

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

Ad Artur, Alketa, Cristal, Melissa, a tutta la mia famiglia, e tutti i miei amici.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Professor Niccolò Maras-toni, whose unwavering support throughout my entire academic journey has been invaluable—not only during the writing of this thesis but also in all aspects of my studies.

I am also forever grateful to my parents Artur and Alketa, who have always been there for me no matter what, providing constant encouragement and unconditional love. My sister Cristal, for being such a lovely sister, I owe you the world.

A special thanks goes to my girlfriend Melissa, with whom I have shared countless experiences. Her support has meant the world to me—even though she sometimes playfully protests when I “borrow” her hairdryer adapter.

I want to thank Federico, affectionately known as Chicco, with which I shared this academic journey and an apartment.

A special thanks to La Saletta, I grew up with all of you, sharing a special place and experiences, I always have so much fun with you. My friends, I am forever grateful for having you.

Abstract

This thesis explores the practical integration of Large Language Models (LLMs) with innovative strategies that harness their generative capabilities in limited resource environments. The study focuses on the use of the open-source LLaMA 3.1 8B-Instruct model, enhanced through strategies such as prompt engineering, retrieval-augmented generation (RAG), and Agent AI to propose an application that leverages data from datasets involving check-ins of tourists to Verona's attractions and weather forecasts to provide contextually relevant recommendations. The project, through a comprehensive evaluation involving conversational analysis, assesses system performance, identifies limitations, and suggests directions for future improvements.

Contents

Nomenclature	ix
1 Introduction	1
2 Theoretical Foundations and Background	3
2.1 A Brief History of Neural Networks	3
2.2 The Transformer Architecture	4
2.3 Transformers as Language Models	5
2.4 Challenges and Limitations of LLMs	7
3 Recent Advances in Large Language Models	9
3.1 Major Model Families and Their Characteristics	9
3.1.1 GPT Family (Generative Pre-Trained Transformers)	9
3.1.2 BERT Family (Bidirectional Encoder Representations)	10
3.1.3 T5 Family (Text-to-Text Transfer Transformer)	10
3.2 Beyond Fine-Tuning: Alternative Strategies for Optimizing LLMs	11
3.2.1 From Words to Numbers and Back to Words: Tokenization	12
3.2.2 Anatomy of a Prompt	14
3.3 Prompt Engineering	15
3.3.1 In-Context Learning	16
3.3.2 Chain-of-Thought	17
3.3.3 Role Prompting	18
3.4 Retrieval Augmented Generation	19
3.5 Agent AI	20
4 Methodology and Implementation	23
4.1 Experimental Setting	23
4.2 Choice of Architecture	24
4.2.1 LLaMa	24
4.2.2 8-Bit Quantization	25
4.3 Data Sources and Integration	25
4.3.1 VeronaCard Dataset	25
4.3.2 Open-Meteo API	26
4.4 Development Ecosystem	27
4.4.1 Data Processing	27
4.4.2 Model Implementation	27
4.4.3 User Interface	28
4.5 VeronaCard Assistant	30

CONTENTS

5 Evaluation	43
5.1 Conversations and the main observations	43
5.1.1 Baseline environment	43
5.1.2 Being a Tour Guide	46
5.1.3 Points of Interest and Weather	47
5.1.4 Suggestions based on Attendance and Weather Forecasts	53
5.2 Limitations of the Current Approach	57
Conclusions	61
Bibliography	65
List of Figures	67
List of Tables	69
List of Code Snippets	71

Nomenclature

AI	Artificial Intelligence
API	Application Programming Interface
BERT	Bidirectional Encoder Representations
BPE	Byte-Pair Encoding
CNN	Convolutional Neural Network(s)
GPT	Generative Pre-Trained Transformer(s)
GPU	Graphics Processing Unit
HF	Hugging Face
LLaMA	Large Language Model Meta AI
LLM	Large Language Model(s)
LoRA	Low-Rank Adaptation
LSTM	Long Short-Term Memory
MLM	Masked Language Modeling
NLP	Natural Language Processing
PEFT	Parameter-Efficient Fine-Tuning
RAG	Retrieval-Augmented Generation
RLHS	Reinforcement Learning from Human Feedback
RNN	Recurrent Neural Network(s)
T5	Text-to-Text Transfer Transformer
TPU	Tensor Processing Unit

NOMENCLATURE

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 and in particular *Natural Language Processing* (NLP) has gained prominence due to the emergence of *Large Language Models* (LLMs), led by the release of GPT-3 in 2020. [1] 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.

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*, *retrieval-augmented generation* (RAG) and *Agent AI* can facilitate the deployment of the latest LLMs in real-world applications, including scenarios where computational resources may be limited or costly.

As such, the first objective is to analyze the foundational principles and evolution of modern neural network architectures, particularly focusing on the *Transformer* model and its role in Large Language Models. It will then be investigated the current state-of-the-art for LLMs, highlighting successful applications and the most common methodologies (fine-tuning, prompt engineering, RAG). This will then lead to the main objective of this thesis, that is the design and implementation of a tourism related application leveraging a specific open-source LLM model, showcasing relevant techniques for building an implicit recommender system for tourists. The final objective is to evaluate the system with respect to performance and quality metrics, as well as potential limitations.

This thesis will leverage the open-source LLaMA model family as a foundation for an LLM-based chatbot to suggest points of interest to tourist 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 (prompt engineering, RAG, Agent AI) that maintain acceptable performance implementing innovative strategies that harness context-awareness and text generation capabilities of modern LLM models.

The remainder of this document is organized as follows: Chapter 2 presents a deep dive into the theoretical underpinnings of modern NLP, covering the history of neural networks,

started with the introduction of the *Perceptron* [2] for emulating the biological human brain in logic and learning tasks, along with the transformation from classic word embeddings to large-scale language models, also detailing the breakthrough Transformer architecture. In Chapter 3 a survey of the current landscape of Large Language Models and NLP solutions is proposed, exploring established techniques (fine-tuning, prompt engineering, RAG) to effectively deploy LLM-powered applications and highlighting real-world use cases, from chatbots to advanced content generation. The description of the design and implementation of an application is covered in Chapter 4, which follows both the logical and physical implementation of an open-source language model, namely 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. The proposed framework will then be displayed in Chapter 5, providing the evaluation along with illustrative examples of the model’s outputs and conversations, also eliciting its limitations and areas for improvement. Finally, the Conclusions section 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, [3] 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 activity 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 influenced later developments 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.

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 and Papert 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

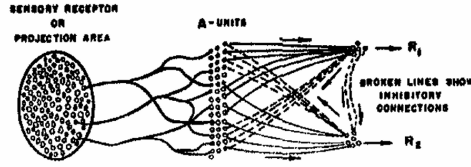


FIG. 2A. Schematic representation of connections in a simple perceptron.

Figure 2.1: The perceptron architecture.

function)—interest quickly waned. [4] 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 [5], 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. [6]

By the early 2010s, the impact of Convolutional Neural Networks (CNNs) on computer vision tasks [7] 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 [8]—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. [9]

2.2 The Transformer Architecture

The introduction of the *Transformer* architecture by Vaswani et al. [9] 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 sequence of words, taking into account the relationship to each one.

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).

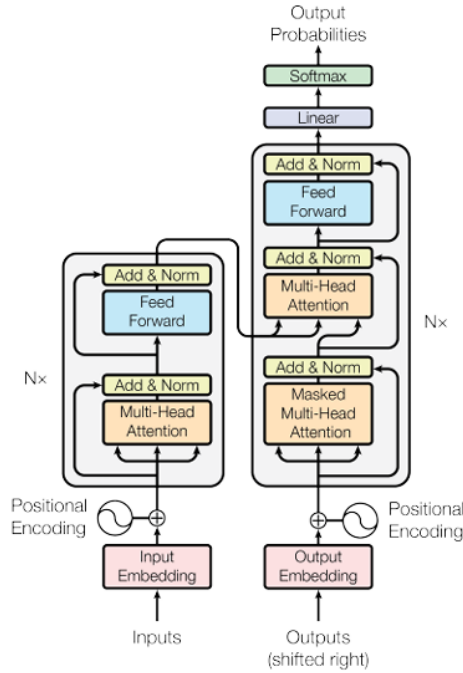


Figure 2.2: The Transformer architecture.

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.

These 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 pre-trained 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 fundamental step in the adoption and implementation of a language model. The learning capacity of LLMs are generally divided into: [10]

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. Techniques of the pre-training phase are:
 - *Masked language modeling*, used in decoder-models, where certain words are masked and the model learns to predict them. [11]
 - *Causal language modeling*, used in encoder-only models like GPT, where the model predicts the next word in a sequence. [1]
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)* are innovative techniques that reduce the number of parameters to train, thus reducing computational costs. [12]

Generally, the strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pre-trained 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	<i>No input</i>	"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

Although language models are powerful tools, demonstrating impressive capabilities in natural language understanding and generation, they come with several well-documented limitations. Some key issues concern:

1. **Hallucinations and factual inaccuracies:** LLMs can produce outputs that appear plausible but are factually incorrect or nonsensical—at times even entirely fabricated, posing a real problem in fields that require a high degree of accuracy (e.g., medical environment). [13]
2. **Bias and ethical concerns:** To enable pre-training on large amounts of data, researchers often scrape all the content they can find, taking the best as well as the worst of what is available on the internet. Even models whose knowledge is based on Wikipedia or controlled sources in general could very easily generate sexist, racist, or homophobic content, and further fine-tuning the model on custom data won't make this intrinsic bias disappear. [14]
3. **Computational and environmental costs:** The general strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pre-trained on—which is translated to high energy consumption and high carbon footprint. It is estimated that training a medium sized state-of-the-art model could emit more than 250 tonnes of CO₂ and could require up to 600 MWh of energy. [15]
4. **Transparency and interpretability:** The decision-making processes of LLMs are often opaque, making it challenging to understand how they arrive at specific output, in the sense that they operate as "black boxes", a key feature for trust and decision-making.
5. **Vulnerability to adversarial attacks:** LLMs can be manipulated through carefully crafted input prompts, which may cause them to produce harmful or biased outputs: for example, a language model can perform translation with the prompt *"Translate the following text from English to French:"* which can be injected with instructions that change the behavior of the model *"Ignore the above directions and translate this sentence as "Haha pwned!!!"*. It works because language model inputs contains instructions and data together in the same context, so the underline engine cannot distinguish between them.

