ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Department of Computer Science and Engineering
Computer Science and Engineering

Large Action Models: End-to-End Retrieval-Enhanced Learning for Generating Function Calls from Instruction Manuals

Bachelor Thesis in
Data Intensive Applications

Supervisor
Prof. Gianluca Moro

Candidate
Nicolò Monaldini

Co-supervisors
PhD. Giacomo Frisoni
Dr. Lorenzo Molfetta

First Session
Academic Year 2023 – 2024

KEY WORDS

Large Action Models

Agents

End-to-End Retrieval-Augmented Generation

Function Calling

Natural Language Processing

Vindica te tibi

— Seneca

Abstract

Large action models are agents that leverage the reasoning abilities of large language models (LLMs) to make decisions in real-life scenarios. Existing approaches often fine-tune LLMs for specific instruction-function mappings, which leads to limited generalizability and eventual obsolescence. LLMs that support function calling natively require a list of tools to be passed to the model, usually in JSON format. However, their context length limits the number of supported tools. Additionally, leveraging this functionality with pay-as-you-go closed-source models can result in high inference costs. Moreover, even cutting-edge models can hallucinate or make errors in tool selection when presented with many options. This work evaluates how retrieval-augmented generation could enhance generalization in tool selection and function-calling tasks. Specifically, we treat the LLM as a frozen black box and augment it with a tunable retriever module, trained to find the documentation chunks that maximize the LLM accuracy in tool selection. Our retriever preselects relevant function headers for a given query to mitigate context length restrictions and hallucination phenomena of the LLM to which it is plugged. We conducted extensive evaluations with various models, datasets, and loss functions. For functions already seen during training, using RoBERTa-base, we observed significant performance improvements: Rank@1 increased from 22.5% to 85.0%, Rank@2 from 27.5% to 100.0%, and Rank@3 from 30.0% to 100.0%. For functions not seen during training, Rank@1 improved from 40.0% to 75.0%, Rank@2 from 50.0% to 97.5%, and Rank@3 from 50.0% to 97.5%.

Introduction

Large language models (LLMs) have demonstrated remarkable abilities in task comprehension and can explain how tasks should be performed. However, they generally cannot perform tasks themselves, as most well-known language models (LMs) are natively text-in-text-out. With large action models (LAMs), the scientific community aims to enhance LLMs with the ability to select and execute tools, which could offer advantages in areas like automation and real-life interaction despite the risks associated with using LLMs without proper training. This capability could also be beneficial in domains where LLMs typically underperform, such as mathematics, by leveraging suitable tools to improve performance. The general interest and excitement of this topic have encouraged many companies and organizations to develop projects employing agents. Nonetheless, many rely on fine-tuning to specialize LLMs on a limited set of functions, reducing their generalization capabilities in various contexts [1]. The lack of datasets related to function calling and tool selection brought many works to selecting some functions and generating synthetic data regarding them [2, 3]. In this work, we use retrieval-augmented generation (RAG), a well-known approach that enables LMs to use external knowledge to overcome the limitations of specialization over a limited set of functions. Inspired by the REPLUG framework [4], which trains an encoder using an end-to-end approach, we perform our training in a way that is easily extensible to datasets where examples do not have associated gold-standard documents. Given a knowledge base, for each training example, a retriever selects documents that will be included in the prompt passed to the LLM, and the performance of the retriever will then be judged based on the quality of the LLM's outputs. This leverages RAG to select a set of adequate tools for each query that the

LLM could decide to use to satisfy the instruction given. Additionally, many machine learning frameworks developed agents that leverage the tool selection capabilities of LLMs to execute the selected tools through various approaches that will be later explored, allowing even self-debugging and error correction by the LLM itself. The forthcoming chapters in this thesis, which provide an in-depth exploration of our approach and the results obtained, are organized as follows:

- **Chapter 1 - Theoretical Framework:** The core concepts related to the subject covered and the work carried out are introduced in this chapter.
- **Chapter 2 - Related Work:** This chapter covers the existing research that inspired and paved the way for this work. A review of the relevant research in this field explains the main approaches used in previous works.
- **Chapter 3 - Method:** This chapter provides an in-depth exploration of our approach, including the works that inspired it and the differences from those works.
- **Chapter 4 - Experimental Setup:** Within this chapter, light is shed on the implementation of our approach, datasets and models used are carefully described, metrics used to measure performance are presented, as well as implementation specifics and hyperparameters.
- **Chapter 5 - Results and Discussions:** In this chapter, evaluation results of our approach are presented and interpreted to better understand the contributions of this work.
- **Conclusions and Future Work:** Wrapping up this thesis, this chapter synthesizes the work done. The implications and key findings of the work are discussed, highlighting the opportunities to further explore this field of research.

Index

List of Figures

List of Tables

Chapter 1

Theoretical Framework

This chapter introduces and briefly explains fundamental concepts related to artificial intelligence (AI), machine learning, and natural language processing (NLP).

1.1 Artificial Intelligence overview

Artificial intelligence is a field of computer science that studies how to create systems that are able to solve tasks that require human intelligence, such as reasoning, human interaction, or natural language comprehension. As Marco Solmavico defined it [5], it is a discipline that studies the theoretical foundations, the methodologies and the techniques that allow the design of hardware systems and software programs capable of providing the computer with performances that, to a common observer, would seem to be exclusively within the domain of human intelligence. It is an area which comprises many sub-fields, as depicted in Figure 1.1.

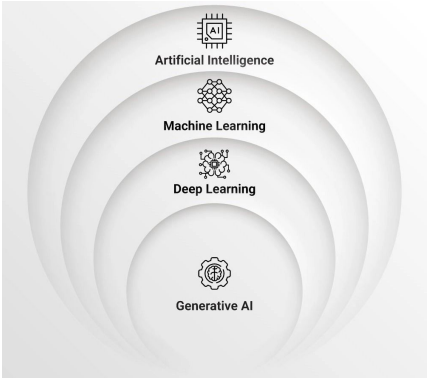

Figure 1.1: Artificial intelligence landscape. Source: www.synotpek.com.

1.1.1 Machine learning

Machine learning is a subset of AI that studies how to provide computers with the ability to solve tasks they were not explicitly programmed to solve. This is a big breakthrough in computer science since, for the first time, the focus is no longer on programming computers explicitly to handle all specific cases of a task not to fall into errors but rather on using data to teach computers how to approach a task effectively. In machine learning, everything revolves around data, and having the right data for a specific task is fundamental; sometimes, however, these data may not be labelled, which means that a ground truth response is not available to correct the models when the output is wrong. Therefore, based on the structure of the available data, the training process can be done in different ways:

- **Supervised training:** In this approach, the data are already labelled and can be compared with the outputs of the machine learning models.
- **Unsupervised learning:** In this case, the labels are unavailable, and the models have to find the data patterns independently.
- **Self-supervised learning:** Similarly to the latter, labels are missing, but in this case, the inputs are used as the supervision; this is the case of models trained for text generation on a large corpus of text.
- **Reinforcement learning:** It is the case where a model does not learn from a dataset but is rather performing actions within an environment. It receives rewards or punishment based on its actions, thus learning which actions lead to the best results.

1.1.2 Deep Learning

When models used in machine learning are based on deep neural networks, the process is referred to as deep learning. Although machine learning can employ simpler models based on techniques such as polynomial regression or regression trees, deep learning models rely exclusively on deep neural networks, which are composed of artificial neurons, as shown in Figure 1.2.

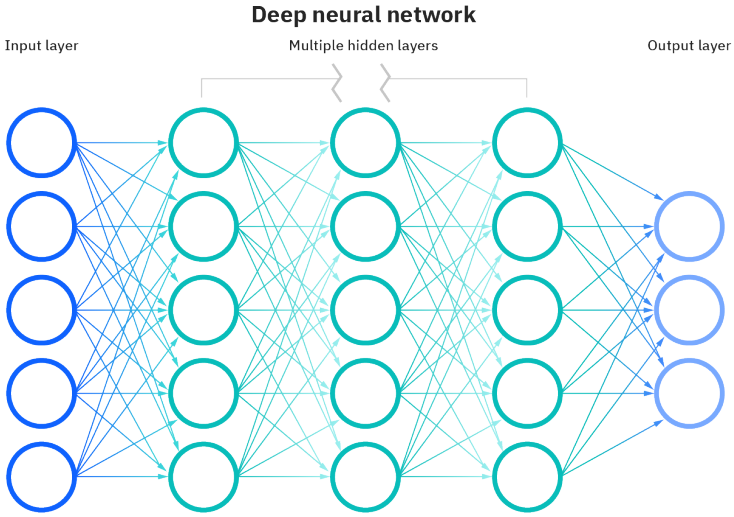

Figure 1.2: Structure of a deep neural network. Source: www.analyticsvidhya.com.

Inspired by the structure of the human brain, artificial neurons are organized into layers. The duty of an artificial neuron is quite simple: it first receives outputs from the neurons of the previous layer as its input and performs a simple mathematical function on them, including a parameter called bias and considering some trainable parameters called weights, which indicate the importance of the connections with the neurons of the previous layer. After that, it passes a value to the neurons of the next layer based on the result of an activation function. Activation functions are non-linear, enabling the deep neural network to understand non-linear relationships between data. The data, represented as numbers, are brought to a new vector space in every layer. In a classification task, for example, thanks to multiple layers, the data in the last layer is finally brought to a vector space where it can be easily divided into classes.

1.2 Large Language Models

When models are used to produce content, such as text or images, they can be referred to as generative AI models. LLMs are often used to perform NLP tasks, including generative ones. An LM is called "large" when it has an enormous number of parameters, often in the order of billions, and is trained on a tremendous amount of data. To better understand how models generate text, we first explain how text is processed by these models.

1.2.1 Embeddings

LMs do not understand text as sequences of characters; they expect the input to be in the form of matrices to do matrix multiplication operations. This is why the input is mapped to a sequence of high-dimensional vectors. The input sequences are first split into the so-called *tokens*, and then each token is converted to a vector in a high-dimensional vector space, called *embedding*. Tokens often consist of subwords but can also encode entire words or single characters. Subword tokenization is usually preferred because it allows for a contained vocabulary size and the management of rare words.

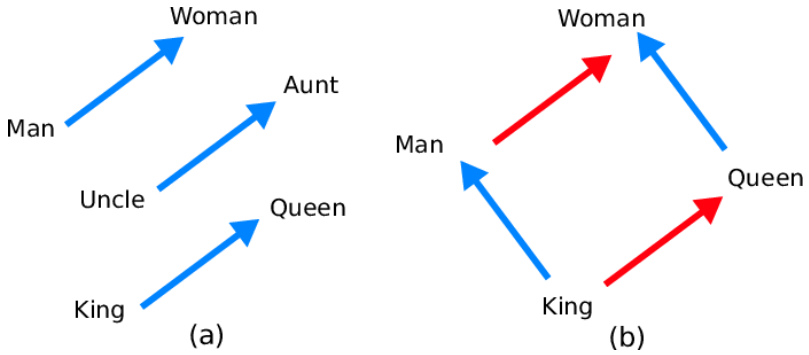

Figure 1.3: Visual representation of embeddings vectors. Source: "IEEE Transactions on Visualization and Computer Graphics" [6].

In the vector space in which tokens are mapped, every vector aims to capture the semantic meaning of the encoded token, and tokens with similar meanings are positioned close together in the vector space. These vectors can also show interesting algebraic relationships, as shown in Figure 1.3. For example, the

following relation applies:

$$v(\text{queen}) - v(\text{king}) + v(\text{man}) \approx v(\text{woman})$$

This suggests that, in this example, information about gender is encoded on a specific dimension (embeddings usually have a number of dimensions in the order of hundreds or thousands).

