

# A Study on Deploying Large Language Models as Agents

by

Jiannan Cao

B.S. Computer Science, University of California, Davis, 2015

Submitted to the System Design & Management Program  
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING AND MANAGEMENT

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

© 2024 Jiannan Cao. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Jiannan Cao  
System Design & Management Program  
June 26, 2024

Certified by: John R. Williams  
Professor of Information Engineering and Civil and Environmental Engineering  
Thesis Supervisor

Accepted by: Joan S. Rubin  
System Design & Management Program  
Executive Director

# A Study on Deploying Large Language Models as Agents

by

Jiannan Cao

Submitted to the System Design & Management Program  
on June 26, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING AND MANAGEMENT

## ABSTRACT

This thesis investigates the deployment and utilization of Large Language Models (LLMs) as agents, exploring their potential in automating workflows and enhancing user interactions. The study begins with an in-depth analysis of language models, tracing their evolution from pure statistical models to advanced neural network architectures like Transformers and their bidirectional variants. It then delves into the operational framework of LLM agents, detailing user interactions, environmental considerations, memory management, task planning, and tool use. The study addresses critical limitations in LLM inputs, such as the context window and introduces Retrieval-Augmented Generation (RAG) as a solution to extend the model's capability. Key APIs provided by OpenAI for deploying GPT models are discussed, highlighting their functionalities and applications. Finally, the practical application of LLMs in creating Robotic Process Automation (RPA) workflows is demonstrated through a divide-and-conquer methodology, showcasing the efficiency, scalability, flexibility, and accuracy of this approach. This comprehensive study underscores the transformative impact of LLMs in automating complex processes and enhancing user experiences through intelligent agent deployment.

Thesis supervisor: John R. Williams

Title: Professor of Information Engineering and Civil and Environmental Engineering

# Acknowledgments

I would like to express my deepest gratitude to those who have supported and guided me throughout the journey of completing this thesis.

First and foremost, I am immensely thankful to my thesis supervisor, Professor John R. Williams, for his expert advice and mentorship. His profound knowledge and thoughtful guidance have been pivotal in the successful completion of this work.

I would also like to thank Joan S. Rubin, Executive Director of the System Design & Management Program, for her unwavering support and encouragement. Her dedication to the program and its students has been a constant source of inspiration.

I am profoundly grateful to William Foley and Bryan Moser for their invaluable advice and guidance for the courses. Their insights and expertise have been instrumental in shaping the direction and quality of this journey.

Special thanks go to Ignacio Vazquez for his support on the SDM courses and events.

I also wish to extend my heartfelt appreciation to Professor Zhiyuan Liu for his invaluable suggestions and to Yining Ye and Yujia Qin for their collaborative efforts on the **ProAgent**[1] Project, which lead to a continue discussion on RPA Agent in Chapter 7.

Finally, I would like to acknowledge the unwavering support of my family and friends, whose encouragement and understanding have been my steadfast companions throughout this journey.

Thank you all for your contributions, without which this thesis would not have been possible.

# Contents

<b>Title page</b>	<b>1</b>
<b>Abstract</b>	<b>2</b>
<b>Acknowledgments</b>	<b>3</b>
<b>List of Figures</b>	<b>7</b>
<b>List of Tables</b>	<b>7</b>
<b>1 Introduction</b>	<b>8</b>
<b>2 What is Language Model?</b>	<b>9</b>
2.1 Pure Statistical Language Models . . . . .	9
2.2 Vector Representation of Words/Tokens . . . . .	10
2.2.1 One-Hot Vector Representation . . . . .	10
2.2.2 Word2Vec/Embeddings . . . . .	11
2.3 Text Preprocessing . . . . .	12
2.3.1 The Plain Text . . . . .	12
2.3.2 Tokens . . . . .	12
2.3.3 Token IDs . . . . .	12
2.3.4 Embedding into N-Dimension Vectors . . . . .	12
2.3.5 Summary . . . . .	12
2.4 Recurrent Neural Networks . . . . .	13
2.4.1 Vanilla RNNs . . . . .	13
2.4.2 LSTMs (Long Short-Term Memory Networks) . . . . .	13
2.4.3 GRUs (Gated Recurrent Units) . . . . .	13
2.4.4 Summary . . . . .	13
2.5 Large Language Models . . . . .	14
2.5.1 Transformer Neural Networks . . . . .	14
2.5.2 BERT (Bidirectional Encoder Representations from Transformers) . . . . .	14
2.5.3 GPTs (Generative Pre-trained Transformers) . . . . .	14
2.5.4 Summary . . . . .	14
2.6 Multi-modal Models . . . . .	15

<b>3</b>	<b>What is LLM Agent?</b>	<b>16</b>
3.1	Overview . . . . .	16
3.2	User Interactions . . . . .	17
3.3	Environments . . . . .	17
3.4	Memory . . . . .	17
3.5	Planning . . . . .	17
	3.5.1 Task Decomposition . . . . .	17
	3.5.2 Self-Reflection . . . . .	18
3.6	Tool Use . . . . .	18
3.7	LLM and Agent deployment . . . . .	18
3.8	Agent Frameworks . . . . .	19
<b>4</b>	<b>Input Limitation of the LLMs</b>	<b>20</b>
4.1	Context Window . . . . .	20
	4.1.1 GPT Model Descriptions and Context Window . . . . .	20
4.2	RAG (Retrieval-Augmented Generation) . . . . .	23
	4.2.1 Process . . . . .	23
	4.2.2 Components . . . . .	23
	4.2.3 Advantages . . . . .	23
4.3	Summary . . . . .	23
<b>5</b>	<b>Important OpenAI APIs for GPT Models</b>	<b>24</b>
5.1	Text Completions API . . . . .	25
5.2	Chat Completions API . . . . .	26
	5.2.1 Roles of a Chat Message . . . . .	27
	5.2.2 Example of Chat Messages . . . . .	27
	5.2.3 Training Data for Chat Messages . . . . .	28
	5.2.4 Reason for using Chat Messages . . . . .	29
5.3	Chat Completions API with Function Calling . . . . .	30
	5.3.1 Example of Function Calling . . . . .	31
	5.3.2 Importance of Function Calling . . . . .	35
5.4	Assistants API . . . . .	36
	5.4.1 Features . . . . .	36
	5.4.2 Choosing Between the Chat Completions API and the Assistants API . . . . .	37
5.5	Conclusion . . . . .	38
<b>6</b>	<b>How to Gain Function Calling Capabilities</b>	<b>39</b>
6.1	Prompt Engineering . . . . .	40
6.2	Fine-tuning an LLM . . . . .	41
6.3	LLM + Planner . . . . .	42
6.4	Conclusion . . . . .	43