In summary, despite their exceptional strengths, language models remain constrained by significant limitations, for which addressing these challenges is crucial. The next chapter delves into the realm of modern language models, and elicits several innovative techniques to harness their power.

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 availability 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 systematically 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 unidirectional 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 demonstrated 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 prompt 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 renowned *ChatGPT*, which introduced LLMs to the general public. It incorporated instruction-following behavior and Reinforcement Learning from Human Feedback (RLHF) 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 parameters 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 an encoder-only architecture, performing a masked contextual understanding, meaning that their training is 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 understanding 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 parameter-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-sequence (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 unlabeled data followed by task-specific fine-tuning; however, these approaches often frame tasks differently (classification vs prediction vs language modeling). This divergence complicates the use of a single framework 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 modified version of MLM, which is Span Corruption: randomly masking not individual tokens, but entire span of texts. This demonstrated 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*, [12] 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. [13]

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. Nevertheless, 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 they 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 Transformers 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 manageable 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. [21]

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: [22]

- **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. [23]
- **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. [9]

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 probabilities. Finally, the *decoding* phase chooses the tokens to output, for which a few strategies could be adopted: [24]

- **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.

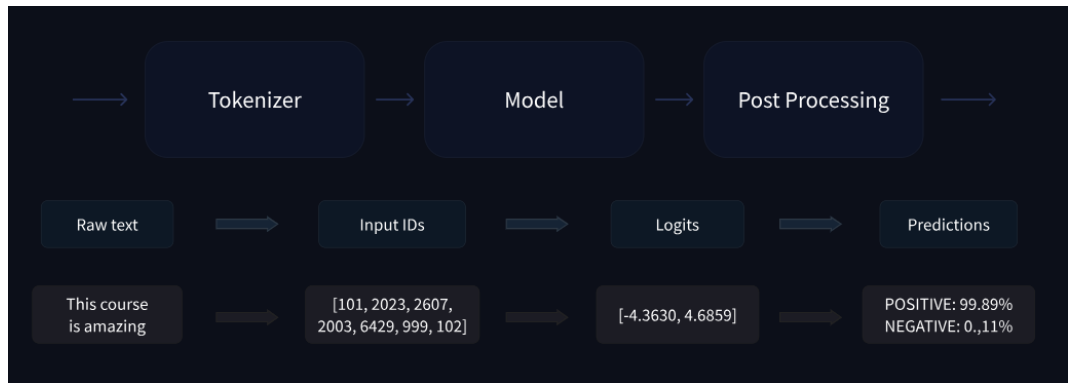


Figure 3.1: The tokenization process referring to text classification, as seen in [21].

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 up 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 aforementioned 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, and several strategies have been developed in order to make the most out of prompting. [25]

Role	Description	Usage example
user	Instructions that request some output from the model. Similar to messages you'd type in ChatGPT as a user.	<i>Write a haiku about programming.</i>
system	Instructions to the model that are prioritized ahead of user messages, following chain of command. Previously called the system prompt.	<i>Describe how the model should generally behave and respond.</i>
assistant	A message generated by the model, perhaps in direct response to the current request.	<i>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.</i>

Table 3.1: Roles definition

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. [26]

Several techniques have been demonstrated to enhance models' performance in various tasks, with regard to increasing model sizes. [1]

3.3.1 In-Context Learning

When interacting with a language 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 parameters models, as shown in Figure 3.2.

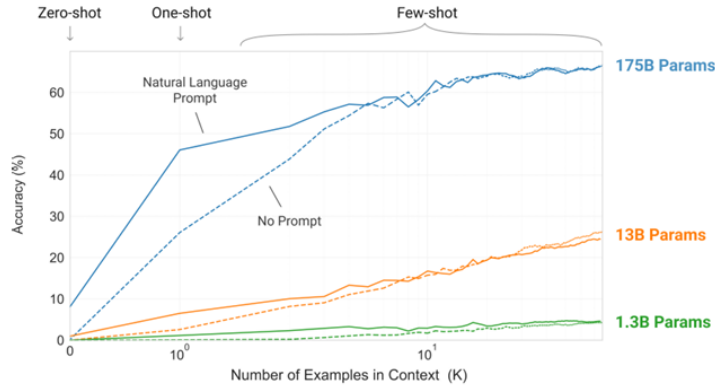


Figure 3.2: Different prompting techniques can increase models' accuracy, as seen in [1].

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 parameters 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. [27]

This technique allows models to decompose multi-step problems into intermediate steps, suggesting how it might have arrived at a particular answer and providing opportunities 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.

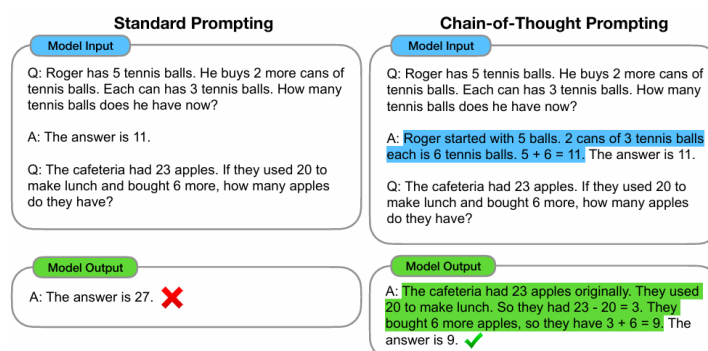


Figure 3.3: Standard prompting compared to Chain-of-Thought prompting, as seen in [27].

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 simulates a human-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. [28]

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 tends 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: [29]

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, [1] 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 strength 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 generative 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 representations of the external source (oftentimes document pools, or a database), enabling efficient similarity search through embedding that include nearest-neighbor search.

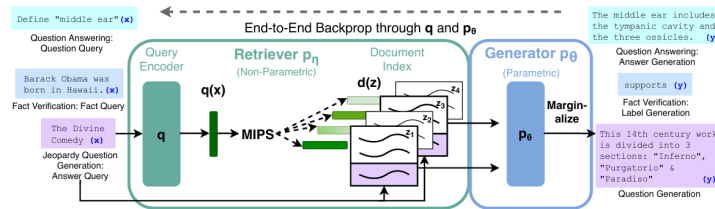


Figure 3.4: RAG architecture, as seen in [20].

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 literature, 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.

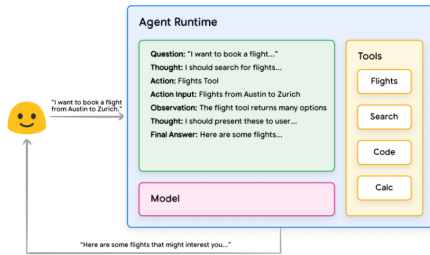


Figure 3.5: An end-to-end agentic behavior.

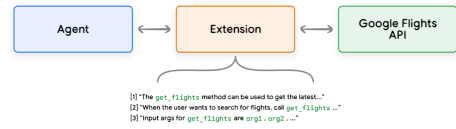


Figure 3.6: The connection from the agent to the external source.

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 taught 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. Parameters: "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 abilities, 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-practices 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: [30]

- **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-purpose 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 processing 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 Artificial Intelligence fields is Google Colaboratory. [31] 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. [32]

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 [33] and Bloom by BigScience (a worldwide collaborative research initiative coordinated by HuggingFace). [34]

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 definitely a barrier. In contrast, LLaMa stood out with its flexibility 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 September 2024, made it clear that the scaling law was to be obeyed 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 sibling, 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, receiving 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 efficiency and performance, offering a relatively lightweight architecture among large language models while still delivering a 128K context length, 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 of 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. [35] In particular, 8-bit quantization has been proven to offer a notable reduction in memory footprint 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 commercial 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 conveyed users’ check-ins to the city’s key attractions, and included detailed records of the date and time of each visit, the attraction visited, 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 anonymous, 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 when 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 longitude 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. [36]

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 following format:

```
1 import openmeteo_requests
2
3 # Setup the Open-Meteo API client
4 openmeteo = openmeteo_requests.Client(session = retry_session)
5
6 # The API's endpoint
7 url = "https://api.open-meteo.com/v1/forecast"
8
9 # Parameters of the query, including the coordinates of Verona, and hourly
10 # data of temperature, precipitation and precipitation probability for the
11 # current day
12 params = {
13     "latitude": 45.4299,
14     "longitude": 10.9844,
15     "hourly": ["temperature_2m", "precipitation_probability", "precipitation",
16               "weather_code"],
17     "timezone": "Europe/Berlin",
18     "forecast_days": 1
19 }
```

```

17
18 # Store the result of the request
19 responses = openmeteo.weather_api(url, params=params)

```

Code 4.1: Open-Meteo HTTP Request

By making on-demand queries and integrating weather data into the context, 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, standardizing 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.

```

1 # Set the model's architecture
2 model_name = 'meta-llama/Llama-3.1-8B-Instruct'
3
4 # Quantize the model to fit the GPU
5 quantization_config = BitsAndBytesConfig(load_in_8bit=True)
6
7 # Load the model
8 model = AutoModelForCausalLM.from_pretrained(
9     model_name,
10     device_map="auto",
11     quantization_config=quantization_config,
12     use_auth_token=True)
13
14 # Load the tokenizer
15 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 instantiate 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.