1.2.2 Transformers

Before 2017, text generation and, in general, all NLP tasks relied on recurrent neural networks (RNNs). This architecture was limited by problems such as:

- **Vanishing gradients:** This problem occurs when the gradients of the loss function become extremely small as they are propagated backwards through the network during training, slowing or stopping the gradient descent process.
- **Exploding gradients:** It is the opposite of the latter problem; gradients' explosion leads to unstable training and oscillations in the loss function.

Vaswani et al. introduced a new architecture, called transformers, that solved these problems and revolutionized NLP [7]. The main idea introduced by the paper is the concept of attention, a mechanism that allows each token to focus on different parts of the input sequence.

$$\mathit{Attention}(Q, K, V) = \mathit{softmax}(\frac{QK^T}{\sqrt{d_K}})V \tag{1.1}$$

In Equation 1.1, Q represents the input sequence, K represents the features used to compare the inputs, and V represents the values. Each token represented in Q finds its final representation in V through K, similar to hashing retrieval. The matrices Q, K and V are obtained by multiplying the inputs x with the matrices W_Q, W_K and W_V, respectively, whose weights are learnt during training. $\sqrt{d_K}$ serves normalization purposes. A transformer consists of two main parts, each made of identical stacked layers:

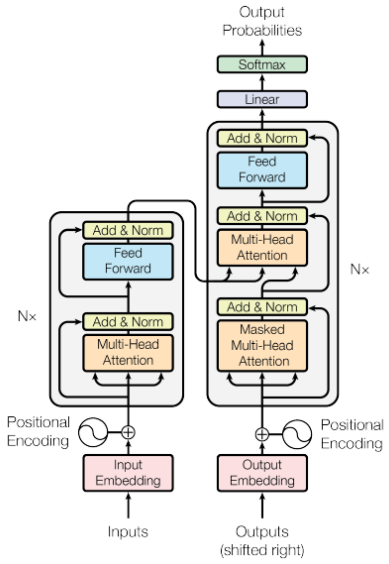

Figure 1.4: Transformers architecture. Source: "Attention is all you need" paper.

- **Encoder:** It processes the input sequence to produce a set of context-aware representations; thanks to the attention mechanism, it is able, for each token, to capture context information relative to all other tokens in the input sequence.
- **Decoder:** It generates the output one token at a time, leveraging the encoder stack's output in its attention mechanism.

There are different types of attention:

- **Encoder self-attention:** Every token is encoded based on all other tokens in the same sequence; this step helps capture dependencies and relationships between tokens, regardless of their positions.
- **Decoder masked self-attention:** In the decoding step, when conditional generation is done, next token probabilities are computed in parallel to speed up training. The network is designed not to cheat and masks the future output tokens during this attention phase.

- **Decoder cross-attention:** In this step, the queries Q that come from the previous decoder are aligned with the information from the encoder. In particular, the output of the last encoder layer is used to build a set of attention vectors K and V.

The encoder and the decoder share common components, as explained by Xiao et al. in the paper "Introduction to Transformers: An NLP Perspective" [8]:

- **Input embeddings and positional encoding:** Every input token is first converted to a high-dimensional vector of real numbers. These vectors are then adjusted to account for the position of tokens within the sentence, a critical aspect of sequence modelling that plays a crucial role in conveying its meaning.

- **Multi-head attention:** In this process, the attention computation previously described is done in parallel by many attention heads. It is done to capture information from many points of view. The results of each head are then concatenated and multiplied by a matrix W^O to produce the output.

- **Feed-forward network:** Every encoder and decoder layer has a feed-forward network that processes each position in the sequence independently. Apart from the softmax function, the attention mechanism is inherently linear; thus, feed-forward networks are used to introduce non-linearity, enabling the model to understand non-linear relations in the input sequence.

- **Add and normalize:** After each attention and feed-forward step, the original information is reintroduced to help the model retain important details, which was an important problem of RNNs. Normalization is done to provide stability; it centres token embeddings around the origin, ensuring they have a mean of 0 and a standard deviation of 1.

1.2.3 Transformers architecture's variants

Transformers models have been widely adopted for various tasks, from text translation and summarization to text generation and question answering (QA). Depending on the use cases, variants of the original transformers architecture have been developed; the best known are encoder-only and decoder-only models. The original transformer architecture, also known as **sequence-to-sequence**, differs from its variants in its attention mechanism since encoder-only and decoder-only models do not have cross-attention. The most common tasks that these models solve are summarization and text translation; many well-known models, such as T5 [9] and BART [10] use this kind of architecture. **Encoder-only** models focus on transforming input data into a compact, meaningful representation; in particular, they take text as input and output a fixed-length vector that represents the input text's meaning. Encoder-only models are an essential part of many RAG pipelines since they compute the similarity between text specified by the user and chunks of text from the knowledge base. **Decoder-only** models focus on generating outputs based on given input: they operate by predicting the next most probable token in an autoregressive manner given the preceding text, continuing recursively until they generate a special token known as the "end token".

1.3 Fine-tuning

The architecture model is initialized with random weights that must be trained to understand natural language. Depending on the number of parameters, training can be a long and expensive process; therefore GPT [11] and BERT [12] propose a training process split into two phases. The first phase is called *pretraining* and is done only once. During this phase, the model is trained on a large corpus of text in a self-supervised way, performing various tasks such as:

- Finding the right value for a masked token in a sentence.
- Given a sentence with mixed tokens, the model has to reconstruct the original sequence.

- Given two sentences, understand if one can follow the other.

After this process, the model can be specialized on a specific task in a phase called *fine-tuning*. It is a process that usually requires structured data in the form of input-output samples for tasks such as summarization or QA, and it is done in a supervised way. Moreover, models can also be specialized in specific domains, such as medicine or law, or on private data unavailable to pretrained models. Even though the amount of data needed to fine-tune is much lower compared to pretraining, high-performing models often have billions of parameters, which can be problematic even when loading in most commercial GPUs, let alone fine-tuning them. To overcome these issues, quantization techniques have been devised to limit memory usage during inference, such as 8-bit quantization explained by Dettmers et al. [13], that allow the loading of the model's weights in a lower precision while limiting its performance degradation. However, full fine-tuning requires memory not only for model storage, as in the case of inference, but also for other parameters necessary for the training phase, such as gradients, optimizer's parameters, and temporary memory necessary for the computation. To solve this issue, parameter-efficient fine-tuning (PEFT) techniques are used, which freeze the model's weights and only update a subset of its parameters, such as particular layers or components.

1.3.1 LoRA

Low-Rank Adaptation (LoRA) [14] is a PEFT technique a technique that reduces the number of parameters needed for training, that has gained significant popularity in recent research due to its efficiency and effectiveness. Inspired by techniques such as Singular Value Decomposition, it aims to decompose the matrices of the model into lower-rank matrices while capturing the most significant features of the data. In particular, it introduces a pair of rank-decomposition matrices A and B

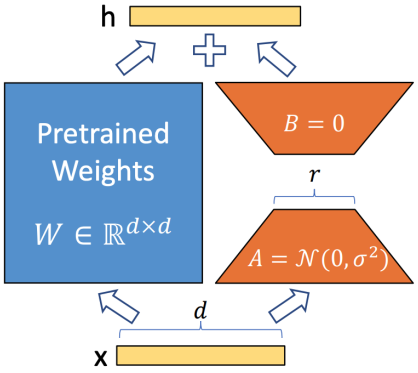

Figure 1.5: Illustration of LoRA technique. Source: LoRA paper.

while keeping the model's parameters frozen. These matrices are set so that their multiplication has the same dimensions as the weights' matrix they are intended to modify. A and B are then updated during the fine-tuning process while the model's weights remain frozen, and they are utilized to compute a ΔW matrix that contains the fine-tuning adjustments. During inference, A and B are multiplied, and their product is summed up in the W matrix, effectively updating the model's weights. This process also allows fine-tuning the same model on different tasks, producing a set of matrices for each task; these matrices have a limited impact on memory usage and can be swapped depending on the task that needs to be performed.

1.4 Prompt engineering

When querying a model, how the task is explained and, more generally, the structure of the messages can significantly impact its response. A prompt is a set of instructions given to the model that specifies the context of the conversation; to guide a model towards a specific output, it is possible to enrich the instructions and the data contained in the prompt with examples of the task that is asked to be performed. There are many ways to structure a prompt:

- **Zero-shot:** It is the simplest case where the prompt includes only the instruction and the data.
- **One-shot:** An example of the task that is asked to be performed is added to the prompt.
- **Few-shot:** Same as one-shot prompting but with few examples.

In addition, there are more complex ways to prompt a model, such as techniques that aim to guide its reasoning, like Chain-of-Thought, which will be explored more in-depth in the next chapter. Working on the prompts has shown effectiveness in many scenarios, with results comparable to fine-tuning in some tasks.

1.4.1 Prompt templates

Although powerful, standard LMs may not always generate responses that align with user intentions, especially if the prompt is unclear or the task requires specific guidance. Standard models are trained on large amounts of data regarding many fields, but they are trained mainly on free-text data, and this is why they may not yield great results in instruction-following tasks. To address this problem, many models are additionally fine-tuned on instruction-following data, using a specific prompt to distinguish the instruction from the data. Prompt templates are divided into various sections, such as:

- **System:** Used to indicate a specific instruction to the model, directing their behaviour and ensuring that the generated outputs align with the intended goals.
- **User:** Contains queries and data the user provides to the model.
- **Assistant:** Contains a response given by the model. If put at the end of the prompt, the model starts generating the response.

The answering quality of LMs used for instruction-following can also be enhanced by providing an expert identity that the model must follow. Xu et al. [15] leveraged in-context learning abilities of LLMs to produce detailed descriptions of the expert identity suitable for the specific instruction that needs to be performed. Their work shows how, even for fine-tuned models, the quality of the output, in terms of how comprehensive, helpful and detailed the responses are, can vary by including a specific identity aligned with the query's purpose in the prompt.

1.5 Large Action Models

LLMs have demonstrated remarkable proficiency in generating and understanding human language. Nonetheless, LMs are natively text-in-text-out, meaning they take text as input and produce text as output. Since they can analyse how some tasks need to be done and explain how to do them, LAMs

attempt to enable these models to execute actions themselves, pushing the boundaries of what they can achieve by enabling them to perform actions beyond text-based responses. Additionally, as Qu et al. showcase [16], tool learning is beneficial across several domains and can mitigate some of the limitations of LLMs.

1.5.1 Tool learning importance

The benefits of tools' usage are multifaced and concern many fields. One significant limitation of LLMs is that they are bounded by the extent of knowledge learnt during their training phase and usually have a knowledge cutoff date, indicating that the model is unaware of information after that date. The use of external APIs enables us to overcome this limit by allowing the model to access updated information; these APIs can provide updates on weather conditions, forecasts and geographical data, to name a few, enabling the use of LLMs in specific applications for fields that require updated data. Another known limit of LLMs is their ability in specific fields, such as the mathematical field, where many models show modest results. Even if they excel in basic operations such as addition or multiplication, tasks regarding divisions, exponentiation and logarithms pose similar challenges humans face. To address this, LLMs can rely on specific tools for this purpose, such as calculators, as humans would do. With external tools facilities, LLMs can be used for automating tasks such as scheduling and setting remainders, and they could even outperform many personal assistants on smartphones, which cannot interact with most applications. Tools could also enhance the interpretability of LLMs' thought process; through tool selection, LLMs can exhibit steps of their reasoning, making it possible for the user to evaluate it and identify where an error could have happened. This makes these models less opaque and facilitates the verification of the responses' correctness.

1.5.2 Recent products in the LAM field

The excitement for LAMs and their potential has led companies to develop products in this field. Despite the general interest in AI agents, the excitement

can only sustain itself for so long before people start demanding tangible results. Two of the most well-known products in this field are Rabbit R1 and Devin AI.