<b>7</b>	<b>A Divide-and-Concur RPA Agent</b>	<b>44</b>
7.1	Introduction . . . . .	44
7.2	Objective . . . . .	45
7.2.1	Workflow Nodes . . . . .	45
7.3	Methodology . . . . .	47
7.3.1	Steps . . . . .	48
7.4	Result . . . . .	53
7.4.1	Visual Results . . . . .	53
7.5	Conclusion . . . . .	54
<b>8</b>	<b>Conclusion</b>	<b>55</b>
	<b>References</b>	<b>56</b>

# List of Figures

2.1	An example of a statistical language model . . . . .	9
3.1	Components of LLM Agents . . . . .	16
5.1	Text Completions API . . . . .	25
5.2	Chat Completions API . . . . .	26
5.3	Chat Completions API with Function Calling . . . . .	30
5.4	function calling Process regarding the weather in Boston . . . . .	32
5.5	Chat Completions API with Function Calling . . . . .	36
6.1	MRKL System . . . . .	40
6.2	Toolformer . . . . .	41
6.3	TALM (Tool Augmented Language Models) . . . . .	41
6.4	LLM + P . . . . .	42
7.1	RPA Workflow Builder Task Overview . . . . .	45
7.2	RPA Workflow Builder Step (1) . . . . .	48
7.3	RPA Workflow Builder Step (2) . . . . .	50
7.4	RPA Result n8n Workflow . . . . .	53
7.5	RPA Result Gmail . . . . .	53
7.6	RPA Result Slack . . . . .	54

# List of Tables

5.1	An example of chat messages input-output . . . . .	27
5.2	Message History Regarding the Weather in Boston . . . . .	33
7.1	LLM translates (2b) into Slack Node . . . . .	52
7.2	LLM translates (2c) into Gmail Node . . . . .	52

# Chapter 1

## Introduction

The deployment of Large Language Models (LLMs) as agents in various sectors represents a groundbreaking advancement in the field of artificial intelligence (AI).

LLMs, such as OpenAI's GPT[2][3][4][5] series and Google's BERT[6], based on the neural network structure Transformer[7], have demonstrated unprecedented capabilities in understanding and generating human-like text[8], thereby offering transformative potential across a wide range of applications[9]. From enhancing performance in natural language processing tasks to making a meaningful conversation with end users[10], even helping users finish complex tasks[1], LLMs are reshaping the landscape of AI technology and its practical implications.

The advent of LLMs has spurred extensive research and experimentation, leading to notable achievements such as performance improvements in machine inference benchmarks and advancements in agent applications. As the technology continues to evolve, the future of deploying LLMs as agents hinges on addressing these challenges while capitalizing on the opportunities they present. Ultimately, the responsible development and deployment of LLMs as agents offer a promising path forward in harnessing the power of AI to augment human capabilities and drive progress across industries.

These agents utilize a particular feature of LLMs known as function calling[11][12]. The LLM can identify missing information and initiate a function call to retrieve it. Large language models have shown remarkable proficiency in function calling, which has been a significant factor in the increasing role of AI agents in the software industry. Research into AI agents has been vigorous, with advances in thought sequencing and improved prompting methods.

We will conduct an in-depth review of the important OpenAI APIs, specifically focusing on Text Completions, Chat Completions, and Function Calling feature. We will provide a comprehensive explanation of these APIs and their functionalities. Additionally, we will explore the three major methods for re-implementing Function Calling feature from LLMs without it. Building upon these foundational tools for creating Language Model (LLM) Agents, this paper will introduce a novel divide-and-conquer algorithm. This innovative approach is designed to enhance the generation of RPA (Robotic Process Automation) workflows, offering a more efficient and effective method for automating tasks. Through detailed analysis and practical examples, we aim to demonstrate how this new algorithm can be utilized to improve RPA workflow generation, contributing to advancements in automation.



# Chapter 2

## What is Language Model?

First of all, to better understanding Large Language Model (LLM), here we introduce the basic concept of the classical statistical language model and the neural network language model concept. They are important concepts of Natural Language Processing.

### 2.1 Pure Statistical Language Models

Pure statistical language models[13] use statistical methods to predict the likelihood of the next word of a text. For example, the most possible next word of the sentence "The weather in Seattle is always" is "rainy".

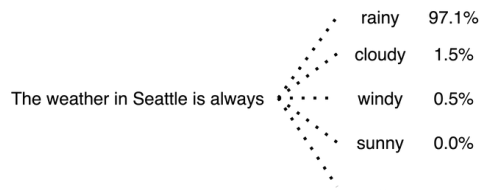


Figure 2.1: An example of a statistical language model

Assume we could express an n-words text as  $w_1w_2 \dots w_n$ , we could use the equation to express this possibility:

$$P(w_m | w_1w_2 \dots w_{m-1}) = \frac{\#(w_1w_2 \dots w_m)}{\#(w_1w_2 \dots w_{m-1})}$$

There is an important variant of the statistical language models: Word n-gram language model, which assumes the possibility is only determined by the previous n-words:

$$P(w_m | w_1, \dots, w_m) \approx P(w_m | w_{m-n}w_{m-n+1} \dots w_{m-1}) = \frac{\#(w_{m-n}w_{m-n+1} \dots w_m)}{\#(w_{m-n}w_{m-n+1} \dots w_{m-1})}$$

We could use basic count-based method to get this model: we could collect all the text on the Internet or books in the library to return the exact statistical result of the possibilities. We could also use machine learning methods to train a model to give the result. Especially the Neural Network Language Models, which we'll introduce in the next section.