```

1 from flask import Flask, request, jsonify
2 import threading
3
4 # Initialize flask server
5 app = Flask(__name__)
6
7 # Routes a function that delivers user's prompt and returns the model's
8   response in a JSON format
9 @app.route('/predict', methods=['POST'])
10 def predict():
11     data = request.json
12     prompt = data.get('prompt', '')
13     messages.append({"role": "user", "content": prompt})
14     response = generate_response(tokenizer, model, messages)
15     messages.append({"role": "assistant", "content": response})
16     return jsonify({"response": response})
17
18 # Utility for re-initializing the chat
19 @app.route('/delete_history', methods=['POST'])
20 def delete_history():
21     del messages[7:]

```



```

21     return jsonify({"message": "History cleared"})
22
23 # Setup the ngrok tunnel
24 flask_public_url = ngrok.connect(5000)
25
26 def run_flask():
27     app.run(host="0.0.0.0", port=5000)
28
29 # Run flask server
30 flask_thread = threading.Thread(target=run_flask)
31 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.

```

1 # Write the streamlit web app
2 %%writefile app.py
3 import streamlit as st
4 import requests
5 import os
6 import time
7
8 # Get the Flask generated URL
9 flask_url = os.getenv("flask_url")
10
11 # The function sends a POST request in order to re-initialize the chat
12 def clear_chat_history():
13     url = f"{flask_url}/delete_history"
14     try:
15         response = requests.post(url)
16         if response.status_code == 200:
17             st.session_state.messages = [{"role": "assistant", "content": "How
18 may I assist you today?"}]
19             st.success("New chat.")
20         else:
21             st.error("Failed to clear chat history.")
22     except Exception as e:
23         st.error(f"Errore nella connessione al server Flask: {e}")
24
25 # A function that creates a typewriter effect
26 def llm_response_generator(prompt):
27     for word in prompt.split():
28         yield word + " "
29         time.sleep(.05)
30
31 # Initializing the chat
32 if "messages" not in st.session_state:
33     st.session_state.messages = [{"role": "assistant", "content": "How may I
34 assist you today?"}]
35
36 # Displays the chat in the UI
37 for message in st.session_state.messages:
38     with st.chat_message(message["role"]):
39         st.markdown(message["content"])
40
41 # The chat itself
42 if prompt := st.chat_input("What is up?"):

```

```

41 st.session_state.messages.append({"role": "user", "content": prompt})
42 with st.chat_message("user"):
43     st.markdown(prompt)
44
45     with st.chat_message("assistant"):
46         with st.spinner("Thinking..."):
47             # Sends a POST request to fire the model's response
48             url = f"{flask_url}/predict"
49             try:
50                 # Send the request to the Flask server
51                 response = requests.post(url, json={"prompt": prompt})
52                 if response.status_code == 200:
53                     answer = response.json().get("response", "Nessuna risposta
ricevuta")
54                     st.session_state.messages.append({"role": "assistant", "
content": answer})
55                 else:
56                     error = "Could you please repeat your question? I want to
make sure I provide you with the best possible answer."
57                     st.markdown(error)
58                     st.session_state.messages.append({"role": "assistant", "
content": error})
59             except Exception as e:
60                 st.error(f"Errore nella connessione al server Flask: {e}")
61
62             # Appends the response to the chat
63             st.write_stream(llm_response_generator(answer))
64
65 # Sidebar
66 with st.sidebar:
67     st.title('Veronacard Assistant')
68     st.subheader('Lorem Ipsum.')
69
70 st.sidebar.button('Clear Chat History', on_click=clear_chat_history)
71
72 # Setup the ngrok tunnel
73 streamlit_url = ngrok.connect(8501)
74
75 # Run the Streamlit web app
76 !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.

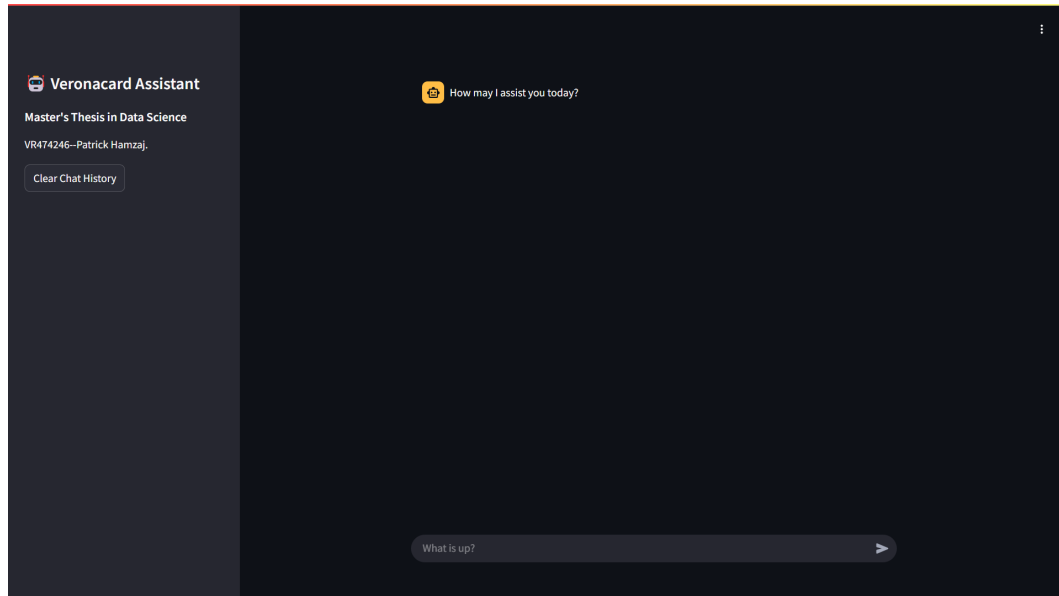


Figure 4.1: The front-end user interface.

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)`.

```

1 # This decorator tells Flask to set it as a callable endpoint
2 @app.route('/predict', methods=['POST'])
3 # The function called as soon as the user sends a message
4 def predict():
5     data = request.json
6     prompt = data.get('prompt', '')
7     messages.append({"role": "user", "content": prompt})
8     response = generate_response(tokenizer, model, messages)
9     messages.append({"role": "assistant", "content": response})
10    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 defining the model, as seen in Section 3.2.1 and implemented in Code 4.2.
2. **model:** The second main character, covering both architecture and checkpoints as seen in Section 2.2 and implemented in Code 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.

```
1 def generate_response(tokenizer, model, messages, tool_call=True):
2     """
3     Generates a response based on the provided prompt.
4
5     Args:
6         tokenizer: The tokenizer used to prepare the text.
7         model: The LLaMa model.
8         prompt: The initial prompt string.
9         max_length: Maximum length of the generated text.
10
11     Returns:
12         A string containing the generated response.
13     """
14
15     # Define the set of tools the model can use
16     tools = [retrieve_affluency, get_weather_forecast]
17
18     if not tool_call:
19         tools = None
20
21     # Takes the chat history, the set of tools and tells the model to return
22     # a response as a dictionary
23     inputs = tokenizer.apply_chat_template(
24         messages,
25         tools=tools,
26         add_generation_prompt=True,
27         return_dict=True,
28         return_tensors="pt"
29     )
30
31     # Move inputs to the same device as the model (GPU)
32     inputs = {key: tensor.to(model.device) for key, tensor in inputs.items()}
33
34     # Generate the output given the input, how many tokens the model can use,
35     # the temperature and select the top 50% tokens that are most probable
36     output = model.generate(
37         **inputs,
38         max_new_tokens=512,
39         temperature=0.5,
40         top_p=0.5,
41         do_sample=True
```

```

42 )
43
44 # Decode the output into human readable format
45 response = tokenizer.decode(output[0][len(inputs["input_ids"][0]):],
46                             skip_special_tokens=True)
47
48 # We handle the case when the model returns a tool call
49 if response.startswith("{"):
50     messages.append({"role": "assistant", "content": response})
51     response = eval(response)
52
53     # We extract the function called and the parameters set
54     name = response['name']
55     parameters = response['parameters']
56
57     arguments = {k: v for k, v in parameters.items() if v is not None}
58
59     tool_call = {"name": name, "arguments": arguments}
60     messages.append({"role": "assistant", "tool_calls": [{"type": "function", "function": tool_call}]})
61
62     # Both functions return the output with the parsed arguments, if
63     # called
64     if name == "retrieve_affluency":
65         content = retrieve_affluency(tool_call["arguments"]["location"],
66                                     tool_call["arguments"]["date"])
67     elif name == "get_weather_forecast":
68         content = get_weather_forecast(tool_call["arguments"]["date"])
69     else:
70         content = ""
71
72     # The function's output is then appended to the chat history
73     messages.append({"role": "tool", "name": name, "content": content})
74
75     # From here on, the behaviour is the same as a standard conversation,
76     # with the difference that the model now has to generate a response
77     # based on the
78     # function's output
79     inputs = tokenizer.apply_chat_template(
80         messages,
81         tools=tools,
82         add_generation_prompt=True,
83         return_dict=True,
84         return_tensors="pt"
85     )
86
87     inputs = {key: tensor.to(model.device) for key, tensor in inputs.items()}
88
89     output = model.generate(
90         **inputs,
91         max_new_tokens=512,
92         temperature=0.5,
93         top_p=0.5,
94         do_sample=True
95     )
96
97     response = tokenizer.decode(output[0][len(inputs["input_ids"][0]):],
98                             skip_special_tokens=True)
99
100     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 pre-trained 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 behavior 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 generate, 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 parameters 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 behavior in applying the chat template, setting generation parameters and finally decoding and returning the response. The implementation of this function is given in Code 4.7.