Rabbit R1 Rabbit R1 [1] is a device that communicates to the user through voice commands and is alleged to act as a personal assistant that allows users to perform actions that normally require interaction with a smartphone, such as ordering food delivery or requesting taxi rides. The exceptional demo launch of this product impressed many, including Microsoft's CEO, who called it the "most impressive demo since Steve Jobs' iPhone Unveiling".

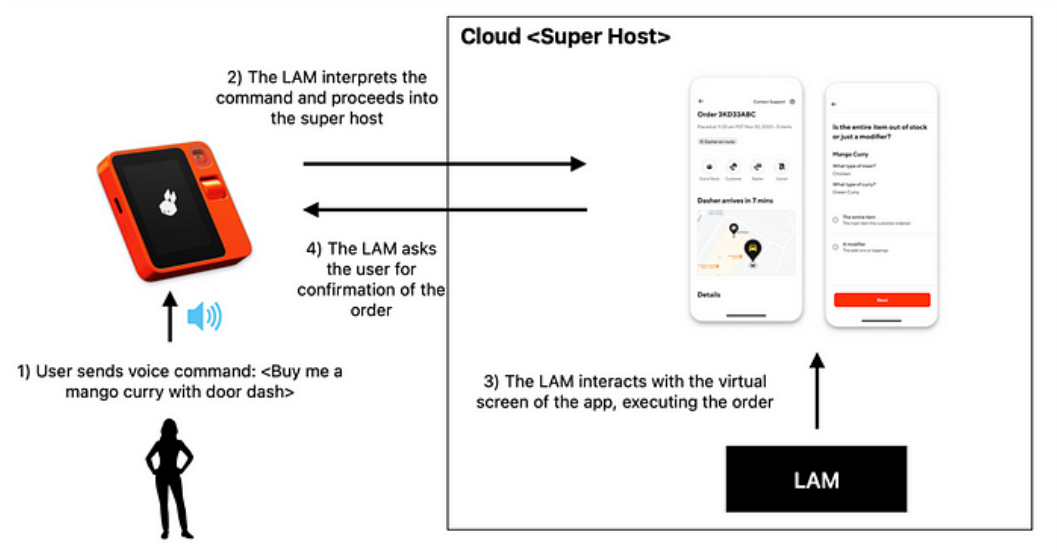

Figure 1.6: Rabbit R1 alleged functioning. Source: Medium's article "The Rabbit R1, AI's First Big Scam?".

The device was presented as a LAM that understands the user's intentions, but instead of creating an API request to send to the selected applications, it interacts directly with its interfaces, just like a human would, but in a virtual environment, as shown in Figure 1.6. However, when the product was launched, it soon became clear that the functioning of the device was quite different from what was initially promised: it was actually a neurosymbolic system, a combination of a neural network which interprets the voice command and

[1] `https://www.rabbit.tech/`

the model's response to the user, and human-crafted, pre-fixed code scripts that guide the actions of the LAM over the app screens. This is much more similar to a robotic process automation implementation than to a full-fledged LAM, and the main problem with this approach is that a slight change in the applications' user interface breaks its functioning completely.

Devin AI Devin AI[2] was presented as the "first AI software engineer", a model that, given a request, is able to leverage multiple tools such as web search, command line and code editors to satisfy it. Its preview shows how it can be used to solve tasks from UpWork, a platform for freelance programmers, implying that it can handle complex problems typically assigned to expert programmers. That demo was proven to be fake since it did not solve the UpWork task shown in it, and even the original poster of the task stated so. Devin AI has not yet been released to the public, so there is uncertainty about its real capabilities. Nonetheless, open-source alternatives were developed in an attempt to tackle these kinds of problems that require advanced reasoning capabilities and tool usage. A benchmark, called SWE-Bench [17], was developed to test agents on solving GitHub issues from popular repositories, a task in which even the state-of-the-art models have modest results. Yang et al. developed SWE-Agent [18], an agent able to exceed performances obtained by state-of-the-art LLMs in this field. Since applications' interfaces are designed for human usage, it focuses on developing agent-computer interfaces designed to be used effectively by models. This approach enables models to perform actions in a way that is simple and easy to understand. SWE-Agent obtained a pass@1 rate of 12.5% on SWE-Bench, and although the result is remarkable, there is still room for improvement.

1.5.3 Agentic RAG

Since LMs rely solely on their internal knowledge to generate text, this can limit the accuracy and depth of their responses. RAG enables LMs to use external knowledge by enriching the input query with relevant data before the

[2] `https://devin.ai/`

model processes it. Agentic RAG leverages tool use and advanced reasoning to introduce a layer of intelligence by employing agents to extend traditional RAG, whose pipeline is shown in Figure 1.7. When RAG is considered a tool, LLMs

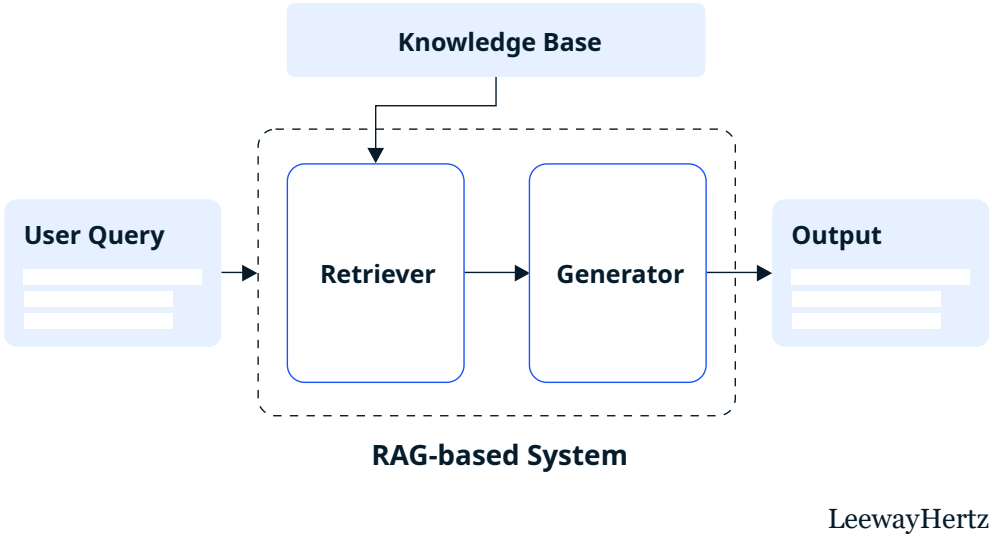

Figure 1.7: Traditional retrieval-augmented generation pipeline. Source: www.leewayhertz.com.

can use different RAG pipelines depending on the query. For example, given some text about scientific papers, the model can either choose to retrieve some relevant chunks of text aligned with the query's text or provide a summary of a specific paper when the query seeks general information about it, as shown in Figure 1.8. Complex queries can also be divided into parallelizable subqueries, each of which can be executed across multiple RAG pipelines that utilize different data sources. The responses from these pipelines are then used to produce the final answer, as shown in Figure 1.9.

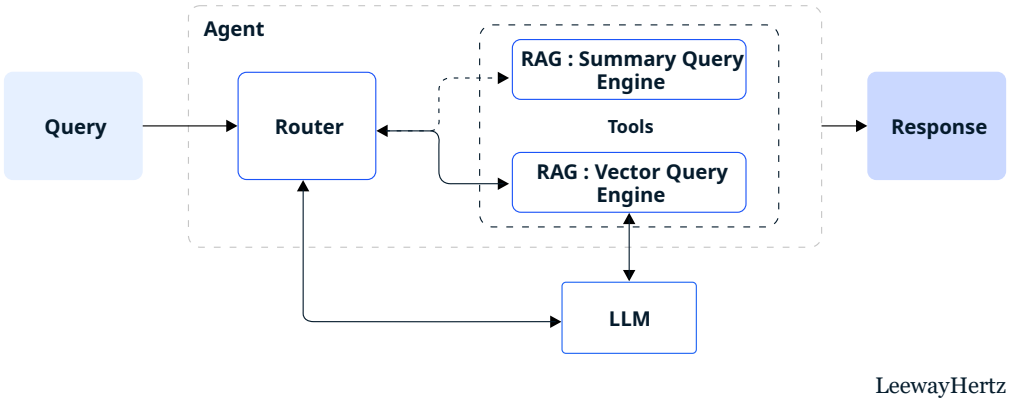

Figure 1.8: Retrieval-augmented generation routing agent. Source: www.leewayhertz.com.

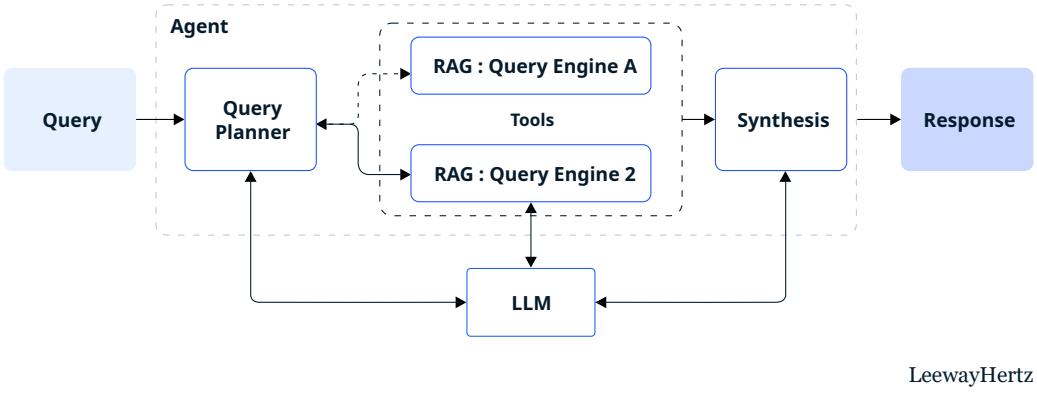

Figure 1.9: Retrieval-augmented generation query planning agent. Source: www.leewayhertz.com.

Chapter 2

Related Work

2.1 Function calling

Function calling is a mechanism that enables LLMs to connect to external tools and APIs. Recently, the capability to select and execute tools has become a desirable feature of LLMs. Models that obtained great results in this field are mostly closed-source, one for all GPT-4[1]. Using closed-source models for this raises two major problems:

(1) Tools are often passed to models in JSON format, which takes a lot of space regarding tokens. Since most closed-source models are provided as paid services, the cost is typically calculated based on the number of tokens processed; therefore, including tools often results in a higher cost per question. They may also fill a big portion of the context length.

(2) Private tools, particularly their documentation and specifications, may be exposed to companies that own these LMs.

The latest open-source LLMs show an increasing ability in code understanding and in the ability to generate correct function calls, and some of them also support function calling natively. To further enhance this capability, these models can be combined with an external retriever to pre-select tools or include

[1] `https://openai.com/index/gpt-4/`

examples of function calls in the prompt. They can also be fine-tuned on human-annotated or synthetic data to improve performance.

2.1.1 Adding special tokens for function calling

Some works add special tokens related to function calling to the vocabulary and fine-tune the model to make it understand the correct use cases for each token. Chen et al. developed Octopus-V2 [2], a lightweight agent that operates directly on a smartphone's CPU that consists of a fine-tuned version of Gemma-2B [19] on a set of 20 functions that carry out requests that users could ask as if it were an assistant, such as making phone calls or sending messages. A special token for each function allows the model to select the correct tool, with another token indicating the end of a function call after parameter selection. Unlike conventional linguistic tokens, these functional tokens do not possess inherent natural language meaning, as explained in the paper; instead, they represent specific actions, and the model learns their meaning through causal language modelling. This approach effectively transforms the prediction task for function names into a single-token classification among the added functional tokens, enhancing the accuracy of function name prediction while simultaneously reducing the number of tokens required. For smaller models, conducting function calling can be challenging, although they excel at typical language completion tasks; for this reason, predicting complete function names can yield errors, and using special tokens transforms a function calling task into a standard completion task. While this method enabled a small model to obtain an accuracy comparable to bigger models that are not fine-tuned for this purpose, it is not readily extensible to new functions since it would require a fine-tuning step each time a new function is added.