## 2.2 Vector Representation of Words/Tokens

We could use neural network methods to build the language model. But before putting text into the neural network language model, we need preprocessing text into small words/tokens and then represent each word/token as a vector.

### 2.2.1 One-Hot Vector Representation

Before Word2Vec[14], one-hot vector representations were commonly used to represent words. However, one-hot vectors have significant limitations, such as high dimensionality and lack of semantic information.

In a one-hot vector representation, each word in the vocabulary is represented by a vector with the same length as the size of the vocabulary. The vector is filled with zeros except for a single one at the index corresponding to the word. For example, in a vocabulary of 5 words: ["apple", "banana", "cherry", "date", "elderberry"], the one-hot vectors would be:

$$\begin{aligned} \text{"apple"} &\rightarrow [1, 0, 0, 0, 0] \\ \text{"banana"} &\rightarrow [0, 1, 0, 0, 0] \\ \text{"cherry"} &\rightarrow [0, 0, 1, 0, 0] \\ \text{"date"} &\rightarrow [0, 0, 0, 1, 0] \\ \text{"elderberry"} &\rightarrow [0, 0, 0, 0, 1] \end{aligned}$$

While one-hot vectors are very simple, there are serious limitations of One-Hot Representation:

- (1) **High Dimensionality:** For a large vocabulary, the vector size becomes extremely large, leading to inefficiency in storage and computation, especially for multilingual purpose.
- (2) **Sparsity:** The vectors are sparse, containing mostly zeros, which makes them inefficient in terms of space.
- (3) **Lack of Semantic Information:** One-hot vectors do not capture any semantic relationships between words. For example, "apple" and "banana" are as different as "apple" and "cherry" despite the former being more semantically related.

Thus one-hot vector representation is hard to implement into neural network models, since they waste a lot of dimensions. A normal word may contain 3000+ dimensions.

## 2.2.2 Word2Vec/Embeddings

Word2Vec, introduced by Mikolov et al. in 2013[14], revolutionized word representation by addressing these limitations. Word2Vec uses neural networks to learn dense vector representations (embeddings) of words, capturing semantic relationships between them. There are 3 important features of Word2Vec:

(1) **Dense Representations** Words are represented as dense vectors in a lower-dimensional space, typically 100-300 dimensions.

(2) **Semantic Similarity**: Words with similar meanings are mapped to nearby points in the vector space. For example, "king" and "queen" are closer to each other than "king" and "apple".

(3) **Training Methods**: There are two main approaches to training Word2Vec:

(3.1) **Continuous Bag of Words (CBOW)**: Predicts a target word based on its context (surrounding words).

(3.2) **Skip-Gram**: Predicts the context words given a target word.

By using Word2Vec, the limitations of one-hot encoding are mitigated, and more meaningful word representations are obtained, which significantly improve the performance of downstream NLP tasks.

## 2.3 Text Preprocessing

After understanding the embedding concept, We could focus on a step-by-step description of the preprocessing of text before sending it to neural network language models:

### 2.3.1 The Plain Text

The starting point is a plain text string, which is the input text that needs to be processed. Example: "*The quick brown fox jumps over the lazy dog*"

### 2.3.2 Tokens

The text string is then tokenized. Tokenization is the process of breaking the text into smaller units called tokens. Tokens can be words, sub-words, or even characters, depending on the tokenization strategy used by the language model.

Example tokens (word-based): ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]

Example tokens (sub-word-based, as used in models like Transformers): ["The", "quick", "brown", "fox", "jump", "##s", "over", "the", "lazy", "dog"]

### 2.3.3 Token IDs

Each token is then mapped to a unique integer ID using a predefined vocabulary. This mapping converts the tokens into a sequence of numbers that the model can process.

Example token IDs: [2026, 1018, 2154, 4419, 3497, 2052, 1996, 7173, 3899]

### 2.3.4 Embedding into N-Dimension Vectors

The embedding layer produces these n-dimensional vectors for each token ID. This step transforms the sequence of token IDs into a sequence of n-dimensional vectors. These vectors capture semantic information about the tokens.

The embedding layer weights will be changed during training, just like in the Word2Vec.

Example:

$$\begin{bmatrix} \text{"The"} & \rightarrow & 0.12 & -0.45 & 0.68 & \cdots & 0.23 \\ \text{"quick"} & \rightarrow & 0.34 & 0.56 & -0.12 & \cdots & -0.78 \\ \vdots & & \vdots & \vdots & \vdots & \ddots & \vdots \\ \text{"dog"} & \rightarrow & 0.91 & -0.37 & 0.15 & \cdots & 0.67 \end{bmatrix}$$

### 2.3.5 Summary

The preprocessing steps transform the input text into a form that the language model can work with. Starting from the original string, the text is tokenized into smaller units, converted into unique IDs, mapped to dense vectors through an embedding layer, and then further processed into high-dimensional vectors that can be used by the model to perform various tasks like text generation, classification, or translation.

## 2.4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs)[15] are a class of neural networks designed for processing sequential data. They are commonly used in tasks such as language modeling, machine translation, and time series prediction. The most important types of RNNs include Vanilla RNNs, GRUs (Gated Recurrent Units), and LSTMs (Long Short-Term Memory Networks).

