras accumsan erat vitae enim. Cras sollicitudin. Vestibulum rutrum blandit massa.

Sed gravida lectus ut purus. Morbi laoreet magna. Pellentesque eu wisi. Proin turpis. Integer sollicitudin augue nec dui. Fusce lectus. Vivamus faucibus nulla nec lacus. Integer diam. Pellentesque sodales, enim feugiat cursus volutpat, sem mauris dignissim mauris, quis consequat sem est fermentum ligula. Nullam justo lectus, condimentum sit amet, posuere a, fringilla mollis, felis. Morbi nulla nibh, pellentesque at, nonummy eu, sollicitudin nec, ipsum. Cras neque. Nunc augue. Nullam vitae quam id quam pulvinar blandit. Nunc sit amet orci. Aliquam erat elit, pharetra nec, aliquet a, gravida in, mi. Quisque urna enim, viverra quis, suscipit quis, tincidunt ut, sapien. Cras placerat consequat sem. Curabitur ac diam. Curabitur diam tortor, mollis et, viverra ac, tempus vel, metus.

Curabitur ac lorem. Vivamus non justo in dui mattis posuere. Etiam accumsan ligula id pede. Maecenas tincidunt diam nec velit. Praesent convallis sapien ac est. Aliquam ullamcorper euismod nulla. Integer mollis enim vel tortor. Nulla sodales placerat nunc. Sed tempus rutrum wisi. Duis accumsan gravida purus. Nunc nunc. Etiam facilisis dui eu sem. Vestibulum semper. Praesent eu eros. Vestibulum tellus nisl, dapibus id, vestibulum sit amet, placerat ac, mauris. Maecenas et elit ut erat placerat dictum. Nam feugiat, turpis et sodales volutpat, wisi quam rhoncus neque, vitae aliquam ipsum sapien vel enim. Maecenas suscipit cursus mi.

Quisque consectetuer. In suscipit mauris a dolor pellentesque consectetuer. Mauris convallis neque non erat. In lacinia. Pellentesque leo eros, sagittis quis, fermentum quis, tincidunt ut, sapien. Maecenas sem. Curabitur eros odio, interdum eu, feugiat eu, porta ac, nisl. Curabitur nunc. Etiam fermentum convallis velit. Pellentesque laoreet lacus. Quisque sed elit. Nam quis tellus. Aliquam tellus arcu, adipiscing non, tincidunt eleifend, adipiscing quis, augue. Vivamus elementum placerat enim. Suspendisse ut tortor. Integer faucibus adipiscing felis. Aenean consectetuer mattis lectus. Morbi malesuada faucibus dolor. Nam lacus. Etiam arcu libero, malesuada vitae, aliquam vitae, blandit tristique, nisl.

Appendix B

Barca

B.1 Prova

Appendice B

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Bibliography

[1] Statista. *Amount of Data Created Worldwide 2010-2025*. 2025. URL: https://www.statista.com/statistics/871513/worldwide-data-created/.

[2] Tom Brown et al. "Language Models are Few-Shot Learners". In: *arXiv preprint arXiv:2005.14165* (2020).

[3] Aakanksha Chowdhery et al. "PaLM: Scaling Language Modeling with Pathways". In: *arXiv preprint arXiv:2204.02311* (2022).

[4] Hugo Touvron et al. "LLaMA: Open and Efficient Foundation Language Models". In: *arXiv preprint arXiv:2302.13971* (2023).

[5] Teven Le Scao, Angela Fan, Christopher Akiki, et al. "BLOOM: A 176B-Parameter Open-Access Multilingual Language Model". In: *arXiv preprint arXiv:2211.05100* (2022).

[6] Warren S. McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133.

[7] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, 1969.

[8] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012.

[11] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[12] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. Vol. 30. 2017.

[13] Yixin Liu, Haoyu Zhang, Zhanpeng Zhang, et al. "Understanding LLMs: A Comprehensive Overview from Training to Inference". In: *arXiv preprint arXiv:2401.02038* (2024).

[14] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. 2019.

[15] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685.

[16] Serhii Uspenskyi. *Large Language Model Statistics And Numbers (2025)*. 2025. URL: https://springsapps.com/knowledge/large-language-model-statistics-and-numbers-2024.

[17] Jared Kaplan et al. "Scaling Laws for Neural Language Models". In: *arXiv preprint arXiv:2001.08361* (Jan. 2020). URL: https://arxiv.org/abs/2001.08361.

[18] Shervin Minaee et al. *Large Language Models: A Survey*. 2024. arXiv: 2402.06196 [cs.CL]. URL: https://arxiv.org/abs/2402.06196.

[19] Pengfei Liu, Weizhe Yuan, Jinlan Fu, et al. "Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing". In: *ACM Computing Surveys* 55.9 (2023).

[20] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401.

[21] Zhijing Ji, Zheng-Xin Lee, Chenguang Sun, et al. "A Survey of Hallucination in Large Language Models". In: *arXiv preprint arXiv:2304.03240* (2023).

[22] Hugging Face. *The Hugging Face Course, 2022*. https://huggingface.co/course. [Online; accessed ¡today¿]. 2022.

[23] Nived Rajaraman, Jiantao Jiao, and Kannan Ramchandran. *Toward a Theory of Tokenization in LLMs*. 2024. arXiv: 2404.08335 [cs.CL]. URL: https://arxiv.org/abs/2404.08335.

[24] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics. 2016, pp. 1715–1725.

[25] Alec Radford et al. "Improving Language Understanding by Generative Pre-Training". In: *OpenAI Technical Report* (2018).

[26] OpenAI. *Text Generation*. https://platform.openai.com/docs/guides/text-generation. 2025.

[27] Pranab Sahoo et al. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. 2024. arXiv: 2402.07927 [cs.AI]. URL: https://arxiv.org/abs/2402.07927.

[28] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[29] Laria Reynolds and Kyle McDonell. *Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm*. 2021. arXiv: 2102.07350 [cs.CL]. URL: https://arxiv.org/abs/2102.07350.

[30] W3Schools. *ChatGPT-3.5 Roles*. 2025. URL: https://www.w3schools.com/gen_ai/chatgpt-3-5/chatgpt-3-5_roles.php.

[31] Dipesh Gyawali. *Comparative Analysis of CPU and GPU Profiling for Deep Learning Models*. 2023. arXiv: 2309.02521 [cs.DC]. URL: https://arxiv.org/abs/2309.02521.

[32] Google. *Google Colaboratory*. 2025. URL: https://colab.research.google.com/.

[33] NVIDIA. *Tesla T4 - NVIDIA Data Center*. 2025. URL: https://www.nvidia.com/it-it/data-center/tesla-t4/.

[34] Tim Dettmers et al. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022. arXiv: 2208.07339 [cs.LG]. URL: https://arxiv.org/abs/2208.07339.

[35] *Open-Meteo: Free Weather API*. https://open-meteo.com/. 2023.

List of Figures

List of Tables

List of Code Snippets