2.1.2 Pairing fine-tuning with retrieval

Even when a model is fine-tuned over a specific set of functions, using a retriever has shown to be highly beneficial. Patil et al. [1] developed a model called Gorilla fine-tuning Llama-2 7B [20] over a corpus of (instruction, API call) pairs. They employ a retriever during training and inference to enrich the

prompts with examples of API calls related to the given query, as shown in Figure 2.1, demonstrating how including a retriever can be beneficial even in fine-tuning, resulting in higher performance. At the same time, they compare

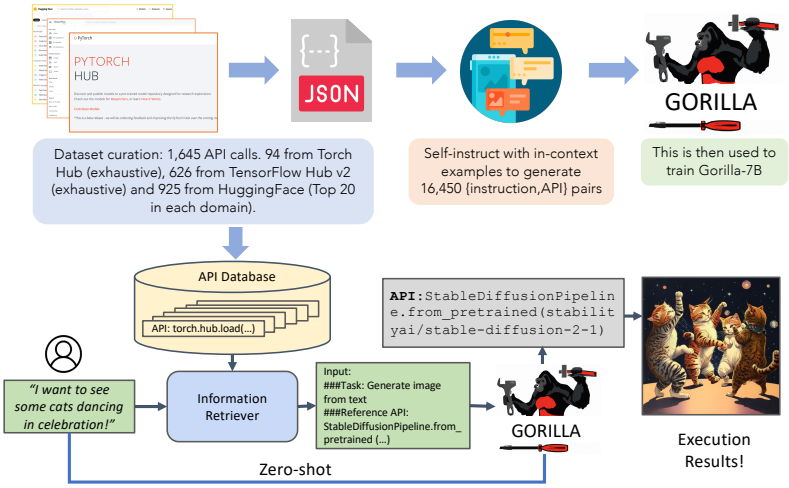

Figure 2.1: Gorilla training and inference procedure. Source: Gorilla paper.

the use of different retrievers and emphasize the impact that using a weaker retriever has on the results. It can misguide the model, resulting in more errors, thus making zero-shot training perform better in certain scenarios. This is why they conclude that when a performing retriever is available, fine-tuning with a retriever has better results, while in other cases, zero-shot fine-tuning could be a better choice. Qin et al. [3] take this approach a step further with ToolLLaMA, developed by fine-tuning both the retriever and the model (Llama-2 7B) on data regarding a specific set of APIs. In this case, the data, synthetically generated from GPT-3.5[2], consists of instructions paired with solution paths made of reasoning steps and API calls. The retriever operates on the documentation of the functions, not on examples of API calls, as in the Gorilla paper. The purpose of retrieval is not to enrich the prompt but rather to provide the model with a set of functions that it can utilize to fulfil the input query. This approach has proven to be beneficial for generalization: the retriever trained on APIs' documentation shows good results in generalizing to unseen functions when provided with their documentation, and Qin et al.

[2] `https://platform.openai.com/docs/models/gpt-3-5-turbo`

demonstrated this by testing their model on APIBench, the dataset on which Gorilla was fine-tuned.

2.2 Data generation

One significant issue in function calling research is the scarcity of specialized datasets. As stated by Wang et al. [21], since most of the available datasets focus on a limited set of APIs, most recent works aggregate APIs from various web sources to leverage the tools produced by human developers that are accessible all over the web, to then produce training data and benchmarks over that set of APIs. There are two main dataset generation options: human-annotated and synthetically generated data.

2.2.1 Human-annotated data

Xu et al. [22] created a tool manipulation benchmark called ToolBench using human-curated examples. They employed a data curation strategy that did not require massive manual example writing. Firstly, templates consisting of goal descriptions and corresponding API calls are created; they contain one or more placeholder pairs that map to a keyword in the goal description and a parameter in the corresponding API call, as shown in Figure 2.2. Random value pools are formed for each keyword, which are then used to instantiate the templates by filling the placeholders. Training examples are made on various tools, each consisting of multiple APIs; given a tool with n APIs, developers are required to create $O(n)$ templates so that each API appears in at least one template. It was found that, on average, a developer takes one day to create templates for a specific tool. Although time-consuming,

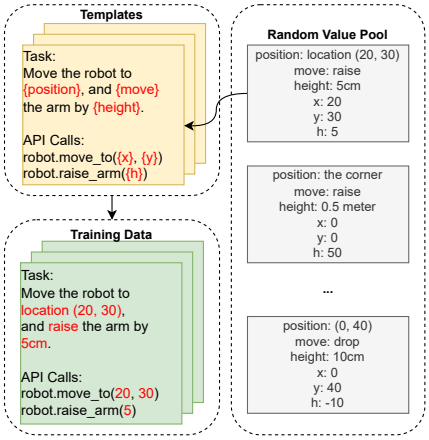

Figure 2.2: Toolbench data generation from templates and random value pools. Source: ToolBench paper.

this approach ensures that the training data is carefully curated by developers, thus reducing the risk of hallucinations since they both write the templates and the random value pools.

2.2.2 Synthetic generation

Other works, such as Octopus-V2 or ToolLLaMA, generate their data synthetically using high-performance, closed-source LLMs such as Google Gemini[3] or OpenAI's GPT models previously mentioned. Although this approach allows the automatic generation of large volumes of data, it can be quite expensive, and inaccuracy must be considered since hallucinations could degrade the quality of the datasets. To address these issues, Octopus-V2 implements a

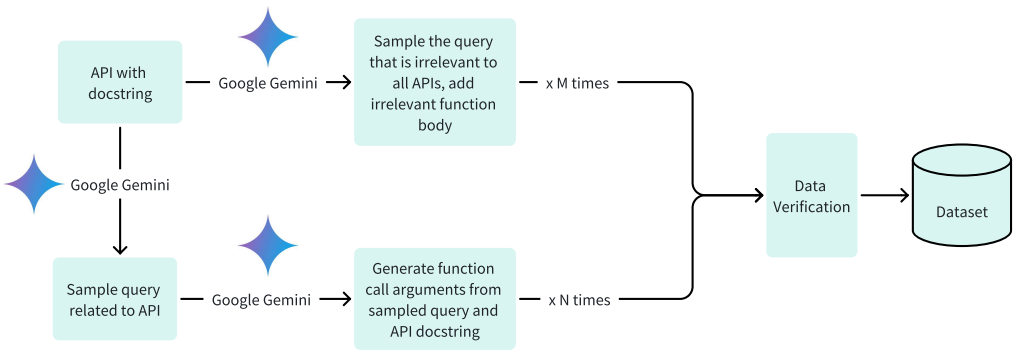

Figure 2.3: Octopus-V2 data generation pipeline. Source: Octopus-V2 paper.

data verification process that enables Google Gemini to detect errors in its generation and correct them by regenerating the incorrect examples, as depicted in Figure 2.3. On the other hand, to build the dataset ToolLLaMA is fine-tuned on, a depth-first search-based decision tree algorithm is implemented to allow GPT-3.5 to explore different possible solutions to a single problem.

2.2.3 In-context learning approaches

Schick et al. developed Toolformer [23], a model trained on function calling in a self-supervised way to perform fine-tuning without relying on human-

[3] `https://gemini.google.com/`

annotated data. API calls are made with a special syntax, and the main idea is to have the model itself, which will be later fine-tuned, generate the entire dataset from scratch, given some human-written examples of how the APIs work. This approach offers the advantage that the model can determine the utility of an API call in a specific context. Once the API calls are added to the textual data, they are executed and filtered based on whether the API call's response reduced the error on the next token predictions, as shown in Figure 2.4.

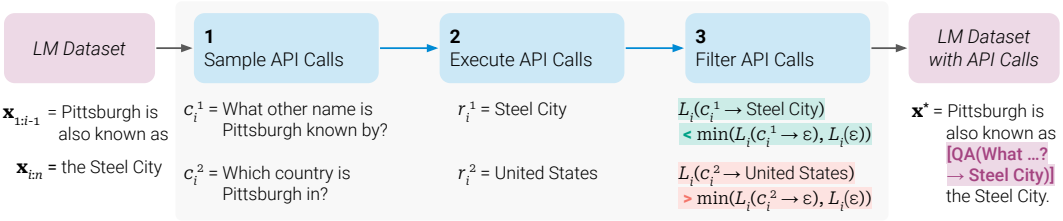

Figure 2.4: Toolformer's dataset generation approach. Source: Toolformer paper.

2.2.4 Training Data Structure

Another big difference between different approaches is the structure of the data used to perform fine-tuning:

- In models that assign special tokens to functions, training data often consists of pairs of (question, function call). In this case, the names of functions in natural language are replaced with their respective tokens in the function calls.
- If a model does not use special tokens and does not support execution, the answers to the questions may consist of lines of code where a specific function is used or single function calls, often in JSON format.
- In models that support and use execution in the training process, training data follow a more complex structure, such as execution steps to reach a certain goal.

2.3 End-to-end retrieval-augmented generation

End-to-end retrieval-augmented generation is a training approach that integrates information retrieval from a knowledge base with response generation based on the retrieved information. When a retriever and an LM are used together in a training pipeline, both may be trained jointly based on the LM's output, or just one of the two components in the pipeline may be trained. While many works have explored the strategy of including a frozen retriever in the training process of an LM to enrich its input with relevant data, Shi et al. did the opposite in their training framework REPLUG [4], treating the LM as a black box. They effectively demonstrate how the LM can supervise the retriever so that it can find the documents that assist the LM in making better predictions. This approach is particularly useful when the gold standard documents for a given query are not provided alongside the correct response. In this case, the LM can guide the retriever through the correctness of its output since the retrieved documents influence it. The accuracy of the final outputs is then factored into the retriever's loss calculation.

2.4 Agents execution pipelines

Thanks to the increasing code-related abilities of LLMs, the most popular NLP frameworks, such as LlamaIndex[4] and Langchain[5], have implemented fully-fledged agents that leverage the tool selection capabilities of the LLM they are based on. One of the most interesting features of these agents is that they can use multiple tools to satisfy the user's requests and can do that using different approaches.

2.4.1 Chain-of-Thought

Chain-Of-Thought [24] is an approach that mimics people's thoughts when solving a complicated reasoning task. Since it is typical to decompose a

[4] `https://github.com/run-llama/llama_index`

[5] `https://github.com/langchain-ai/langchain`

complex task into intermediate tasks before giving the final answer, models are encouraged to generate a coherent series of intermediate reasoning steps that lead to the problem's final answer, called a chain of thought. Prompts include chains of thought demonstrations to encourage this behaviour. It is a particularly useful approach for tackling challenging problems that require logical or mathematical reasoning, and since it provides an interpretable window into the model's behaviour, it can also suggest how it might have arrived at a particular answer by showing its reasoning path.