### 2.4.1 Vanilla RNNs

Vanilla RNNs are the simplest type of RNN. They process sequences of data by maintaining a hidden state that is updated at each time step based on the input at that step and the previous hidden state. This hidden state acts as a form of memory, allowing the network to capture information from previous time steps. However, Vanilla RNNs are limited by their short-term memory and struggle with learning long-term dependencies due to issues such as vanishing and exploding gradients.

### 2.4.2 LSTMs (Long Short-Term Memory Networks)

LSTMs[16] are a type of RNN specifically designed to overcome the limitations of Vanilla RNNs. They use a more complex architecture that includes memory cells and gates (input, output, and forget gates) to control the flow of information. This allows LSTMs to remember long-term dependencies more effectively, making them suitable for tasks that require capturing long-range patterns in data. LSTMs have been widely used in various applications, including language modeling, speech recognition, and time series prediction.

### 2.4.3 GRUs (Gated Recurrent Units)

GRUs[17] are another type of RNN that aim to solve the vanishing gradient problem and improve the ability to capture long-term dependencies. GRUs are similar to LSTMs but have a simpler architecture with fewer gates. They combine the input and forget gates into a single update gate and merge the hidden state and cell state into one. This simplification makes GRUs computationally more efficient while still maintaining the ability to model long-term dependencies effectively.

### 2.4.4 Summary

- (1) **Vanilla RNNs:** Suitable for sequential data but limited by short-term memory.
- (2) **LSTMs:** A type of RNN designed to remember long-term dependencies using memory cells and gates.
- (3) **GRUs:** Simplified RNNs with fewer gates that are computationally efficient and can model long-term dependencies.

These models have revolutionized the field of natural language processing (NLP) by enabling more sophisticated understanding and generation of human language.

## 2.5 Large Language Models

Large Language Models (LLMs) are advanced computational systems designed to understand and generate human language. They are built on the architecture of Transformer neural networks, which have revolutionized the field of natural language processing (NLP).

### 2.5.1 Transformer Neural Networks

The concept of Transformers, introduced by Vaswani et al. in 2017[7], marked a significant shift in how language models are constructed. Unlike previous models that relied heavily on recurrent neural networks (RNNs) and their variants, Transformers utilize a mechanism known as self-attention. This allows them to weigh the importance of different words in a sentence relative to one another, leading to better understanding and generation of text.

### 2.5.2 BERT (Bidirectional Encoder Representations from Transformers)

One of the pioneering models based on the Transformer architecture is BERT[6], developed by Google. BERT's innovation lies in its bidirectional training approach, meaning it considers the context from both directions (left-to-right and right-to-left) when processing a sentence. This approach enables BERT to achieve a deeper understanding of language context and nuances, making it highly effective for a wide range of NLP tasks, such as question answering and sentiment analysis.

### 2.5.3 GPTs (Generative Pre-trained Transformers)

Following BERT, OpenAI introduced the Generative Pre-trained Transformer (GPT) series[2][3][4][5]. GPT models are designed with a unidirectional approach, primarily focusing on generating coherent and contextually relevant text. Each iteration, from GPT to GPT-4, has demonstrated substantial improvements in text generation capabilities, making them suitable for applications such as chat-bots, content creation, and language translation.

### 2.5.4 Summary

In summary, Large Language Models, exemplified by BERT and GPT, leverage the power of Transformer neural networks to push the boundaries of what is possible in natural language processing. Their ability to understand and generate human-like text has opened up new possibilities in various domains, transforming how we interact with and utilize language technology.

## 2.6 Multi-modal Models

Multi-modal models are advanced AI systems capable of processing and generating various types of data, such as text, images, speech, and videos. These models integrate natural language processing (NLP) with computer vision and other modalities, allowing them to understand and produce content in multiple formats. For instance, GPT-4V[5] can analyze images to generate descriptions, while models like DALL-E[18] and Stable Diffusion[19] convert text into images. Sora[20] creates videos from text, and Suno[21] generates music from textual input. Launched in May 2024, GPT-4o[22] is a comprehensive end-to-end voice model with both text-to-speech (TTS) and automatic speech recognition (ASR) capabilities.

In this thesis, however, we will focus on text-to-text large language models and their agents rather than multi-modal models.

# Chapter 3

## What is LLM Agent?

LLM (Large Language Model) Agents refers to a system or entity that leverages large language models to perform various tasks. These tasks can include generating text, answering questions, summarizing content, translating languages, and more. One of the most important key component, is to call outside tools and use the tools to get information they do not know, and return the result to the user.

This chapter will introduce the concept and components of LLM agents, most of the content is from "*LLM Powered Autonomous Agents*" written by *Lilian Wang* from OpenAI[23].

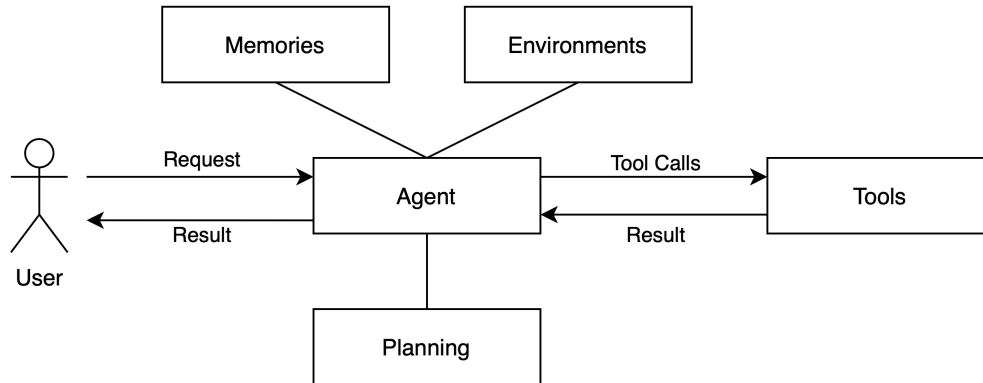


Figure 3.1: Components of LLM Agents

### 3.1 Overview

In an LLM-powered autonomous agent system, the LLM serves as the agent's brain, supported by several key components: User Interaction, Environments, Planning, Memory, Tool Use.