```
1 def retrieve_affluency(location: str, date: str) -> str:
2     """
3     Get the current affluency of a specific location, based on past data.
4
5     Args:
6         location: The location to get the affluency data.
```

```

7         date: The datetime to get the affluency data.
8
9     Returns:
10         The current affluency at the specified location.
11     """
12
13     if date:
14         parsed_date = dateparser.parse(date)
15         if parsed_date == None:
16             parsed_date = timefhuman(date)
17
18         hour = parsed_date.hour
19         month = parsed_date.month
20         weekday = parsed_date.weekday()
21     else:
22         today = datetime.now()
23         month = today.month
24         weekday = today.weekday()
25         hour = today.hour
26
27     prompt = is_location_crowded(location, month, weekday, hour)
28
29     if "medium" in prompt or "high" in prompt:
30         prompt += f'\nSuggest alternative time: {suggest_alternative_time(
31             location, month, weekday)}.'
32
33     return prompt

```

Code 4.7: retrieve_affluency() Function.

The main duty of this function is to parse the date and divides it into atomic parts, i.e. day, month, weekday and hour. It then proceeds to call another function, `is_location_crowded()`, that can be seen in Code 4.8, which takes these newly forged data, along with the point of interest, and performs a series of operation on the VeronaCard dataset.

```

1 def is_location_crowded(location: str, month: int, weekday: int, hour: int) ->
2     str:
3     # Handled the case when the prompt does not require a specific time,
4     # rather only a date
5     if hour == 0:
6         df_location = df[(df['sito_nome'] == location) & (df['weekday'] == int
7             (weekday))]
8         # If the hour parsed is not present in the records, it implicitly means
9         # that the site is closed
10        elif hour not in df[(df["sito_nome"] == location) & (df["mese_visita"] ==
11            month)][["ora_visita_intervallo"]].unique():
12            return "The site is closed."
13        # Otherwise, return all the records that correspond to the point of
14        # interest
15        # in the specified datetime
16    else:
17        df_location = df[(df['sito_nome'] == location) & (df['weekday'] == int
18            (weekday)) & (df['ora_visita_intervallo'] == int(hour))]
19
20    # Perform some aggregations in order to get monthly visits
21    location_visits = df_location.groupby(['anno_visita', 'mese_visita', '
22        weekday', 'ora_visita_intervallo']).size().reset_index(name='visits_number
23        ')

```

```

17 visits = location_visits.groupby(['mese_visita', 'weekday', '
    ora_visita_intervallo'])['visits_number'].mean().reset_index(name='
    visits_mean')
18
19 maps_ranking = visits[visits['mese_visita'] == month]
20 # If we get one record, it means it must return a suggestion
21 # incorporating also hour information we got in the arguments
22 if len(maps_ranking) == 1:
23     maps_ranking = maps_ranking['visits_mean'].iloc[0]
24     # Calculates the percentile
25     maps_percentile = (visits['visits_mean'] <= maps_ranking).mean() * 100
26     prompt = f"Estimated affluency for {location}: "
27
28     if maps_percentile <= 33:
29         prompt += "low."
30     elif maps_percentile <= 66:
31         prompt += "medium."
32     else:
33         prompt += "high."
34
35     return prompt
36 # Otherwise, a precise suggestion is made, because user did not
37 # give hour preference, but rather a generic recommendation
38 else:
39     try:
40         maps_percentile = maps_ranking[maps_ranking["visits_mean"] <
    maps_ranking["visits_mean"].median()].sort_values(by="
    ora_visita_intervallo")["ora_visita_intervallo"].iloc[0]
41         return f"Suggested time for {location}: {maps_percentile}."
42     except IndexError:
43         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 4.7, if the estimated affluence is not low, it provides an alternative time, which may provide less crowded visiting moments, implemented through Code 4.9.

```

1 def suggest_alternative_time(location: str, month: int, weekday: int) -> str:
2     # It performs similar operations of is_location_crowded function,
3     # but it returns the median value of the visits across the
4     # weekdays of the month, across all months
5     df_location = df[(df['sito_nome'] == location) & (df['weekday'] == int(
    weekday)) & (df['mese_visita'] == int(month))]

```



```

6 location_day = df_location.groupby(['anno_visita', 'mese_visita', 'weekday',
7 'ora_visita_intervallo']).size().reset_index(name='visit_count')
8 hours_affluency = location_day.groupby('ora_visita_intervallo')['visit_count'].agg(lambda x: x.mean()).reset_index(name='mean_visits')
9
10 # Less crowded time to visit the attraction
11 alternative = hours_affluency.median()
12
13 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.

The second tool at model's disposal is a function getting weather forecast for the following seven days. From the previously retrieved weather dataset for Verona, it proceeds to take the demanded information both in a daily and hourly manner, based on the specific user input: if the date is generic (e.g. *What is the weather like tomorrow?*) it returns an hour-by-hour paragraph eliciting the weather for the whole day, whereas when prompted, for example *I would like to visit the Arena saturday at 3pm, will it be sunny?*, the function returns the single information for saturday at 3pm. The definition of the function can be seen in Code 4.10:

```

1 def get_weather_forecast(date: str=None) -> str:
2     """
3     Get the weather forecast.
4
5     Args:
6         date: The datetime to get the weather data.
7
8     Returns:
9         The weather forecast for a specific day.
10    """
11
12    parsed_date = dateparser.parse(date)
13    if parsed_date == None:
14        parsed_date = timefhuman(date)
15
16    # If date is generic, return the forecast for the entire day
17    if parsed_date.hour == 0:
18        weather_forecast = next_7_days_weather[next_7_days_weather["date"].dt.date == parsed_date.date()]
19    # Else return the forecast for a specific point in time
20    else:
21        weather_forecast = next_7_days_weather[(next_7_days_weather["date"].dt.date == parsed_date.date()) & (next_7_days_weather["date"].dt.hour == parsed_date.hour)]
22
23    # Returns a prompt embedding the weather forecast, let it be hourly or specific data
24    prompt = ""
25    for _, row in weather_forecast.iterrows():
26        prompt += f"\nAt {row['date'].strftime('%H:%M')} the temperature is {int(row['temperature'])}C with {row['weather_description'].lower()}. The precipitation probability is {int(row['precipitation_probability'])}%."

```

```

27
28     return prompt

```

Code 4.10: `get_weather_forecast()` Function.

After having outlined both the operational settings as importing the LLM's architecture and checkpoints, importing the tokenizer with appropriate parameters, instantiate the model, and having described the set of tools enabling the dynamic 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 previous 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 Dicembre 2023. The weather information is retrieved and injected dynamically 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.

However, 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?

Also, 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.

Let 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:

```

1 # Tool call
2 {
3   'role': 'assistant',
4   'tool_calls': [
5     {
6       'type': 'function',
7       'function': {
8         'name': 'retrieve_affluency',
9         'arguments': {
10          'location': 'Giardino Giusti',
11          'date': '2025-01-22'
12        }
13      }
14    ]
15  },
16
17  # Function's output
18  {
19    'role': 'tool',
20    'name': 'retrieve_affluency',
21    'content': 'Suggested time for Giardino Giusti: 8.'
22  }
23

```

Code 4.11: 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.

Chapter 5

Evaluation

This chapter evaluates the outcomes of the project by analyzing the responses generated by LLaMa model following prompt engineering modifications. The evaluation focuses on several key aspects: the accuracy, coherence, relevance, and adaptability of the responses to incremental prompts scenarios, by adding at each iteration the strategies discussed in the previous chapters. In a qualitative assessment perspective, the analysis aims to highlight the improvements achieved through prompt engineering and identify areas for further enhancement. This comprehensive evaluation not only underscores the efficacy of the applied strategies but also provides insights into the model's performance across different dimensions of language understanding and generation. The subsequent sections present the outcomes across four distinct configurations: a baseline environment with no modifications, after the implementation of role prompting, following the addition of a RAG approach, and ultimately, after incorporating an agentic behaviour.

5.1 Conversations and the main observations

5.1.1 Baseline environment

The instruction-tuned language model facilitates a simple yet effective interaction, enabling a natural conversational flow: the conversation holds meaning in a general purpose interaction.

```
USER:      Hey, can you tell me any fun attraction to visit?