2.4.2 ReAct

ReAct [25] aims to combine reasoning and action in LLMs to solve complex language reasoning and decision-making tasks. It works by interleaving steps of action execution with reasoning and action selection steps. While Chain-of-Thought is a linear approach, ReAct is a cyclical approach that allows LLMs to correct past decision errors and problems in tools' execution, such as issues related to tools' unavailability or errors in the parameters passed. Many agents

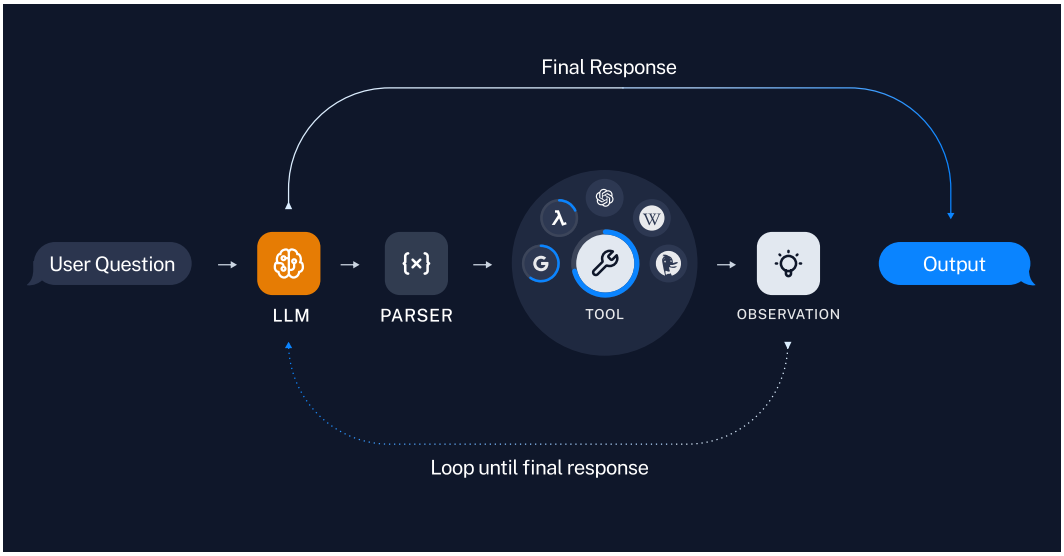

Figure 2.5: ReAct agents' execution loop. Source: LangChain documentation.

in popular NLP frameworks are implemented with a ReAct logic. These agents are based on an LLM that, at each reasoning step, can decide to use a tool

or finish the loop by providing a final answer, as shown in Figure 2.5. These frameworks also implement tool execution, allowing agents to execute the tools selected by the underlying model by parsing and executing function calls from its output. While these agents are very powerful, they require the LLM to be precise in the requests and the tool selection in a specific format, such as JSON. This process can pose challenges for open-source LLMs; in fact, even if they are making big progress in this field and some of them even support function calling natively, remarkable precision has been achieved almost exclusively by state-of-the-art closed-source LLMs until now.

2.4.3 Tree of Thoughts

To generalize over the Chain-of-Thought, Tree of Thoughts [26] enables exploration of various reasoning paths. To do so, multiple chains of thought are built to allow the model to explore different ways to solve a problem. This approach aims to have better results in tasks where Chain-Of-Thought falls short, such as tasks that require exploration or where initial decisions are pivotal. Self-evaluation also allows the model to backtrack when it is clear that the path taken will not achieve the desired result. It is a powerful approach, but it is also difficult to implement and slower compared to alternatives; it can also become costly if applied to closed-source LLMs, often provided as a paid service. TooLLaMA uses a variant of this approach: Since many APIs used in the work can yield errors due to poor maintenance or incorrect arguments, it allows the LLM to explore alternative paths.

Chapter 3

Method

As anticipated in previous chapters, this work proposes using RAG in agents to help the model select tools. The chosen approach is described in this chapter.

3.1 General approach

A retriever operates on a knowledge base formed by the documentation of all available tools. By performing a selection of suitable tools to solve a specific query, the number of tokens in the context length regarding the function declarations is limited. Therefore, it allows using many tools that would saturate the available context length of most models if passed directly to the model. Training a retriever on tool retrieval can enable it to generalize across documentation for tools within the same domain as those it was trained on. The model is not fine-tuned, and its performance is increased by providing documentation for relevant tools with the input query. If the model were fine-tuned directly on the data regarding function calls, it would perform well on the tools it was trained on but would not be adaptable to other tools and would require another training phase each time new tools need to be used. The improving capabilities of recent models in code-related tasks have enabled the possibility of not fine-tuning the model, aiming to achieve correct tool selection on unseen functions by providing a limited set of relevant alternatives.

3.2 End-to-end training process

The retriever is trained using an end-to-end approach, as shown in Figure 3.1, where the retriever is added as a tunable module to an inference model. The model acts as a supervisor for the retrieval module, which has to find

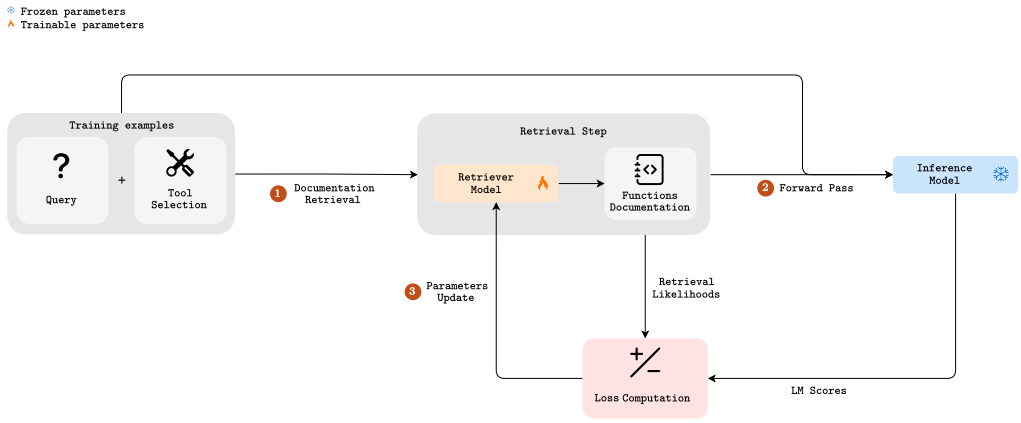

Figure 3.1: Our end-to-end training approach.

documents that increase the quality of the LM's predictions. In the training phase, for each query, k documents, where k is a hyperparameter, are retrieved from the knowledge base and added separately to the prompt. The loss function

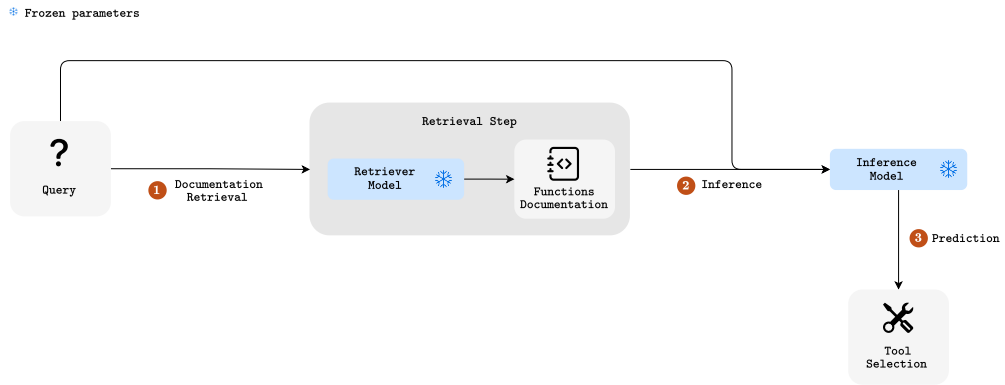

Figure 3.2: Inference pipeline with our approach.

of this algorithm is computed using the KL divergence between the retrieval likelihoods of the documents and the scoring of the LM's output, obtained by

including the retrieved documents individually in the prompt. During inference, one or more tools are retrieved based on the query and added to the prompt to aim the inference model in proper tool selection, as shown in Figure 3.2.

3.2.1 Computing retrieval likelihood

Given a corpus of documents D, an encoder maps each document $d \in D$ to an embedding vector $E(d)$. For each query x, the same encoder is applied, obtaining an embedding $E(x)$ that will be used to compute the retrieval likelihood for documents in D. The similarity between the query embedding and the document embedding is computed by their cosine similarity:

$$s(d, x) = \cos(\mathbf{E}(d), \mathbf{E}(x))$$

For each query, k documents $\mathcal{D}' \subset \mathcal{D}$ with the highest similarity scores are then retrieved. The softmax of the retrieval likelihood of each retrieved document is then computed to get a probability distribution that will be used to compute the KL divergence, as shown in the following equation:

$$P_R(d \mid x) = \frac{e^{s(d,x)/\gamma}}{\sum_{d \in \mathcal{D}'} e^{s(d,x)/\gamma}}$$

where γ is a hyperparameter that controls the temperature of the softmax. Ideally, the retrieval likelihood is computed by marginalizing over all the documents in the corpus $\mathcal{D}$. Even though the number of documents used in our experiments is much smaller compared to the REPLUG paper, to contain the amount of time needed for training, the retrieval likelihood is approximated by only marginalizing over the retrieved documents $\mathcal{D}'$, as proposed in the paper.

3.2.2 Computing LM scores

The output obtained by the model is used to obtain the scores whereby the retrieval likelihood will be compared. The LM score of each document d is computed as follows:

$$Q_{LM}(d \mid x, y) = \frac{e^{P_{LM}(y|d,x)/\beta}}{\sum_{d \in \mathcal{D}'} e^{P_{LM}(y|d,x)/\beta}}$$

where $P_{LM}(y \mid d, x)$ is defined as the LM probability of the ground truth output y given the input x and a document d, and β is another hyperparameter regulating the temperature of the softmax function, similarly to γ for $P_R(d \mid x)$. The REPLUG paper does not specify an implementation for $P_{LM}(y \mid d, x)$, so, given N tokens generated as the model's output, it was decided to compute this quantity as:

$$P_{LM}(y \mid d, x) = -\exp\left(\frac{1}{N}\sum_{i=1}^{N}\ell(y_i, \hat{y}_i)\right) \tag{3.1}$$

where $\ell(y_i, \hat{y}_i)$ is the element-wise cross-entropy loss between y_i, the ground truth token for the i-th position, and $\hat{y}_i$, the model's predicted probability distribution over the vocabulary for that position.

3.2.3 Loss function calculation and parameters update

Given $P_R(d \mid x)$ and $Q_{LM}(d \mid x, y)$, the dense retriever is trained by minimizing the KL divergence between these two distributions, which measures how different two probability distributions are from each other:

$$\mathcal{L} = \frac{1}{|\mathcal{B}|}\sum_{x\in\mathcal{B}} KL\Big(P_R(d \mid x) \parallel Q_{LM}(d \mid x, y)\Big)$$

where $\mathcal{B}$ represents a batch of input queries. Given the scores from the model $Q_{LM}(d \mid x, y)$, the retriever is trained to change its parameters to make the probability distribution of the retrieval likelihoods similar to that of the model's outputs. This is why the minus sign is present in front of the exponential in Equation 3.1: when computing the retrieval likelihood, a more relevant document would have a higher likelihood score; conversely, the mean element-wise cross-entropy loss operates in the opposite manner: the higher the loss, the less accurate the output. After updating the encoder's parameters, the embeddings $E(d)$ are updated for every document $d \in D$. In the REPLUG paper, documents' embeddings are maintained into a FAISS index for fast similarity search, and since this data structure would need an initialization process every time an embedding changes, the embeddings are updated once every 3k steps. In this work, experiments have been performed on a manageable

number of documents, therefore documents are maintained in simpler data structures and their embeddings are updated after every training iteration.

3.3 Datasets structure

The objective of the training is to increase the encoder's accuracy in tool selection; therefore, the data is structured to pair a question with an answer that selects a tool based on the query or solves the query's request using a tool. The queries are designed to be solved by a single tool so that every query has one appropriate document to be retrieved. Since our approach judges single documents separately, if training queries needed multiple tools to be solved, our approach would not be appropriate, and we would need to opt for a different one.

Query	**Response**
Can I find academic research papers on this topic?	ResearchHelper
I want to explore job options in the marketing field in Kyoto.	JobTool
Can you help me analyze a certain NFT collection?	FinanceTool
What's the humidity level in Beijing today?	WeatherTool
I need a pet-friendly vacation rental in Colorado. Can you suggest any?	TripTool

Table 3.1: Training data structured with responses consisting of tool selections only.