## 3.2 User Interactions

Handles user requests and returns results.

## 3.3 Environments

The various contexts and scenarios the agent operates within. This environments information will be included into the system role prompt.

## 3.4 Memory

**In-context Learning:** Utilizes the model’s short-term memory to adapt and learn from immediate data.

**Long-term Memory:** Retains and recalls extensive information over extended periods, often leveraging an external vector store and fast retrieval methods.

## 3.5 Planning

**Subgoal and Decomposition:** The agent divides large tasks into smaller, manageable subgoals to efficiently tackle complex tasks.

**Reflection and Refinement:** The agent engages in self-criticism, reflects on past actions, learns from mistakes, and refines its approach for future tasks, thereby enhancing the quality of final results.

### 3.5.1 Task Decomposition

Complex tasks typically involve multiple steps that need careful planning and execution. Effective task decomposition strategies enable an agent to break down a large problem into smaller, manageable steps, enhancing problem-solving efficiency.

**Chain of Thought (CoT; Wei et al., 2022)[24]** has emerged as a standard prompting technique to enhance model performance on complex tasks. CoT instructs the model to "think step by step," transforming large tasks into smaller, more manageable ones. This approach illuminates the model’s thought process and facilitates better task execution.

**Tree of Thoughts (ToT; Yao et al., 2023)[25]** extends CoT by exploring multiple reasoning possibilities at each step. It creates a tree structure where each node represents a thought, and multiple thoughts are generated per step. The search process can employ either breadth-first search (BFS) or depth-first search (DFS), with each state evaluated by a classifier or majority vote.

Task decomposition can be achieved through: (1) Simple prompting: e.g., "Steps for XYZ." (2) Task-specific instructions: e.g., "Write a story outline." (3) Human inputs.

Another approach, **LLM+P (Liu et al., 2023)[26]**, involves using an external classical planner for long-horizon planning. This method utilizes the Planning Domain Definition

Language (PDDL) to describe the planning problem, outsourcing the planning step to an external tool.

### 3.5.2 Self-Reflection

Self-reflection is crucial for autonomous agents to iteratively improve by refining past decisions and correcting mistakes. It is particularly important in real-world tasks where trial and error are inevitable.

**ReAct (Yao et al., 2023)**[27] integrates reasoning and acting within a large language model (LLM) by combining task-specific actions with language-based reasoning. This approach allows the LLM to interact with the environment and generate reasoning traces in natural language, leading to better performance in both knowledge-intensive and decision-making tasks.

**Reflexion (Shinn & Labash, 2023)**[28] equips agents with dynamic memory and self-reflection capabilities to enhance reasoning skills. Reflexion follows a reinforcement learning setup, where the agent uses self-reflection to compute heuristics and decide on new trials based on past experiences.

**Chain of Hindsight (CoH; Liu et al., 2023)**[29] encourages models to improve their outputs by presenting a sequence of past outputs with feedback. This approach involves fine-tuning the model to use feedback to produce better outputs incrementally.

By leveraging task decomposition and self-reflection, autonomous agents can effectively plan and execute complex tasks, leading to continuous improvement and better performance over time.

## 3.6 Tool Use

**External API Integration:** The agent learns to call external APIs for additional information not present in the model weights, including current data, code execution capabilities, access to proprietary information sources, and more.

## 3.7 LLM and Agent deployment

The LLM works as the brain of the agent. It can be installed locally or on a server such as the OpenAI API server, while the agent component can run on a server or a client where is convenient for tasks.

## 3.8 Agent Frameworks

LLM agent frameworks are software architectures and toolkits designed to build intelligent agents powered by large language models (LLMs). These frameworks provide the necessary components and infrastructure to create agents capable of understanding and responding to user queries, accessing external tools and data sources, maintaining memory, and executing complex tasks. Some notable examples of LLM agent frameworks include:

**OpenAI Assistants:** The OpenAI Assistants API and associated frameworks empower developers to build sophisticated multi-agent systems capable of tackling complex tasks through specialized expertise, parallel execution, and coordinated decision-making.[30][31][32]

**LangChain:** A popular open-source framework for building applications and agents with LLMs. It provides a modular and extensible architecture, allowing developers to integrate various LLMs, memory components, and external tools.[33][34]

**LlamaIndex:** A framework focused on connecting LLMs with custom data sources, enabling agents to retrieve and process information from various data formats.[34]

**Haystack:** An NLP framework designed for building LLM-powered applications, including question-answering agents and document retrieval systems.[35]

**AutoGPT:** A framework that provides tools and utilities for building autonomous AI agents capable of self-improvement and task completion.[34]

**AutoGen:** A framework that enables the development of multi-agent LLM applications, where agents can collaborate and communicate to solve complex tasks.[34]

LLM agent frameworks empower developers to create intelligent agents that can perform a wide range of tasks, such as:

**Question answering:** Agents can understand and respond to user queries by retrieving relevant information from various data sources and generating coherent and contextual responses.[33][34][35]

**Task automation:** Agents can automate various tasks by breaking them down into smaller steps, utilizing external tools and APIs, and executing the necessary actions.[33][35]

**Content creation:** Agents can assist with writing, editing, and proofreading tasks by leveraging the language generation capabilities of LLMs.[35]

**Data analysis:** Agents can interact with structured data sources like databases or APIs, extract and analyze information, and provide valuable insights to users.[35]

**Multi-agent collaboration:** Frameworks like AutoGen enable the development of multi-agent systems, where agents can collaborate, communicate, and share information to tackle complex problems more effectively.[34]

**Research and development:** LLM agent frameworks are being actively explored in research and development domains, pushing the boundaries of artificial intelligence and enabling the creation of specialized agents for various fields, such as mathematics, chemistry, and coding.[35]

By combining the power of LLMs with external tools, memory management, and planning capabilities, LLM agent frameworks unlock a wide range of possibilities for building intelligent and versatile agents that can assist humans in various tasks and domains.[33][34][35]

# Chapter 4

## Input Limitation of the LLMs

LLMs like GPTs will finally accept sequence of tokens as described in Chapter 2, and generate sequence of tokens as a result. The Transformer architecture has a limitation of the total length of the input token sequence, which is called context window.