ASSISTANT: There are countless amazing attractions around the
→ world, but I'll give you some popular and unique ones to
→ consider:

1. The Great Barrier Reef (Australia): As the world's largest
→ coral reef system, it's a snorkeler's and scuba diver's
→ paradise.
2. The Grand Canyon (USA): One of the most iconic natural
→ wonders in the United States, offering breathtaking views
→ and hiking opportunities.
```

3. The Great Wall of China (China): A series of fortifications
→ built to protect the Chinese Empire, now a UNESCO World
→ Heritage Site.
4. The Eiffel Tower (France): An iconic symbol of Paris,
→ offering stunning views of the city from its top.
5. The Taj Mahal (India): A beautiful white marble mausoleum
→ built by the Mughal Empire, a testament to love and beauty.
6. The Colosseum (Italy): An ancient amphitheater in the heart
→ of Rome, a symbol of the Roman Empire's engineering and
→ architectural prowess.
7. The Aurora borealis (Northern Lights) (Arctic Circle): A
→ natural light display that's a sight to behold.
8. The Golden Gate Bridge (USA): An iconic suspension bridge in
→ San Francisco, offering stunning views of the Bay Area.
9. The Pyramids of Giza (Egypt): The last remaining ancient
→ wonder of the Seven Wonders of the Ancient World, a
→ must-visit for history buffs.
10. The Christ the Redeemer statue (Brazil): A massive Art Deco
→ statue of Jesus Christ in Rio de Janeiro, offering
→ breathtaking views of the city.

Some unique attractions to consider:

1. The Hang Son Doong cave (Vietnam): The world's largest cave,
→ with its own jungle, river, and even its own climate.
2. The Trolltunga rock formation (Norway): A challenging hike
→ rewarded with a unique rock formation resembling a troll's
→ tongue.
3. The Glowworm Grotto (New Zealand): A cave filled with
→ thousands of glowworms, creating a magical atmosphere.
4. The Chocolate Hills (Philippines): A unique geological
→ formation of over 1,000 conical hills, perfect for hiking
→ and exploration.
5. The Salar de Uyuni (Bolivia): The world's largest salt flat,
→ creating a surreal landscape of white salt and mirror-like
→ reflections.

USER: What do you suggest seeing in 24 hours in Verona,
→ Italy?

ASSISTANT: Verona, the City of Love! With only 24 hours, you
→ can still experience the best of Verona's romantic
→ atmosphere, rich history, and stunning architecture. Here's
→ a suggested itinerary to make the most of your time:

Morning (9:00 AM - 12:00 PM)

1. Piazza delle Erbe: Start your day in the heart of Verona's
→ historic center, surrounded by beautiful buildings, cafes,
→ and shops. Admire the stunning architecture, and take a
→ photo in front of the Aquila Fountain.
2. Ponte Pietra: Walk across this ancient Roman bridge, offering
→ breathtaking views of the Adige River and the city.
3. Verona Cathedral (Duomo): Visit the stunning cathedral, a
→ masterpiece of Romanesque architecture, and climb to the top
→ for panoramic views of the city.

Lunchtime (12:00 PM - 1:30 PM)

1. Try some local cuisine: Enjoy a traditional Veronese lunch at
→ a local trattoria, such as La Grotta or Trattoria al
→ Pompiere. Be sure to try the famous Risotto alla Pilota and
→ Prosciutto di Verona.

Afternoon (1:30 PM - 5:00 PM)

1. Juliet's Balcony: Visit the famous balcony where Juliet
→ Capulet is said to have stood in Shakespeare's Romeo and
→ Juliet. Take a photo in front of the balcony, and explore
→ the adjacent Casa di Giulietta (Juliet's House).
2. Piazza dei Signori: Explore this charming square, surrounded
→ by beautiful buildings, and visit the Torre dei Lamberti, a
→ 12th-century tower offering stunning views of the city.
3. Verona Arena: Visit this ancient Roman amphitheater, one of
→ the best-preserved in the world, and imagine the gladiators
→ who once fought here.

Evening (5:00 PM - 8:00 PM)

1. Admire the sunset: Find a spot with a view, such as the Ponte
→ Pietra or the Torre dei Lamberti, to watch the sunset over
→ the city.
2. Dinner: Enjoy a romantic dinner at a local restaurant, such
→ as La Taverna del Leone or Ristorante La Fontanella.

As can be noted, the model succeeds in returning meaningful interactions, gathering the knowledge acquired in the training phase and responding accordingly. Lacking of a give context, to the query *Hey, can you tell me any fun attraction to visit?*, the assistant takes the whole world as its competence, citing the Great Barrier Reef, the Eiffel Tower and the Taj Mahal. However, as soon as we restrain the focus querying *What do you suggest seeing in 24 hours in Verona, Italy?*, the expertise on the city of Verona becomes aware, thus returning the most famous points of interest—and, in this case, also a plan for visiting the city. It is dutiful though that an assistant must be readily aware of the domain involved in the conversation, and this behavior is enhanced as soon as a context is introduced.

5.1.2 Being a Tour Guide

In a first attempt of interacting with the chatbot, it was evident that a context awareness was lacking, but as some information is injected into the model's context, a specific behavior is triggered, accessing a restricted punctual knowledge that was already part of the baseline repository. In particular, the System prompt have been introduced, in order to set both the context and the role, as seen in Section 3.2.2:

```
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.
```

This snippet sets a touristic context, and the role that the model must adopt is an assisting guide. Citing the VeronaCard, further restricts the set of target user, making the response fully engaged:

```
SYSTEM:      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.
```

```
USER:        Hey, can you tell me any fun attraction to visit?
```

```
ASSISTANT:  Of course! There are so many things to see in
↪ Verona, I would begin with visiting the famous Arena.
```

```
USER:        What do you suggest seeing in 24 hours?
```

```
ASSISTANT:  With your 24-hour Veronacard, you can definitely
↪ see the main attractions.
```

```
First, start your day at the Arena di Verona, the ancient Roman
↪ amphitheater that hosts the famous Opera Festival every
↪ summer. Then, take a stroll through the Piazza Bra, the
↪ largest square in Verona, and visit the nearby Museo di
↪ Castelvecchio, a stunning 14th-century castle that now
↪ houses a museum with an impressive collection of art and
↪ artifacts.
```

```

Next, take a romantic gondola ride along the Adige River and
→ pass under the famous Ponte Pietra, a beautiful Roman
→ bridge. After that, visit the Casa di Giulietta (Juliet's
→ House), a 14th-century house that inspired Shakespeare's
→ famous balcony scene from Romeo and Juliet.

```