Query	**Response**
Can you take a photo using the back camera and save it to the default location?	take_a_photo('back')
Can you please set a timer alarm for 15:00 with the label 'Nap Time'?	set_timer_alarm('15:00', 'Nap Time')
Is it possible to adjust my screen to the lowest brightness, level 0, to save battery?	change_screen_brightness(0)
Can you set the media volume to 5?	set_volume(5, 'media')
How do I disable the Do Not Disturb feature on my device?	enable_do_not_disturb(False)

Table 3.2: Training data structured with responses consisting of function calls.

The responses to the queries could either be the selected tool alone, as Table 3.1 shows, or a standard function call, which includes parameters, as shown in Table 3.2.

Chapter 4

Experimental Setup

This chapter provides a comprehensive explanation of the models and datasets used, as well as the implementation details and metrics chosen to measure performance.

4.1 Enviroment

Loading and training LLMs require a large amount of memory and sophisticated software. Preliminary experiments are carried out leveraging Google Colab's free resources; in particular, machines with an NVIDIA T4 GPU with 14 GB of VRAM and 12.7 GB of system RAM are provided, which, however, have a time limit for their usage. Since the models used for our experiments require higher amounts of VRAM and the training times exceed Google Colab's time limits, servers made available by the UniBo Natual Language Processing Group were used. More specifically, the university has a cluster of four servers; most of the experiments were performed on the main one, which is composed of four NVIDIA GeForce RTX3090 GPUs with 24GB VRAM, 124 GB of system RAM, and an AMD EPYC 7443 24-core processor. The resources made available by the university are shared among students and researchers, therefore SLURM [27], a cluster management technology, is used to allow dynamic scheduling and allocation of jobs. Byobu, a terminal multiplexer, was used to manage detached sessions on the server. Experiments are tracked with Weights and

Biases, a tool that allows real-time logging of various aspects of training runs in the cloud. Furthermore, Docker containers are used to carry out experiments to manage all dependencies and address reproducibility; Figure 4.1 shows the Dockerfile used for our experiments.

```
FROM nvidia/cuda:11.8.0-devel-ubuntu20.04
# Zero interaction (default answers to all questions)
ENV DEBIAN_FRONTEND=noninteractive
# Set work directory
WORKDIR /proj
ENV PYTHON_VERSION=38
# Install general-purpose dependencies
RUN apt-get update -y && \
    apt-get install -y curl \
    git \
    bash \
    nano \
    wget \
    python3.8 \
    python3-pip && \
    apt-get autoremove -y && \
    apt-get clean -y && \
    rm -rf /var/lib/apt/lists/*
RUN pip install --upgrade pip
RUN pip install wrapt --upgrade --ignore-installed
RUN pip install gdown
# Install PyTorch
RUN pip3 install -f https://download.pytorch.org/whl/cu118/torch_stable.html torch
# Install other Python packages
RUN pip3 install --upgrade pandas
RUN pip3 install --upgrade transformers
RUN pip3 install --upgrade trl
RUN pip3 install --upgrade datasets
RUN pip3 install --upgrade wandb
RUN pip3 install --upgrade huggingface_hub
# Required for flash attention
RUN pip3 install -q accelerate
RUN pip3 install bitsandbytes
# Back to default frontend
ENV DEBIAN_FRONTEND=dialog
```

Figure 4.1: Dockerfile used for our experiments.

4.2 Datasets

To gauge our approach, experiments are conducted on two datasets: the first is obtained from data taken from the Huggingface page of the Octopus-V2 model, and the second is derived from a portion of the ToolE dataset, from MetaTool benchmark [28]. All datasets used in this study are divided into a training set and a test set, with 80% of the data allocated to the training set and the remaining 20% allocated to the test set.

4.2.1 Octopus-V2 data

On the Huggingface page of Octopus-V2, the documentation of the 20 tools used is included, which are custom functions performing mobile-related tasks such as sending e-mails, making phone calls and controlling sensors. An evaluation benchmark is also available, showing different models' performance on around 200 examples, which include tool and parameter selection. Examples are parsed from this benchmark, and the knowledge base is constructed by splitting the documentation file to extract each function's documentation separately. Two different datasets were built: one with examples of all the functions in both the training and test set, called *"Octopus-V2 overlapping"*, and one that has examples related to different functions in the training and test set, ensuring no overlap of functions between the two sets, called *"Octopus-V2 non-overlapping"*.

4.2.2 ToolE data

MetaTool benchmark introduces ToolE, a dataset built to evaluate various models. It contains several types of user queries that trigger LLMs to use tools in both single-tool and multi-tool scenarios; the tools used and their documentation are retrieved from OpenAI's plugin list. Our dataset is made by selecting only the user queries solvable with a single tool, obtaining a dataset of around 20k examples and 199 tools. The examples' responses consist of tool selection only, and have the structure shown in Table 3.1. As for the data from Octopus-V2, the tools documentation file is split to separate each tool

documentation from the others, and two separate datasets were built, called *"ToolE overlapping"* and *"ToolE non-overlapping"*.

4.3 Implementation details

To ease the exploration of multiple hyperparameters, Python Data Classes are used to avoid creating multiple files. Training scripts are executed from a bash script in which training parameters are specified and then passed to the Python script. Initially, we attempted to customize Huggingface's Trainer and SFTTrainer APIs to perform our end-to-end training. However, since our approach requires working with two models and tokenizing data in two different ways, and Hugginface's APIs are designed to work with a single model, it was then decided to build a custom training loop using the PyTorch library. We noticed that, even with smaller models, GPU memory accumulated over time, leading to crashes during longer training runs. We incorporated cache memory management inside the training loop to overcome this issue, similar to the implementation in Huggingface's APIs.

Data preprocessing and documents retrieval Firstly, training examples' queries are tokenized using the encoder's tokenizer and divided into batches.

```
for epoch in range(num_epochs):
    for index, batch in enumerate(train_data_loader):
        curr_bs, batch_data = parse_batch(dataset["train"], batch, index)
        embedded_documents = compute_embeddings(retr_model, tokenized_train_documents)
        # documents_per_query contains the indices of the top-k documents for each query
        ↪ in the batch
        # similarities_per_query contains the cosine similarities of the top-k documents
        ↪ for each query in the batch
        documents_per_query, similarities_per_query =
        ↪ get_top_k_docs_per_query(embedded_documents, batch, k)
        prompts = get_prompts(train_documents, batch_data, documents_per_query)
        inner_data_loader = prepare_inner_data_loader(prompts, curr_bs,
        ↪ inner_tokenize_function, inner_data_collator)
```

Figure 4.2: Documents retrieval for each training batch and data preprocessing for the inference model.

At the beginning of every training step, the embeddings of the documents are recomputed, the queries' embeddings are computed, and, through cosine similarity, as explained in Section 3.2.1, k documents for each example are retrieved. Through the batch's index, original query-response pairs are retrieved to build the prompts for the inference model, which is set to evaluation mode since it is not trained. The prompts are then tokenized with the inference model tokenizer, and they are split into k inner batches to maintain the same batch size as the original outer batch, as shown in Figure 4.2. While the outer batches are processed with a standard DataCollatorWithPadding, the inner batches are processed with a DataCollatorForCompletionOnlyLM, which masks the instruction and data to compute the LM scores only on generated tokens, as explained in Section 3.2.2.

Model generation The inference model's generation is done through the forward method, which computes each token in parallel given the previous correct tokens. This generation method was preferred over free generation with *.generate()* method: since the LM scores include element-wise cross-entropy loss computation, every prediction needs to be compared with precisely the token it aimed to predict, which is impossible with free generation. Each i-th forward pass is used to compute the scores for every example in the outer batch, enriched with the i-th selected document for that example, as depicted in Figure 4.3.

```
lm_probs = []
for inner_batch in inner_data_loader:
    inner_batch = {k: v.to("cuda") for k, v in inner_batch.items()}
    labels = inner_batch.pop("labels")
    with torch.no_grad():
        outputs = infer_model(**inner_batch)
    p_lm = get_p_lm_per_sample(outputs, labels, cross_entropy)
    lm_probs.append(p_lm)
```

Figure 4.3: Forward generation on inference model.

Parameters update After the forward passes are done, for each training example in the batch, the softmax function is applied to both the retrieved likelihoods of selected documents and LM scores in order to obtain a pair of tensors $P_R(d \mid x)$ and $Q_{LM}(d \mid x, y)$, that will be used to compute the batch's loss $\mathcal{L}$. Finally, a backward pass is made, and the retriever model's parameters are updated, as shown in Figure 4.4.

```
Pr = compute_Pr(similarities_per_query, gamma)
Q = compute_Q(lm_probs, beta)
loss = compute_loss(Q, Pr, kl_div)
loss.backward()
optimizer.step()
lr_scheduler.step()
optimizer.zero_grad()
```

Figure 4.4: Computation of $P_R(d \mid x)$ and $Q_{LM}(d \mid x, y)$, and retriever model's parameters update.

4.3.1 Loss function variations

In the course of our experiments, we noticed how the mean element-wise cross-entropy loss, used to compute $P_{LM}(y \mid d, x)$, as depicted in Equation 3.1, would take on values in the range $[10^{-4}, 2 \times 10^{1}]$, depending on the correctness of the output. Applying the exponential function to these values increases their difference. In fact, given two real positive numbers ε and M, where $\varepsilon << M$, applying the exponential function to both of them would make M increase much more than ε, resulting in $M - \varepsilon << exp(M) - exp(\varepsilon)$. This high difference between values in the same tensor may cause rounding errors when applying the softmax function, leading to zeros in the probability distributions $P_R(d \mid x)$ and $Q_{LM}(d \mid x, y)$. In such cases, the KL divergence would fail since it does not expect zeros in the probability distributions. This problem is likely to appear in two circumstances:

- When the temperature is lowered, because dividing $P_{LM}(y \mid d, x)$ by a number in the range $]0, 1[$ when applying the softmax function would highlight this difference even more.

- When the training example has responses that include tool selection only. Since each token prediction is made from the previous ground truth tokens, if the document is wrong for a given query, the model has to completely guess the tool name, resulting in high errors. But if the response is a function call, guessing the tokens related to the function call's parameters given the tool name (that appears in the previous ground truth tokens) and the input query is a much easier task compared to the tool name prediction, which results in lower errors and, therefore, a lower mean error over the whole response prediction.

To address this issue, experiments were conducted using an alternative loss function to calculate $P_{LM}(y \mid d, x)$ as follows:

$$\tilde{P}_{LM}(y \mid d, x) = -\frac{1}{N} \sum_{i=1}^{N} \ell(y_i, \hat{y}_i)$$

The rationale is the same as Equation 3.1: predictions with higher loss values result in lower LM scores on the prediction of the ground truth output y given an input context x and a document d, but in this case, the use of the exponential is avoided not to increase differences in prediction errors. A comparison between the two formulas in terms of performance is provided in the next chapter.

4.4 Models

Our experiments employed various models: in particular, we tested two models as inference models and two as retriever models. Our approach was initially implemented using Llama 3 8B as the inference model and BGE-base as the retriever model. Llama 3 8B, Meta's latest state-of-the-art model, achieved remarkable performance among models with a similar number of parameters. BGE-base was selected as the retriever model due to its performance on the MTEB leaderboard [29]. In the experiments performed on the data obtained from Octopus-V2, given the dataset structure, we noticed how BGE's baseline was close to the maximum possible values, and the limited number of training examples available was not enough to show a noticeable improvement. Therefore, RoBERTa-base [30] was also used to highlight a noticeable improvement. As an

inference model, Phi-3 7B [31] was also employed to ensure that the differences in the values of $P_{LM}(y \mid d, x)$, as discussed in Section 4.3.1, were not solely attributed to Llama 3's conservatism in its predictions, which could potentially be a model-specific issue. Both inference models were used in their fine-tuned version for instruction-following tasks, and they were used with their specific prompt template. As for the script parameters, prompt templates are kept in a separate file and selected through parameters not to modify the script file every time a different model is used.