### 4.1 Context Window

The context window is the maximum number of tokens they can consider at once. This context window is critical because it determines how much text the model can take into account when generating responses.

The size of the context window impacts the model’s ability to understand and generate coherent, contextually relevant text. A larger window allows for more context and can improve performance, especially on tasks requiring long-range dependencies.

If the input text exceeds the context window, the model cannot consider the entire text at once. Instead, it processes the most recent portion of the input within the window size. When generating long responses, the model might lose track of earlier parts of the conversation or document if the total token count exceeds the context window.

#### 4.1.1 GPT Model Descriptions and Context Window

[36]

MODEL	DESCRIPTION	CONTEXT WINDOW
gpt-4o	New GPT-4o: Our most advanced, multimodal flagship model that’s cheaper and faster than GPT-4 Turbo. Currently points to gpt-4o-2024-05-13.	128,000 tokens
gpt-4o-2024-05-13	gpt-4o currently points to this version.	128,000 tokens

gpt-4-turbo	New GPT-4 Turbo with Vision: The latest GPT-4 Turbo model with vision capabilities. Vision requests can now use JSON mode and function calling. Currently points to gpt-4-turbo-2024-04-09.	128,000 tokens
gpt-4-turbo-2024-04-09	GPT-4 Turbo with Vision model. Vision requests can now use JSON mode and function calling. gpt-4-turbo currently points to this version.	128,000 tokens
gpt-4-turbo-preview	GPT-4 Turbo preview model. Currently points to gpt-4-0125-preview.	128,000 tokens
gpt-4-0125-preview	GPT-4 Turbo preview model intended to reduce cases of “laziness” where the model doesn’t complete a task. Returns a maximum of 4,096 output tokens. Learn more.	128,000 tokens
gpt-4-1106-preview	GPT-4 Turbo preview model featuring improved instruction following, JSON mode, reproducible outputs, parallel function calling, and more. Returns a maximum of 4,096 output tokens. This is a preview model. Learn more.	128,000 tokens
gpt-4-vision-preview	GPT-4 model with the ability to understand images, in addition to all other GPT-4 Turbo capabilities. This is a preview model, we recommend developers to now use gpt-4-turbo which includes vision capabilities. Currently points to gpt-4-1106-vision-preview.	128,000 tokens
gpt-4-1106-vision-preview	GPT-4 model with the ability to understand images, in addition to all other GPT-4 Turbo capabilities. This is a preview model, we recommend developers to now use gpt-4-turbo which includes vision capabilities. Returns a maximum of 4,096 output tokens. Learn more.	128,000 tokens

gpt-4	Currently points to gpt-4-0613. See continuous model upgrades.	8,192 tokens
gpt-4-0613	Snapshot of gpt-4 from June 13th 2023 with improved function calling support.	8,192 tokens
gpt-4-32k	Currently points to gpt-4-32k-0613. See continuous model upgrades. This model was never rolled out widely in favor of GPT-4 Turbo.	32,768 tokens
gpt-4-32k-0613	Snapshot of gpt-4-32k from June 13th 2023 with improved function calling support. This model was never rolled out widely in favor of GPT-4 Turbo.	32,768 tokens
gpt-3.5-turbo-0125	New Updated GPT 3.5 Turbo: The latest GPT-3.5 Turbo model with higher accuracy at responding in requested formats and a fix for a bug which caused a text encoding issue for non-English language function calls. Returns a maximum of 4,096 output tokens. Learn more.	16,385 tokens
gpt-3.5-turbo	Currently points to gpt-3.5-turbo-0125.	16,385 tokens
gpt-3.5-turbo-1106	GPT-3.5 Turbo model with improved instruction following, JSON mode, reproducible outputs, parallel function calling, and more. Returns a maximum of 4,096 output tokens. Learn more.	16,385 tokens
gpt-3.5-turbo-instruct	Similar capabilities as GPT-3 era models. Compatible with legacy Completions endpoint and not Chat Completions.	4,096 tokens
gpt-3.5-turbo-16k	Legacy: Currently points to gpt-3.5-turbo-16k-0613.	16,385 tokens
gpt-3.5-turbo-0613	Legacy: Snapshot of gpt-3.5-turbo from June 13th 2023. Will be deprecated on June 13, 2024.	4,096 tokens
gpt-3.5-turbo-16k-0613	Legacy: Snapshot of gpt-3.5-16k-turbo from June 13th 2023. Will be deprecated on June 13, 2024.	16,385 tokens

## 4.2 RAG (Retrieval-Augmented Generation)

RAG[37] combines the strengths of retrieval-based and generation-based approaches to enhance the model's performance on tasks requiring extensive background knowledge.

### 4.2.1 Process

**Retrieval:** When a query is presented, the model retrieves relevant documents or passages from an external knowledge base (e.g., Wikipedia, internal documents).

**Generation:** The model then uses the retrieved information along with the query to generate a response. This ensures that the generated text is informed by the most relevant and up-to-date information.

### 4.2.2 Components

**Retriever:** A system (e.g., a search engine or a neural retriever) that identifies and fetches relevant documents based on the query.