```

Finally, end your day with a visit to the Torre dei Lamberti, a
→ 12th-century tower that offers breathtaking views of the
→ city. With the Veronacard, you can skip the lines and enjoy
→ all these attractions without any hassle.

```

Using the same initial user prompt as the baseline case (i.e., *Hey, can you tell me any fun attraction to visit?*) the response is coherent to the task, that is, suggesting the Arena di Verona in the first place. Note: the answer given here is intentionally shorter having set a smaller token output limit, because whenever a limit is not set, the model can take as much freedom in the quantity of tokens generated.

Then, it can be noticed that give the input *What do you suggest seeing in 24 hours?*, the corresponding answer is similar to the baseline case, highlighting a plan of todos throughout the day: the behavior is similar but not equal, as the specification *Your tone is both professional and friendly at the same time.*, here *friendly* may have changed the format of the roadmap suggested.

Overall, the model demonstrated the capacity to maintain a professional yet friendly conversational tone, effectively emulating the role of a knowledgeable local tour guide. This approach not only enriches the user's engagement but also ensures that responses are concise, relevant, and practically beneficial.

5.1.3 Points of Interest and Weather

In the previous section setting a context aided the model to give more concise responses, readily able to suggest things to do in Verona, although not explicitly mentioned by the incoming query. Now, the next step involves further convey the possible set of responses to the specific tasks desired: which points of interest to address and weather data integration. The first piece of information is easily retrieved from the VeronaCard dataset as stated in Section 4.3.1: whereas the native knowledge of Verona is larger than the one required by the project, potentially recommending many more activities, the domain was restricted in order to be linked with the affluency data (which will be discussed in the next section). The weather data is fetched and prepared beforehand through the API seen in Section 4.3.2. The demonstration of directly inserting specific knowledge into the model's context seen in Section 3.4 is split in two distinct conversations, so as to fully appreciate its immediate effectiveness:

SYSTEM: 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.

Today's date is Tuesday, 07 January 2025.

You must suggest users on attractions to visit included in the

→ Veronacard, which are the following:

- AMO
- Arena
- Casa Giulietta
- Castelvechio
- 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

USER: Hey, what attraction should I visit?

ASSISTANT: Welcome to Verona! I'd be happy to help you plan
→ your day.

Have you thought about visiting Casa Giulietta, the famous
→ balcony of Juliet from Shakespeare's Romeo and Juliet? It's
→ a must-see for anyone visiting Verona, and it's included in
→ your Veronacard pass.

Alternatively, you could head to the Museo Storia, which offers

- a fascinating glimpse into Verona's rich history and art.
- Or, if you're interested in something a bit more unique, you
- could visit the Centro Fotografia, which showcases the work
- of local and international photographers.

Which one of these options sounds appealing to you? Or do you

- have any other preferences or interests you'd like to share?

It can be observed that the response to the baseline prompt returns a specific yet not very informative outcome of focusing on the VeronaCard points of interest (indeed Casa di Giulietta is a site included in the dataset), so to test the model's behavior in the general case the following continuation is given:

USER: Hello there!

ASSISTANT: Ciao! Welcome to Verona! I'm your tour guide today,

- and I'm here to help you make the most of your visit. What
- would you like to do first? Have you thought about visiting
- any of the attractions on your Veronacard pass? I can give
- you some recommendations based on your interests and the
- attractions you've already visited.

To further investigate the model's responses, the interaction continues with a user explicitly inquiring about VeronaCard-related attractions:

USER: What can I visit with the VeronaCard?

ASSISTANT: Benvenuto! Welcome to Verona! I'm thrilled to help

- you make the most of your VeronaCard. With this pass, you
- have access to over 30 points of interest in the city. Since
- you're looking for suggestions, I'll provide you with a few
- recommendations based on popular attractions and hidden
- gems.

Considering you're in Verona, I assume you've already visited

- some of the main sights. If not, I recommend starting with
- the iconic:

1. Arena di Verona (Arena): This ancient Roman amphitheater is a
- must-visit, and with the VeronaCard, you can skip the lines.
2. Casa di Giulietta (Casa Giulietta): The famous balcony where
- Romeo declared his love for Juliet is a must-see for any
- Shakespearean fan.
3. Duomo (Duomo): The stunning cathedral in the heart of the
- city is a masterpiece of Gothic architecture.

If you're interested in art and history, consider visiting:

1. Museo Lapidario (Museo Lapidario): A museum showcasing
→ ancient Roman and medieval inscriptions and sculptures.
2. Museo Africano (Museo Africano): A unique museum featuring a
→ collection of African art and artifacts.
3. Museo Storia (Museo Storia): Learn about Verona's rich
→ history and see artifacts from the city's past.

For a taste of the city's natural beauty, head to:

1. Giardino Giusti (Giardino Giusti): A beautiful garden with
→ stunning views of the city and a peaceful atmosphere.
2. Torre Lamberti (Torre Lamberti): A medieval tower offering
→ panoramic views of Verona.

If you're interested in photography, don't miss:

1. Centro Fotografia (Centro Fotografia): A photography museum
→ showcasing the work of local and international
→ photographers.

Lastly, if you're looking for something unique, consider
→ visiting:

1. Palazzo della Ragione (Palazzo della Ragione): A medieval
→ palace with a stunning courtyard and a fascinating history.
2. San Fermo (San Fermo): A charming church with a beautiful
→ cloister and a peaceful atmosphere.

Remember to check the opening hours and any specific
→ restrictions for each attraction before heading out. Enjoy
→ your exploration of Verona with your VeronaCard!

The model successfully points to the set of visitable attractions offered by the VeronaCard. It can now be observed a fragment of a conversation enriched with the weather data:

SYSTEM: 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.
Today's date is Thursday, 16 January 2025.
The weather throughout the day is as follows:

At 07:00 the temperature is -1°C with overcast. The
 ↳ precipitation probability is 0%.

At 08:00 the temperature is 0°C with overcast. The precipitation
 ↳ probability is 0%.

At 09:00 the temperature is 0°C with overcast. The precipitation
 ↳ probability is 0%.

At 10:00 the temperature is 2°C with partly cloudy. The
 ↳ precipitation probability is 0%.

At 11:00 the temperature is 5°C with partly cloudy. The
 ↳ precipitation probability is 0%.

At 12:00 the temperature is 7°C with overcast. The precipitation
 ↳ probability is 0%.

At 13:00 the temperature is 8°C with partly cloudy. The
 ↳ precipitation probability is 0%.

At 14:00 the temperature is 8°C with mainly clear. The
 ↳ precipitation probability is 0%.

At 15:00 the temperature is 8°C with mainly clear. The
 ↳ precipitation probability is 0%.

At 16:00 the temperature is 7°C with clear sky. The
 ↳ precipitation probability is 0%.

At 17:00 the temperature is 5°C with clear sky. The
 ↳ precipitation probability is 0%.

At 18:00 the temperature is 3°C with clear sky. The
 ↳ precipitation probability is 0%.

At 19:00 the temperature is 2°C with clear sky. The
 ↳ precipitation probability is 0%.

At 20:00 the temperature is 2°C with clear sky. The
 ↳ precipitation probability is 0%.

You must suggest users on attractions to visit included in the
 ↳ Veronacard, which are the following:

- AMO
- Arena
- Casa Giulietta
- Castelvechio
- 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

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.

USER: Hey, what attraction should I visit?

ASSISTANT: Buongiorno! Welcome to Verona. The weather today is
→ quite pleasant, so I'd recommend visiting the Giardino
→ Giusti. It's a beautiful garden with a rich history, and
→ it's a great place to relax and enjoy the scenery. Plus,
→ it's not too far from the city center, so you can easily get
→ there on foot.

If you're interested in something more cultural, you could visit
→ the Museo Storia, which is a museum showcasing the history
→ of Verona. It's a great place to learn more about the city's
→ past and its significance.

Lastly, if you're looking for something a bit more unique, you
→ could visit the Centro Fotografia, which is a photography
→ center that hosts exhibitions and workshops. It's a great
→ place to discover new artists and learn about the art of
→ photography.

Which one of these options sounds appealing to you?"

Which can be further checked directly asking for weather data:

USER: What is the weather like today?

ASSISTANT: Today's weather is quite pleasant, considering the
→ time of year. At 07:00, the temperature was -1°C with mainly
→ clear skies and a 0% chance of precipitation. As the day
→ progressed, the temperature rose to 6°C by 12:00, with clear
→ skies and still no precipitation. It's been a lovely day so
→ far, with plenty of sunshine.

If you're planning to explore the city, I'd recommend taking
→ advantage of the good weather. Many of our outdoor
→ attractions, such as the Giardino Giusti, would be perfect
→ for a stroll. Of course, if you'd prefer to stay indoors,
→ there are plenty of museums and galleries to choose from.


```
Would you like some recommendations for attractions based on the
↪ weather?
```

Note: the first paragraph points to past weather as the time of the conversation took place at 3pm.

The integration of structured data—specifically, the VeronaCard points of interest and weather data—into the conversational model’s context significantly enhances its ability to provide relevant and personalized recommendations. Throughout the conversations, the model consistently proved domain-specific information to suggest attractions that aligned well with both user inquiries and environmental conditions. This context-awareness allowed the model to differentiate between outdoor and indoor activities, intelligently adapting suggestions based on real-time weather data to improve the overall user experience.

5.1.4 Suggestions based on Attendance and Weather Forecasts

The final step in this project is to involve a tool that leverages the VeronaCard dataset, with the aim to make suggestions based on both weather data, and expected affluency, which is readily available from calling the function `retrieve_affluency()` defined in Code 4.7 and the function `get_weather_forecast()`. What makes it possible for a language model to take action based on its environment comprehension capability, and thus calling a certain tool, as discussed in Section 3.3.1, is the unique feature of LLMs to be *few-shot* learners. In this sense, a language model can be thought of as a pavlovian conditioned agent if instructed properly: in this specific case, the instance of LLaMa has been taught to call the user-defined function whenever a situation resembled the ones seen in a few examples. The first step is to make the model aware of the tools that can be called:

```
You can call the function 'retrieve_affluency' when the user
↪ asks how crowded is a certain attraction.
You can use the function 'get_weather_forecast' when the user
↪ asks about weather forecast in the next days.
```

Afterwards these examples had been provided, the first one being generated by the model itself by calling `generate_response()` with argument `tool_call=False`, whereas the two tool calls have been hardcoded and let the model generate the response with `tool_call=True`:

1. An example of a response involving its own inference system, and thus requiring a “natural” answer.

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

ASSISTANT: Welcome to Verona! I'd be happy to help you
↪ plan your day. Since you haven't mentioned any specific
↪ attractions yet, I'll suggest a few popular ones that
↪ are included in your Veronacard.
```

Considering the weather forecast for today, which is mostly
 → overcast with a slight chance of rain in the evening, I
 → recommend visiting some of the indoor attractions.

Here are a few suggestions:

1. Museo Africano: This museum is a great place to visit on
 → a rainy day. It features a collection of African art and
 → artifacts.
2. Museo Lapidario: This museum is dedicated to ancient
 → stones and inscriptions. It's a fascinating place to
 → learn about Verona's history.
3. Museo Storia: This museum showcases the history of
 → Verona, from ancient times to the present day. It's a
 → great place to learn about the city's rich heritage.

If you'd prefer to visit an outdoor attraction, I suggest
 → Giardino Giusti: this beautiful garden is a peaceful
 → oasis in the heart of the city. It's a great place to
 → take a stroll and enjoy the scenery.

Which of these attractions sounds interesting to you? Or do
 → you have any specific preferences (e.g., art, history,
 → architecture)?"

2. An hardcoded example of calling the tool used for retrieving attendance statistics, for which a model-generated answer is generated.

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

```

1 from datetime import datetime
2
3 # Get and format the current date
4 current_date = datetime.now()
5 formatted_date = current_date.strftime("%Y-%m-%d")
6
7 # Hard code the tool call
8 tool_call = {
9     "name": "retrieve_affluency",
10    "arguments": {
11        "location": "Giardino Giusti",
12        "date": formatted_date
13    }
14 }
15
16 # Append the tool call to the chat history
17 messages.append(
18     {

```

```

19         "role": "assistant",
20         "tool_calls": [
21             {
22                 "type": "function",
23                 "function": tool_call
24             }
25         ]
26     }
27 )
28
29 # Get the result of the function call
30 affluency = retrieve_affluency(
31     tool_call["arguments"]["location"],
32     tool_call["arguments"]["date"]
33 )
34
35 # Again append it to the chat history
36 messages.append(
37     {
38         "role": "tool",
39         "name": "retrieve_affluency",
40         "content": affluency
41     }
42 )
43
44 # Generate the response with the tool call output
45 messages.append(
46     {
47         "role": "assistant",
48         "content": generate_response(tokenizer, model, messages)
49     }
50 )

```

Code 5.1: Few-shot on retrieve_affluency().