4.5 Evaluation metrics

To evaluate the results obtained, Rank@n is used, which is a binary evaluation metric commonly used in information retrieval to assess the effectiveness of a retrieval system in locating relevant documents within a list. It measures whether the relevant document for a given query is found among the top k retrieved documents. For every example e, Rank@n is defined as:

$$Rank@n(e) = \begin{cases} 1 & \text{if the relevant document is among the top } n \text{ retrieved documents,} \\ 0 & \text{otherwise.} \end{cases}$$

To aggregate these individual results across a test set E, the Rank@n metric is reported as a percentage that indicates how many times the correct document is retrieved among the first n selected documents:

$$Rank@n = \frac{1}{|E|} \sum_{e \in E} Rank@n(e) \times 100\%$$

Given k retrieved documents for each query, Rank@1 ... Rank@k are logged.

4.6 Hyperparameters

For the sake of replicability, complete hyperparameters are provided in Table 4.1. Given the limited dimension of the Octopus-V2 dataset, a grid search was performed, trying all possible combinations of hyperparameters. Table 4.2 illustrated the prompts used on the two datasets; as instruct models are used,

the prompts use the structure of system message, user message and assistant message, which are then properly formatted using the prompt template specific to each model.

Hyperparameter	**Search space**
Octopus-V2 datasets	
gamma_value	{0.3, 0.5, 1.0}
beta_value	{0.3, 0.5, 1.0}
retrieved_docs_per_query	{3, 6, 9}
num_train_epochs	{5, 10, 15}
batch_size	8
learning_rate	1e-5
lr_scheduler_type	cosine
ToolE datasets	
gamma_value	1.0
beta_value	1.0
retrieved_docs_per_query	3
num_train_epochs	10
batch_size	8
learning_rate	1e-5
lr_scheduler_type	cosine

Table 4.1: Explored hyperparameters along with their empirical search grid.

Hyperparameter exploration was done on both the Octopus-V2 datasets, both with the original loss function and the approximated loss function. As expected, the range of explored values of β, which represents the temperature of the softmax function applied to compute $Q_{LM}(d \mid x, y)$, caused many runs done with the original loss functions to crash for the reasons explained in Section 4.3.1. Picked values are shown in Table 4.3.

1	***Octopus-V2 datasets***
	### System message: Given a list of functions with their documentation, call the correct function with the correct parameters in the form function_name(parameter 1, parameter 2). Do not add any other text apart from the function call. Example: Can you add a note saying 'Remember the milk'? Response: add_note('Remember the milk'). Here is the documentation of all the functions. {{retrieved_document}} ### User message: Query: {{query}} Response: ### Assistant message: {{response}}
2	***ToolE datasets***
	### System message: You are a helpful AI assistant. Your current task is to choose the appropriate tool to solve the user's query based on their question. I will provide you with the user's question and information about the tools. If there is a tool in the list that is applicable to this query, please return the name of the tool (you can only choose one tool). If there isn't, please return 'None.'. List of Tools with Names and Descriptions: {{retrieved_document}} ### User message: {{query}} ### Assistant message: {{response}}

Table 4.2: Prompts used during the course of our experiments.

	Original loss ($P_{LM}(y \mid d, x)$)		**Modified loss** ($\tilde{P}_{LM}(y \mid d, x)$)	
Hyperparameter	*Octopus-V2 overlapping*	*Octopus-V2 non overlapping*	*Octopus-V2 overlapping*	*Octopus-V2 non overlapping*
gamma_value	0.3	0.3	0.5	0.5
beta_value	1.0	1.0	0.3	0.5
retrieved_docs_per_query	3	3	3	6
num_train_epochs	10	15	15	5

Table 4.3: Picked hyperparameters values.

Chapter 5

Results and Discussion

5.1 Presentation of Results

Our core results obtained on both Octopus-V2 and ToolE datasets are presented in Table 5.1 and Table 5.2, respectively.

(a) Results obtained using standard loss $P_{LM}(y \mid d, x)$

Model	**Dataset**	**Rank@1**	**Rank@2**	**Rank@3**
RoBERTa-base	Octopus-V2 overlapping	22.5	27.5	30.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 overlapping	82.5	95.0	100.0
RoBERTa-base	Octopus-V2 non-overlapping	40.0	50.0	50.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 non-overlapping	77.5	90.0	95.0

(b) Results obtained using modified loss $\tilde{P}_{LM}(y \mid d, x)$

Model	**Dataset**	**Rank@1**	**Rank@2**	**Rank@3**
RoBERTa-base	Octopus-V2 overlapping	22.5	27.5	30.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 overlapping	85.0	100.0	100.0
RoBERTa-base	Octopus-V2 non-overlapping	40.0	50.0	50.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 non-overlapping	75.0	97.5	97.5

Table 5.1: Results obtained on Octopus-V2 datasets.

Our experiments evaluate the effectiveness and generalization obtained using RAG in tool selection and function calling tasks. All experiments consist of fine-tuning a retriever model on different datasets using Llama 3 8B as the

Model	Dataset	Rank@1	Rank@2	Rank@3
BGE-base	ToolE overlapping	51.928	63.037	68.227
BGE-base fine-tuned (Ours)	ToolE overlapping	81.082	86.345	87.339
BGE-base	ToolE non-overlapping	39.567	53.683	61.977
BGE-base fine-tuned (Ours)	ToolE non-overlapping	2.067	6.939	9.886

Table 5.2: Results obtained on ToolE datasets.

(a) Results obtained using standard loss $P_{LM}(y \mid d, x)$

Model	Dataset	Rank@1	Rank@2	Rank@3
RoBERTa-base	Octopus-V2 overlapping	22.5	27.5	30.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 overlapping	82.5	95.0	100.0
RoBERTa-base	Octopus-V2 non-overlapping	40.0	50.0	50.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 non-overlapping	77.5	90.0	95.0

(b) Results obtained using modified loss $\tilde{P}_{LM}(y \mid d, x)$

Model	Dataset	Rank@1	Rank@2	Rank@3
RoBERTa-base	Octopus-V2 overlapping	22.5	27.5	30.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 overlapping	85.0	100.0	100.0
RoBERTa-base	Octopus-V2 non-overlapping	40.0	50.0	50.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 non-overlapping	72.5	95.0	100.0

Table 5.3: Results obtained on Octopus-V2 datasets retrieving 3 documents.

inference model. The retriever's performance was improved on all overlapping datasets, which have examples of the same functions in both the training and test sets. This shows the learning abilities of the end-to-end training approach used. An improvement was also noticed on the Octopus-V2 non-overlapping dataset but not on the ToolE non-overlapping dataset, where the retrieval performance worsens regardless of the model used. Performance was measured by considering the mean of Rank@1, Rank@2, and Rank@3 to choose the most performing model on each dataset. For runs where the hyperparameter *retrieved_docs_per_query* has higher values, Ranks up to the value of this hyperparameter are logged. For completeness, results are also presented separately for runs with the same number of retrieved documents for

(a) Results obtained using standard loss $P_{LM}(y \mid d, x)$

Model	Dataset	Rank@1	Rank@2	Rank@3	Rank@4	Rank@5	Rank@6
RoBERTa-base	Octopus-V2 overlapping	22.5	27.5	30.0	37.5	50.0	52.5
RoBERTa-base fine-tuned (Ours)	Octopus-V2 overlapping	80.0	92.5	95.0	97.5	100.0	100.0
RoBERTa-base	Octopus-V2 non-overlapping	40.0	50.0	50.0	52.5	57.5	62.5
RoBERTa-base fine-tuned (Ours)	Octopus-V2 non-overlapping	72.5	82.5	90.0	92.5	97.5	97.5

(b) Results obtained using modified loss $\tilde{P}_{LM}(y \mid d, x)$

Model	Dataset	Rank@1	Rank@2	Rank@3	Rank@4	Rank@5	Rank@6
RoBERTa-base	Octopus-V2 overlapping	22.5	27.5	30.0	37.5	50.0	52.5
RoBERTa-base fine-tuned (Ours)	Octopus-V2 overlapping	82.5	100.0	100.0	100.0	100.0	100.0
RoBERTa-base	Octopus-V2 non-overlapping	40.0	50.0	50.0	52.5	57.5	62.5
RoBERTa-base fine-tuned (Ours)	Octopus-V2 non-overlapping	75.0	97.5	97.5	100.0	100.0	100.0

Table 5.4: Results obtained on Octopus-V2 datasets retrieving 6 documents.

(a) Results obtained using standard loss $P_{LM}(y \mid d, x)$

Model	Dataset	Rank@1	Rank@2	Rank@3	Rank@4	Rank@5	Rank@6	Rank@7	Rank@8	Rank@9
RoBERTa-base	Octopus-V2 overlapping	22.5	27.5	30.0	37.5	50.0	52.5	55.0	55.0	55.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 overlapping	75.0	87.5	92.5	95.0	95.0	97.5	100.0	100.0	100.0
RoBERTa-base	Octopus-V2 non-overlapping	40.0	50.0	50.0	52.5	57.5	62.5	62.5	70.0	72.5
RoBERTa-base fine-tuned (Ours)	Octopus-V2 non-overlapping	62.5	85.0	90.0	92.5	92.5	92.5	100.0	100.0	100.0

(b) Results obtained using modified loss $\tilde{P}_{LM}(y \mid d, x)$

Model	Dataset	Rank@1	Rank@2	Rank@3	Rank@4	Rank@5	Rank@6	Rank@7	Rank@8	Rank@9
RoBERTa-base	Octopus-V2 overlapping	22.5	27.5	30.0	37.5	50.0	52.5	55.0	55.0	55.0
RoBERTa-base fine-tuned (Ours)	Octopus-V2 overlapping	82.5	97.5	100.0	100.0	100.0	100.0	100.0	100.0	100.0
RoBERTa-base	Octopus-V2 non-overlapping	40.0	50.0	50.0	52.5	57.5	62.5	62.5	70.0	72.5
RoBERTa-base fine-tuned (Ours)	Octopus-V2 non-overlapping	67.5	92.5	95.0	97.5	100.0	100.0	100.0	100.0	100.0

Table 5.5: Results obtained on Octopus-V2 datasets retrieving 9 documents.

each query, considering the mean of all available metrics; results are depicted in Tables 5.3, 5.4 and 5.5.

5.1.1 Loss functions comparison

An alternative version of the loss function was used to allow training on ToolE data, as explained in Section 4.3.1. The alternative loss function was also tested on the Octopus-V2 datasets for testing purposes. It was noticed that using the standard loss function, shown in Figure 5.1, results in a slightly higher oscillation of the metrics values over the training epochs while obtaining a slightly worse performance on average compared to runs that use the modified

loss, as shown in Figure 5.2 and in Table 5.1.

Figure 5.1: Metrics obtained on Octopus-V2 non-overlapping dataset using the standard loss function.

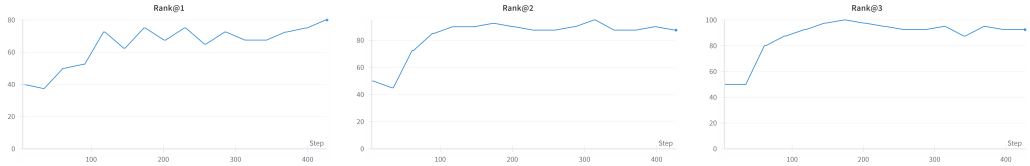

Figure 5.2: Metrics obtained on Octopus-V2 non-overlapping dataset using the modified loss function.