**Generator:** A language model that takes both the query and the retrieved documents as input to generate a coherent and contextually rich response.

### 4.2.3 Advantages

**Enhanced Knowledge:** By incorporating external information, RAG models can provide more accurate and comprehensive answers, especially for specific or niche topics.

**Scalability:** The retrieval component can access vast amounts of information without increasing the model's size, making it more scalable.

**Dynamic Updates:** The model can stay up-to-date with the latest information by retrieving current documents, without needing to retrain the entire model frequently.

## 4.3 Summary

In summary, the context window defines the limit of tokens an LLM can handle at once, and various methods like sliding windows and hierarchical models help manage longer contexts. RAG enhances the model's capabilities by integrating external knowledge through retrieval and generation processes.

# Chapter 5

## Important OpenAI APIs for GPT Models

GPT models are deployed on the Open AI Server, and developers and other services could call the OpenAI API to utilize the model capabilities. In this chapter, we'll focus on OpenAI API for GPT models and how it evolves its input and output format.



## 5.1 Text Completions API

The [Text Completions API](#)[38] takes plain text as input and generates plain text as output, predicting the most likely continuation of the input based on the GPT models' understanding. This API, originally introduced for GPT-1 through GPT-3, has been deprecated for newer models starting from GPT-3.5.

Listing 5.1: [Text Completions API](#) call in Python[38]

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 input_text = "Write a tagline for an ice cream shop."
5
6 response = client.completions.create(
7     model="gpt-3.5-turbo-instruct",
8     prompt= input_text
9 )
10
11 output_text = response["choices"][0]["text"]
12 print(output_text) # "Indulge in happiness, one scoop at a time."
```

We have created a diagram to demonstrate the [Text Completions API](#) for GPT-1 to GPT-3. The Text Completions API service, including the GPT LLM models, is deployed on OpenAI's servers. A developer app or server, or even the [OpenAI's online Playground for Completions](#)[39] is marked as a client in the figure, since they are the caller side of the API. Before sending the input to the model, a preprocessing step occurs. The input text will be translated into a token sequence, and its length must be smaller than the context window. Otherwise, an error will occur.

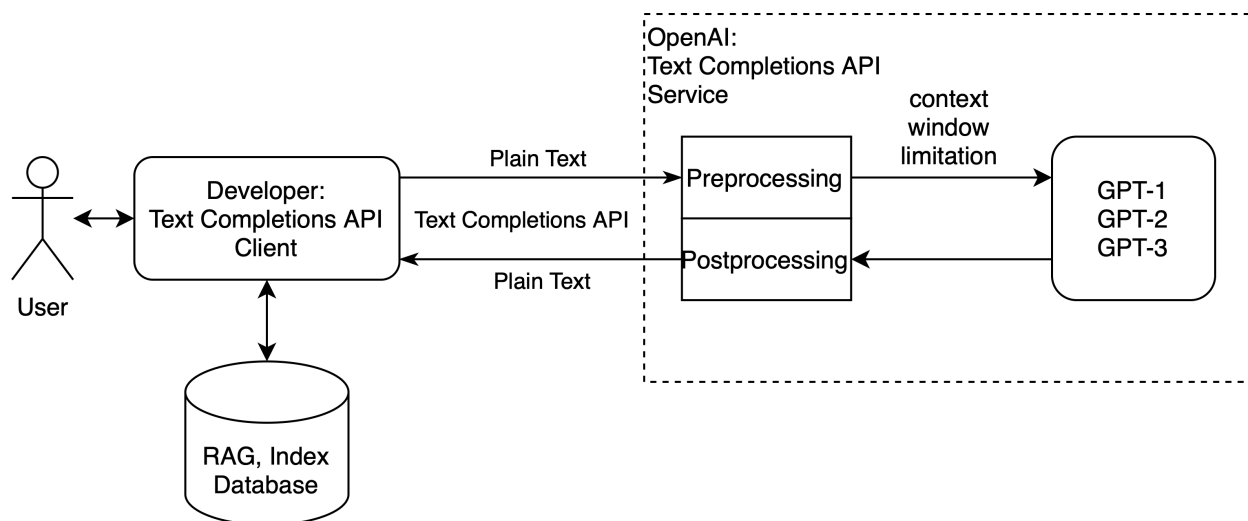


Figure 5.1: Text Completions API

Most open-source language models, including GPT-1 to GPT-3, use plain text for both input and output. These models are typical causal language models that predict a continuation of the input text. They are relatively easy to prepare and train, as we only need a sufficient amount of free text materials, such as articles, tables, and code, to train the language model.

## 5.2 Chat Completions API

Since the introduction of GPT-3.5 in November 2022, the input and output formats have been completely changed. The new [Chat Completions API](#)[\[40\]](#) does not accept plain text but requires a list of messages. The output is the predicted subsequent message.

Listing 5.2: Chat Completions API call in Python[\[40\]](#)