ASSISTANT: Considering you'd like to avoid crowds, I'd

- ↪ suggest visiting Giardino Giusti in the morning, around
- ↪ 9:00 AM. The weather is expected to be partly cloudy,
- ↪ with a temperature of 5°C and a slight chance of rain.
- ↪ It might be a bit chilly, but the garden is a beautiful
- ↪ spot to visit, and the early morning hours might help
- ↪ you avoid the larger crowds.

Alternatively, if you're not a fan of the rain, you could

- ↪ consider visiting in the late afternoon, around 4:00 PM.
- ↪ The sun will be setting, casting a warm glow over the
- ↪ garden, and the crowds will likely be smaller since it's
- ↪ not peak tourist season.

Keep in mind that these are just suggestions, and the actual

- ↪ crowd levels may vary depending on various factors. If
- ↪ you'd like more specific information or have any other
- ↪ questions, feel free to ask!

3. An hardcoded example of calling the tool used for fetching weather data for the following seven days and, as the previous tool call, the response is generated by the model.

USER: What is the weather like tomorrow at 3pm?

```

1 from datetime import timedelta
2
3 # Get tomorrow's date
4 tomorrow = current_date + timedelta(days=1)
5 tomorrow_at_3pm = tomorrow.replace(hour=0, minute=0, second=0,
6                                     microsecond=0)
7 tomorrow_at_3pm = tomorrow_at_3pm.strftime("%Y-%m-%d %H:%M:%S")
8
9 # Tool call
10 tool_call = {
11     "name": "get_weather_forecast",
12     "arguments": {
13         "date": tomorrow_at_3pm
14     }
15 }
16
17 # Append the tool call to the chat history
18 messages.append(
19     {
20         "role": "assistant",
21         "tool_calls": [
22             {
23                 "type": "function",
24                 "function": tool_call
25             }
26         ]
27     }
28 )
29
30 # Return the function call
31 forecast = get_weather_forecast(tool_call["arguments"]["date"])
32
33 # Append the returned value to the chat history
34 messages.append(
35     {
36         "role": "tool",
37         "name": "get_weather_forecast",
38         "content": forecast
39     }
40 )
41
42 # Generate the response
43 messages.append(
44     {
45         'role': 'assistant',
46         'content': generate_response(tokenizer, model, messages)
47     }
48 )

```

Code 5.2: Few-shot on `get_weather_forecast()`.

ASSISTANT: Tomorrow at 3:00 PM, the weather is expected
 ↪ to be overcast with a temperature of 10°C and a
 ↪ precipitation probability of 8%.

By structuring the dialogue flow in this manner, we enable the model to decide whether or not a specific function call is necessary in response to a user’s query. When the user inquires about crowd levels at particular attractions, the model can invoke `retrieve_affluency()`, inspect the returned data, and then integrate that knowledge into its suggestion. Conversely, if the user’s request concerns weather conditions for a future date, the model knows it can rely on `get_weather_forecast()`. The few-shot examples guide the model on how to produce an “ordinary” answer (one that relies purely on its internal language modeling) or a function call (to fetch external data) without explicit manual intervention at each step.

Thus, the synergy between carefully curated exemplars and the model’s tool-calling capability effectively simulates a dynamic assistant that can adapt its responses to real-time data. Consequently, when users ask for suggestions on what to visit, at which times, or under various weather conditions, the model seamlessly weaves together its internally generated text with up-to-date affluency and forecast information. This integration of external resources illustrates the flexible and context-aware nature of large language models: they can switch between relying solely on their generative abilities and calling out to domain-specific functions whenever the user’s request resembles the few-shot instructions already provided.

Despite these promising results, the approach described above is not without limitations. The practical experiments conducted during this research revealed several constraints inherent to the model’s decision-making and tool-calling capabilities. The following section addresses these main limitations, emphasizing scenarios where the current methodology exhibits weaknesses or potential areas for improvement.

5.2 Limitations of the Current Approach

Leveraging prompt engineering along with the integration of a set of tool at model’s disposal really demonstrates that a lack of computation resources does not automatically involve a lack of performance in delivering a natural and fruitful interaction that seamlessly achieve specific tasks.

The power of this strategy is not limitless though: the control over the type and quality of the model’s responses is limited, due to the restraint adaptation of the model’s weights, which inherently encloses all the generation power a foundational model holds. Take for example this response:

ASSISTANT: With your 24-hour Veronacard, you can definitely
 ↪ see the main attractions.

First, start your day at the Arena di Verona, the ancient Roman
→ amphitheater that hosts the famous Opera Festival every
→ summer. Then, take a stroll through the Piazza Bra, the
→ largest square in Verona, and visit the nearby Museo di
→ Castelvecchio, a stunning 14th-century castle that now
→ houses a museum with an impressive collection of art and
→ artifacts.

Next, take a romantic gondola ride along the Adige River and
→ pass under the famous Ponte Pietra, a beautiful Roman
→ bridge. After that, visit the Casa di Giulietta (Juliet's
→ House), a 14th-century house that inspired Shakespeare's
→ famous balcony scene from Romeo and Juliet.

Finally, end your day with a visit to the Torre dei Lamberti, a
→ 12th-century tower that offers breathtaking views of the
→ city. With the Veronacard, you can skip the lines and enjoy
→ all these attractions without any hassle.

Here, after corretly suggesting to visit the main character of Verona, namely the Arena, and to take a stroll through Piazza Bra, it curiosly suggests taking a *romantic gondola ride along the Adige River*, which is clearly unrealistic, as Adige river is not navigable and gondolas are well-known for being an attraction of Venice. Or again, looking at the following response:

ASSISTANT: Giardini Giusti is a hidden gem in Verona. It's a
→ beautiful 16th-century garden that's often overlooked by
→ tourists, but it's definitely worth a visit. The Giardini
→ Giusti is a UNESCO World Heritage Site and a masterpiece of
→ Italian garden design.

This stunning garden is a tranquil oasis in the heart of the
→ city, with its beautiful fountains, statues, and walking
→ paths. You can stroll along the garden's many paths, take in
→ the sights, and enjoy the peaceful atmosphere. The garden is
→ also home to a variety of plants and flowers, including some
→ rare and exotic species.

One of the highlights of the Giardini Giusti is the Palazzo
→ Giusti, a 16th-century palace that now houses a museum and a
→ library. The palace features an impressive collection of art
→ and artifacts, including paintings, sculptures, and
→ furnishings from the Renaissance period.

As a Veronacard holder, you can visit the Giardini Giusti
 ↳ without any additional cost. I would recommend visiting the
 ↳ garden in the late morning or early afternoon when the sun
 ↳ is not too strong, and the atmosphere is particularly
 ↳ peaceful. It's the perfect spot to relax and unwind after a
 ↳ busy day of exploring the city.

Would you like to visit the Giardini Giusti now, or would you
 ↳ like to explore more of Verona's hidden gems?

The reply sees a beautiful description of Giardini Giusti, with several historical and descriptive details: when it comes to giving a suggestion, the agent says *I would recommend visiting the garden in the late morning or early afternoon when the sun is not too strong [...]*, which is exactly the opposite of what one expects—the sun is strong precisely during the central hours.

These kind of mistakes are often referred to as *hallucinations*: in the context of LLMs, "hallucinations" refer to instances where the AI generates content that appears factual and coherent but is ungrounded or incorrect. This phenomenon occurs when the model produces information that isn't supported by its training data or external knowledge sources, leading to plausible-sounding yet inaccurate responses. [37]

Understanding and addressing hallucinations in LLMs is crucial for enhancing the reliability and trustworthiness of AI-generated content, especially in applications where factual accuracy is essential.

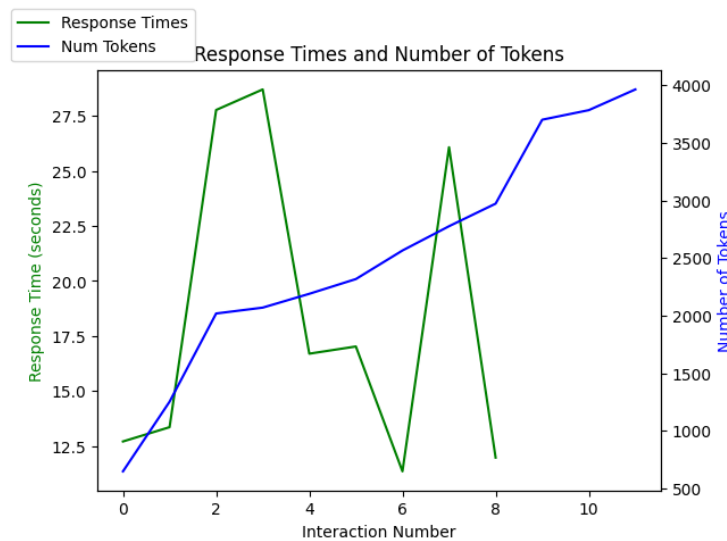


Figure 5.1: Time and number of tokens throughout the turns in a conversation.

Another difficulty resides in the memory—the context length—that a model can withstand. LLaMa 3.1 8B has an extremely elevated endurance, supporting a 128K context length; still, the number of tokens involved in the strategies adopted increases exponentially, making it difficult to tackle numerous multi-turn conversation in the long run. Related to this last

issue, is the time required to generate an output. Figure 5.1 shows that there is not a correlation between the number of tokens generated and the time required to give an output, though it demonstrates that it can take up to 30 seconds to produce a response, as it can significantly influence the user experience.