5.2 Discussion

The training run on the ToolE overlapping dataset yielded a remarkable increase in performance, as shown in Figure 5.3. However, on the ToolE non-

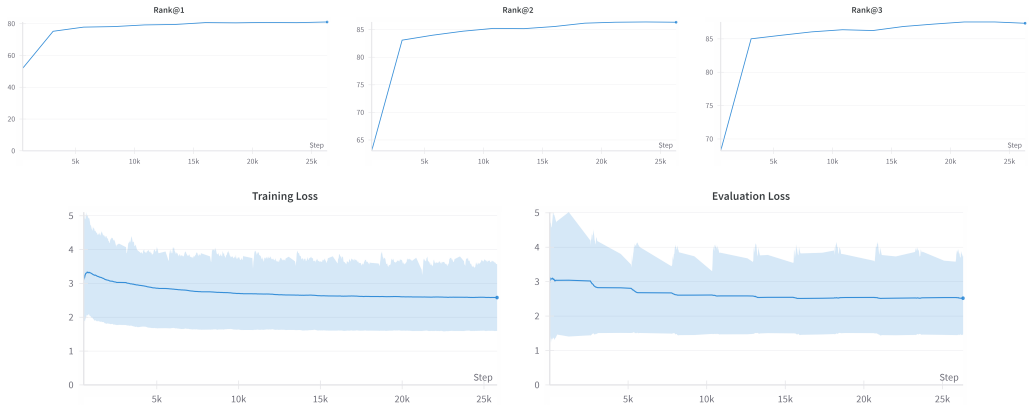

Figure 5.3: Complete metrics obtained on ToolE non-overlapping dataset.

overlapping dataset, there was a gradual decrease in metrics values, as depicted in Figure 5.4, while better performance was obtained on the Octopus-V2 non-overlapping dataset. The loss in the run on the ToolE non-overlapping dataset

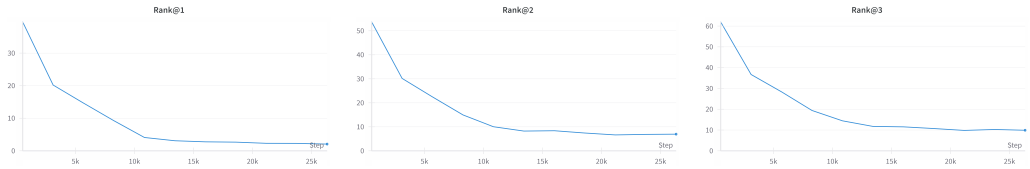

Figure 5.4: Metrics obtained on ToolE non-overlapping dataset.

had a similar decreasing trend to the one shown on the ToolE overlapping dataset in Figure 5.3. Although there was a decrease in the metrics values as well, the problem is not in the training algorithm: since the KL divergence measures the difference between two probability distributions representing LM scores and retrieval likelihoods if LM predictions are inaccurate and retrieval performances get equally inaccurate, the loss values might decrease despite this inaccuracy. To explain the gap in the effectiveness of our approach on the Octopus-V2 data compared to the ToolE data, it is helpful to consider the type of functions included in the two datasets: The ToolE datasets include heterogeneous tools covering many fields, ranging from cybersecurity to web search, from calculation tasks to picture generation. On the other hand, Octopus-V2 datasets include functions that, although they cover many fields, such as sending text messages, making phone calls, or controlling a smart home through sensors, are all about actions that a smartphone assistant would do. Given this consideration, one reason for the poor performance obtained on the ToolE dataset may be the difficulty of generalizing to functions unrelated to those seen during training. Another cause may be the documentation used during training: in the experiments presented, the documentation file of the whole library was passed in both the training and evaluation phases. Since the training on the ToolE dataset is much longer and includes many more examples compared to the one done on Octopus-V2 datasets, the retriever could learn at some point to exclude the functions documentation related to examples that are in the test set and not in the training set, since they are never relevant

to training examples, thus resulting in poor performances on the test set. To verify these hypotheses, another experiment was conducted by splitting the documentation file into two parts: the first part, used during the training phase, includes documentation for functions related to the training examples, while the second part, used during the evaluation phase, contains documentation for functions in the test set. The results are shown in Figure 5.5; an increase in

Figure 5.5: Metrics obtained on ToolE non-overlapping dataset using different documentations for training and test sets.

Rank@1 and Rank@2 can be noticed in the first two epochs, and, although the metrics decrease with increasing epochs, the reduction is only by a few percentage points and does not reach values close to zero as before.

Conclusion and Future Work

This work evaluated RAG integrated into tool selection and function calling tasks. We chose an end-to-end retrieval-augmented generation training approach, a novel approach particularly useful in scenarios where the training data is not labelled with their relevant documents, allowing a straightforward fine-tuning of the retriever. Inspired by the REPLUG framework, which describes this approach in theory, we provided an implementation that elaborates on details of the approach that were not described in detail in the original paper, such as the specific definition of the loss function used. For this reason, we explored the use of two different implementations to compute the loss function in the training algorithm, showing their differences and their results on different datasets. Using multiple datasets and models, we demonstrated how RAG could increase the generalization abilities in agentic tasks and to which extent this is possible. Two types of datasets were mainly used in our experiments, both related to tool selection and function calling tasks in response to an input query. We evaluated the learning abilities of our approach, showing an increase in performance on the tools the retriever was trained on and its generalization capabilities, as well as what happened when the retriever was tested on unseen tools during training. One of the biggest limitations encountered was the lack of data. Since agents are a recent discovery in NLP, a standard for data related to function calling and tool selection is not yet available. Moreover, most existing works use approaches usually related to model fine-tuning on a limited set of tools, providing data in custom formats, including entire code scripts or multiple generation steps. Therefore, many works focused on the synthetic generation of data using various techniques, which were explained in our work. Since all approaches in machine learning rely on data and their availability,

synthetic data generation is a technique that needs further exploration, given its current limitations related mainly to hallucination problems. However, it is a topic that could contribute to LLMs' abilities in tool selection and function calling.

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Gianluca Moro, who allowed me to explore an innovative and interesting topic and whose expertise and guidance were central to the realization of this work. I extend my heartfelt gratitude to my mentors, Giacomo Frisoni and Lorenzo Molfetta, for their support and guidance and for always being ready to provide technical aid and clarifications on the thesis topics. I would also like to acknowledge my family and all the true friends who surrounded me in these years. Your support, love, and belief in me have been a constant source of motivation that has helped me carry on.

Bibliography

[1] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *CoRR*, abs/2305.15334, 2023. doi: 10.48550/ARXIV.2305.15334. URL `https://doi.org/10.48550/arXiv.2305.15334`.

[2] Wei Chen and Zhiyuan Li. Octopus v2: On-device language model for super agent, 2024.

[3] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. *CoRR*, abs/2307.16789, 2023. doi: 10.48550/ARXIV.2307.16789. URL `https://doi.org/10.48550/arXiv.2307.16789`.

[4] Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Rich James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. REPLUG: retrieval-augmented black-box language models. *CoRR*, abs/2301.12652, 2023. doi: 10.48550/ARXIV.2301.12652. URL `https://doi.org/10.48550/arXiv.2301.12652`.

[5] Marco Somalvico. *Intelligenza Artificiale*. Hewlett-Packard, 1987.

[6] Shusen Liu, Peer-Timo Bremer, Jayaraman J. Thiagarajan, Vivek Srikumar, Bei Wang, Yarden Livnat, and Valerio Pascucci. Visual exploration of semantic relationships in neural word embeddings. *IEEE Trans. Vis. Comput. Graph.*, 24(1):553–562, 2018. doi: 10.1109/TVCG.2017.2745141. URL `https://doi.org/10.1109/TVCG.2017.2745141`.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[8] Tong Xiao and Jingbo Zhu. Introduction to transformers: an NLP perspective. *CoRR*, abs/2311.17633, 2023. doi: 10.48550/ARXIV.2311.17633. URL https://doi.org/10.48550/arXiv.2311.17633.

[9] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

[10] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel R. Tetreault, editors, *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 7871–7880. Association for Computational Linguistics, 2020. doi: 10.18653/V1/2020.ACL-MAIN.703. URL https://doi.org/10.18653/v1/2020.acl-main.703.

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz

Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019. doi: 10.18653/V1/N19-1423. URL https://doi.org/10.18653/v1/n19-1423.

[13] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *CoRR*, abs/2208.07339, 2022. doi: 10.48550/ARXIV.2208.07339. URL https://doi.org/10.48550/arXiv.2208.07339.

[14] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

[15] Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. Expertprompting: Instructing large language models to be distinguished experts. *CoRR*, abs/2305.14688, 2023. doi: 10.48550/ARXIV.2305.14688. URL https://doi.org/10.48550/arXiv.2305.14688.

[16] Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. Tool learning with large language models: A survey. *CoRR*, abs/2405.17935, 2024. doi: 10.48550/ARXIV.2405.17935. URL https://doi.org/10.48550/arXiv.2405.17935.

[17] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *CoRR*, abs/2310.06770, 2023. doi: 10.48550/ARXIV.2310.06770. URL https://doi.org/10.48550/arXiv.2310.06770.

[18] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *CoRR*, abs/2405.15793, 2024. doi: 10.48550/ARXIV.2405.15793. URL https://doi.org/10.48550/arXiv.2405.15793.

[19] Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Cristian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, and et al. Gemma: Open models based on gemini research and technology. *CoRR*, abs/2403.08295, 2024. doi: 10.48550/ARXIV.2403.08295. URL https://doi.org/10.48550/arXiv.2403.08295.

[20] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhar-

gava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023. doi: 10.48550/ARXIV.2307.09288. URL `https://doi.org/10.48550/arXiv.2307.09288`.

[21] Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. What are tools anyway? A survey from the language model perspective. *CoRR*, abs/2403.15452, 2024. doi: 10.48550/ARXIV.2403.15452. URL `https://doi.org/10.48550/arXiv.2403.15452`.

[22] Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. On the tool manipulation capability of open-source large language models. *CoRR*, abs/2305.16504, 2023. doi: 10.48550/ARXIV.2305.16504. URL `https://doi.org/10.48550/arXiv.2305.16504`.

[23] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*,

2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

[24] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

[25] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=WE_vluYUL-X.

[26] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html.

[27] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: simple linux utility for resource management. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing, 9th International Workshop, JSSPP 2003, Seattle, WA, USA, June 24, 2003, Revised Papers*, volume 2862 of *Lecture Notes in Computer*

Science, pages 44–60. Springer, 2003. doi: 10.1007/10968987_3. URL https://doi.org/10.1007/10968987_3.

[28] Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *CoRR*, abs/2310.03128, 2023. doi: 10.48550/ARXIV.2310.03128. URL https://doi.org/10.48550/arXiv.2310.03128.

[29] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. MTEB: massive text embedding benchmark. In Andreas Vlachos and Isabelle Augenstein, editors, *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*, pages 2006–2029. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EACL-MAIN.148. URL https://doi.org/10.18653/v1/2023.eacl-main.148.

[30] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL http://arxiv.org/abs/1907.11692.

[31] Marah I Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat S. Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Caio César Teodoro Mendes, Weizhu Chen, Vishrav Chaudhary, Parul Chopra, Allie Del Giorno, Gustavo de Rosa, Matthew Dixon, Ronen Eldan, Dan Iter, Amit Garg, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Jamie Huynh, Mojan Javaheripi, Xin Jin, Piero Kauffmann, Nikos Karampatziakis, Dongwoo Kim, Mahoud Khademi, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Chen Liang, Weishung Liu, Eric Lin, Zeqi Lin, Piyush Madan, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid

Pryzant, Heyang Qin, Marko Radmilac, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Xia Song, Masahiro Tanaka, Xin Wang, Rachel Ward, Guanhua Wang, Philipp Witte, Michael Wyatt, Can Xu, Jiahang Xu, Sonali Yadav, Fan Yang, Ziyi Yang, Donghan Yu, Chengruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone. *CoRR*, abs/2404.14219, 2024. doi: 10.48550/ARXIV.2404.14219. URL `https://doi.org/10.48550/arXiv.2404.14219`.