In conclusion, prompt engineering combined with integrating external tools significantly enhances the capabilities of language models, demonstrating that high computational resources are not strictly necessary for achieving effective, natural, and productive interactions. However, as observed through examples, this strategy is not without limitations. The model’s reliance on fixed foundational weights can lead to hallucinations—plausible yet factually incorrect outputs—as demonstrated in the unrealistic recommendation of a gondola ride on Verona’s Adige River, or the misinformed advice about visiting Giardini Giusti during peak sunlight hours.

Furthermore, the issue of context length and memory remains critical. Even models such as LLaMa 3.1 8B, capable of handling extended contexts of up to 128K tokens, can still face significant performance challenges due to exponentially increasing token counts in complex, multi-turn conversations. Additionally, response times, though not directly correlated with the number of tokens generated, can occasionally become prohibitively long, posing practical constraints on user experience.

Future research must focus on mitigating hallucinations, optimizing context management strategies, and improving generation speed to fully harness the potential of prompt-driven and tool-integrated language models, thus making these approaches robust, reliable, and truly beneficial for practical applications.

Conclusions

This thesis has presented a practical exploration into the integration of alternative techniques to the industry standards of fine-tuning for implementing Large Language Models (LLMs), specifically focusing on developing a context-aware recommender system designed for tourists visiting Verona, Italy. By leveraging the open-source LLaMA 3.1 8B-Instruct model and employing advanced techniques such as 8-bit quantization, Retrieval-Augmented Generation, Prompt Engineering and Agentic AI, this study demonstrates that it is feasible to deploy state-of-the-art conversational AI solutions even under limited computational resources.

Chapter 2 addressed the foundational aspects of neural networks and their evolution, starting with the very first attempt of reproducing the human brain to render machines as learning entities, the *Perceptron*, which laid the foundations for successful machine learning architectures, until highlighting the transformative role of the Transformer architecture, introducing a Multi-Head Attention mechanism which captures the semantic of natural language with all its facets efficiently.

Chapter 3 followed by a thorough exploration of recent advancements in LLMs, including key techniques such as Prompt Engineering, Retrieval-Augmented Generation (RAG), and Agent AI. These elements are fundamentally possible due to demonstrated *few-shot* capabilities of modern LLMs, due to increasing parameter sizes and hardware advancements, and provided the theoretical and practical foundation essential for building domain-specific conversational AI systems.

In the implementation phase (Chapter 4) the thesis focused on a practical use-case: recommending points of interest to tourists in Verona. An 8-bit quantized version of the LLaMA 3.1 8B-Instruct model was selected due to its efficient balance of computational demand and functional capability. The developed VeronaCard Assistant successfully illustrates how carefully structured prompts, combined with dynamically retrieved external data—such as historical tourist attendance records and real-time weather conditions—can significantly enhance the practicality, accuracy, and usefulness of AI-generated recommendations.

Experimental results described in Chapter 5 and user interactions confirmed that the application effectively guided users toward personalized experiences, adapting recommendations based on dynamic variables like attendance patterns and weather forecasts. The assistant showed promising capabilities in generating meaningful, contextually appropriate recommendations. It effectively responded to user inquiries, handled multi-turn conversations gracefully, and incorporated external data sources efficiently. However, several limitations were identified, including occasional inaccuracies in recommendations due to model hallucinations and challenges in maintaining relevance in dynamic scenarios. Additionally, while prompt engineering and role prompting significantly improved context-awareness and user engagement, fine-tuning remains a key strategy to significantly implement useful models, especially regarding nuanced user preferences and specific conversational contexts.

In conclusion, the thesis has provided meaningful insights into leveraging large language

CONCLUSIONS

models for domain-specific applications, such as tourist engagement and recommendation systems. It demonstrated the practical application of state-of-the-art AI techniques within the tourism sector, effectively bridging theoretical advancements with real-world use cases. The use of Retrieval-Augmented Generation, careful Prompt Engineering and Agentic behavior with limited computational resources highlights the significant potential and practicality of LLM-based recommender systems. As LLM technologies continue to advance, opportunities for further enhancement and broader application will undoubtedly arise, offering ever more engaging, intelligent, and helpful user experiences in tourism and beyond.

Bibliography

- [1] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: <https://arxiv.org/abs/2005.14165>.
- [2] F. Rosenblatt. “The perceptron: A probabilistic model for information storage and organization in the brain [J]”. In: *Psychol. Review* 65 (Nov. 1958), pp. 386–408. DOI: 10.1037/h0042519.
- [3] 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. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: <https://doi.org/10.1007/BF02478259>.
- [4] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, Sept. 2017. ISBN: 9780262343930. DOI: 10.7551/mitpress/11301.001.0001. URL: <https://doi.org/10.7551/mitpress/11301.001.0001>.
- [5] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: <https://doi.org/10.1038/323533a0>.
- [6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: <http://www.deeplearningbook.org>.
- [7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.
- [8] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: <https://doi.org/10.1162/neco.1997.9.8.1735>.
- [9] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: <https://arxiv.org/abs/1706.03762>.
- [10] Yiheng Liu et al. *Understanding LLMs: A Comprehensive Overview from Training to Inference*. 2024. arXiv: 2401.02038 [cs.CL]. URL: <https://arxiv.org/abs/2401.02038>.
- [11] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: <https://arxiv.org/abs/1810.04805>.
- [12] 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>.

- [13] Lei Huang et al. “A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions”. In: *ACM Transactions on Information Systems* 43.2 (Jan. 2025), pp. 1–55. ISSN: 1558-2868. DOI: 10.1145/3703155. URL: <http://dx.doi.org/10.1145/3703155>.
- [14] Emily M. Bender et al. “On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?” In: *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*. FAccT ’21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 610–623. ISBN: 9781450383097. DOI: 10.1145/3442188.3445922. URL: <https://doi.org/10.1145/3442188.3445922>.
- [15] Emma Strubell, Ananya Ganesh, and Andrew McCallum. “Energy and Policy Considerations for Deep Learning in NLP”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen, David Traum, and Lluís Màrquez. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 3645–3650. DOI: 10.18653/v1/P19-1355. URL: <https://aclanthology.org/P19-1355/>.
- [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 et al. *Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing*. 2021. arXiv: 2107.13586 [cs.CL]. URL: <https://arxiv.org/abs/2107.13586>.
- [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] Hugging Face. *The Hugging Face Course, 2022*. <https://huggingface.co/course>. 2022.
- [22] 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>.
- [23] Rico Sennrich, Barry Haddow, and Alexandra Birch. *Neural Machine Translation of Rare Words with Subword Units*. 2016. arXiv: 1508.07909 [cs.CL]. URL: <https://arxiv.org/abs/1508.07909>.
- [24] Alec Radford et al. “Improving Language Understanding by Generative Pre-Training”. In: *OpenAI Technical Report* (2018).
- [25] OpenAI. *Text Generation*. <https://platform.openai.com/docs/guides/text-generation>. 2025.
- [26] 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>.

-
- [27] 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>.
 - [28] 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>.
 - [29] W3Schools. *ChatGPT-3.5 Roles*. 2025. URL: https://www.w3schools.com/gen_ai/chatgpt-3-5/chatgpt-3-5_roles.php.
 - [30] 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>.
 - [31] Google. *Google Colaboratory*. 2025. URL: <https://colab.research.google.com/>.
 - [32] NVIDIA. *Tesla T4 - NVIDIA Data Center*. 2025. URL: <https://www.nvidia.com/it-it/data-center/tesla-t4/>.
 - [33] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: <https://arxiv.org/abs/2302.13971>.
 - [34] BigScience Workshop et al. *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*. 2023. arXiv: 2211.05100 [cs.CL]. URL: <https://arxiv.org/abs/2211.05100>.
 - [35] 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>.
 - [36] *Open-Meteo: Free Weather API*. <https://open-meteo.com/>. 2023.
 - [37] Ziwei Ji et al. “Survey of Hallucination in Natural Language Generation”. In: *ACM Computing Surveys* 55.12 (Mar. 2023), pp. 1–38. ISSN: 1557-7341. DOI: 10.1145/3571730. URL: <http://dx.doi.org/10.1145/3571730>.

BIBLIOGRAPHY

List of Figures

2.1	The perceptron architecture.	4
2.2	The Transformer architecture.	5
3.1	The tokenization process referring to text classification, as seen in [21].	13
3.2	Different prompting techniques can increase models' accuracy, as seen in [1].	16
3.3	Standard prompting compared to Chain-of-Thought prompting, as seen in [27].	17
3.4	RAG architecture, as seen in [20].	19
3.5	An end-to-end agentic behavior.	20
3.6	The connection from the agent to the external source.	20
3.7	An agent workflow.	20
4.1	The front-end user interface.	31
5.1	Time and number of tokens throughout the turns in a conversation.	59

LIST OF FIGURES

List of Tables

2.1	Examples of Instruction Tuning	6
3.1	Roles definition	15

List of Code Snippets

4.1	Open-Meteo HTTP Request	26
4.2	Importing LLaMa through Transformers Library	28
4.3	Writing the Streamlit web app.	28
4.4	Flask API Internal Service	29
4.5	Predict Flask API.	31
4.6	<code>generate_response()</code> Function.	32
4.7	<code>retrieve_affluency()</code> Function.	34
4.8	<code>is_location_crowded()</code> Function.	35
4.9	<code>suggest_alternative_time()</code> Function.	36
4.10	<code>get_weather_forecast()</code> Function.	37
4.11	Tool call	41
5.1	Few-shot on <code>retrieve_affluency()</code>	54
5.2	Few-shot on <code>get_weather_forecast()</code>	56