```
1 from openai import OpenAI
2 client = OpenAI()
3
4 response = client.chat.completions.create(
5     model="gpt-3.5-turbo",
6     messages=[
7         {"role": "system", "content": "You are a helpful assistant."},
8         {"role": "user", "content": "Who won the world series in 2020?"},
9         {"role": "assistant", "content": "The Los Angeles Dodgers won the World Series in 2020."
10        },
11        {"role": "user", "content": "Where was it played?"}
12    ]
13)
14 output_message = response["choices"][0]["message"]
15 print(output_message) # {"role": "assistant", content: "The 2020 World Series was played at
16                       Globe Life Field in Arlington, Texas. This was a unique arrangement due to the COVID-19
17                       pandemic, as the league decided to use a neutral-site "bubble" to minimize travel and
18                       reduce the risk of spreading the virus."}
```

When developers work on the client side of the [Chat Completions API](#), it is crucial to prepare a list of messages that includes both the historical messages and the most recent user request. It is the client's responsibility to maintain the correct history of messages and send them in the correct order to the API.

The chat message list will ultimately be processed and translated into tokens before being sent to the GPT models. Thus, be mindful that the total number of tokens must be smaller than the context window limit of the specific large language model being used. If this limit is exceeded, an API error will be returned to the client.

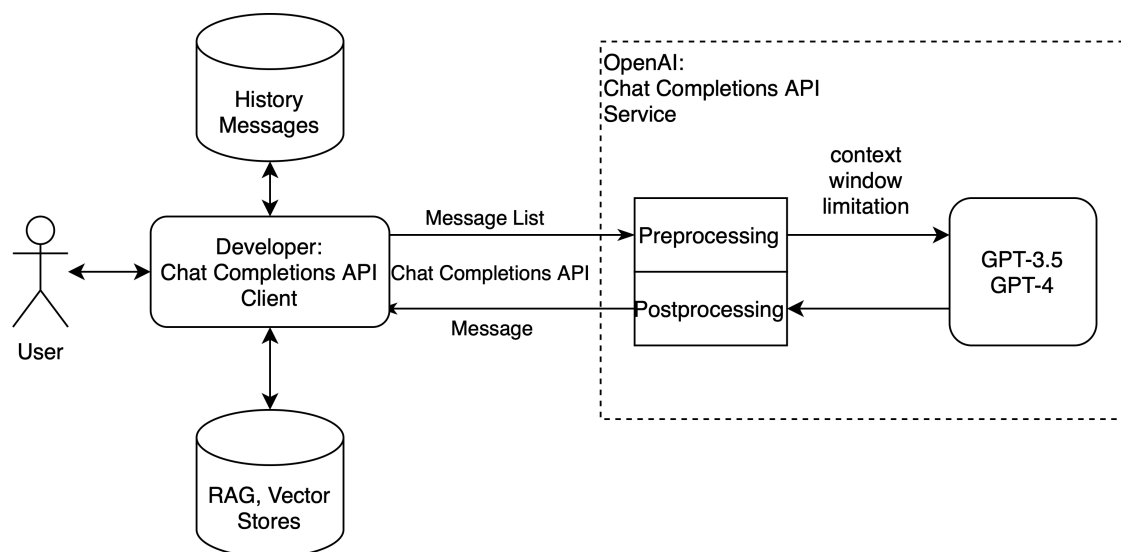


Figure 5.2: Chat Completions API

### 5.2.1 Roles of a Chat Message

Each message has two components: role and the content. The role is an enumerated value, could be one of `system`, `user` and `assistant`.

**System role:** Most of the case, it should be showed up on the first message of the message list send to the Chat Completions API. The System role is typically hidden from users but plays a vital part in guiding the language model to produce relevant and accurate responses. This role provides essential context and instructions that help the model understand the environment and the specifics of the interaction. For instance, knowing the current date allows the model to generate responses that are timely and appropriate. The System role ensures that the interaction flows smoothly by maintaining contextual coherence and aligning the model's responses with the user's needs.

**User role:** The User role represents the individual interacting with the AI assistant. The user asks questions, makes requests, and provides input that drives the conversation. The user's role is crucial as it initiates and directs the flow of interaction, allowing the assistant to understand their needs and preferences. The user can engage with the assistant on a wide range of topics, from seeking information and assistance to casual conversation and creative collaboration.

**Assistant role:** The Assistant role message is the response from GPT models. The assistant interacts directly with the user, providing information, answering questions, and engaging in dialogue based on the prompts and context provided. The assistant leverages the hints and guidelines from the system role to ensure that responses are accurate, relevant, and helpful. The assistant aims to create a seamless and intuitive user experience by being responsive, informative, and empathetic.

Together, these roles create a structured and effective conversational environment, ensuring that interactions are meaningful and aligned with the user's expectations.

### 5.2.2 Example of Chat Messages

Here we have an example of input and output (as shown in Table 5.1). The input will include history messages and the last user request message. Also the output is the same message structure, but it only output one message, with the role of `assistant`.

	<b>Role</b>	<b>Content</b>
Input Messages	System	Assistant is a large language model. Current time is April 2, 2024.
	User	How can you help me today?
	Assistant	I can help answer your questions or provide information on a wide range of topics.
	User	What date is today?
Output Message	Assistant	Today is April 2, 2024.

Table 5.1: An example of chat messages input-output

## Chat Messages in JSON format

We could also use JSON to represent the chat messages. The input list of messages usually formatted in JSON when developers bundled all the history messages in the message lists to the OpenAI API. For example:

```
1 [
2   {"role": "system", "content": "Assistant is a large language model. Current time is is
3     April 2, 2024. "},
4   {"role": "user", "content": "How can you help me today?"},
5   {"role": "assistant", "content": "I can help answer your questions or provide information
6     on a wide range of topics."},
7   {"role": "user", "content": "What date is today?"}
8 ]
```

Listing 5.3: Input in JSON Format

Also, the output could also be represented in JSON format, since it is only one message, we do not need the list to wrap the only one assistant message.

```
1 {"role": "assistant", "content": "Today is April 2, 2024."}
```

Listing 5.4: Output in JSON Format

## Chat Messages in ChatML format

While, finally, the JSON format will be flattened into ChatML<sup>[41]</sup> formatted text as the input of the model on the OpenAI server. So the model actually accepts ChatML formatted text, which list of history messages with different roles.

Listing 5.5: Input in ChatML Format

```
1 <|im_start|>system
2 Assistant is a large language model. Current time is is April 2, 2024.
3 <|im_end|>
4 <|im_start|>user
5 How can you help me today?
6 <|im_end|>
7 <|im_start|>assistant
8 I can help answer your questions or provide information on a wide range of topics.
9 <|im_end|>
10 <|im_start|>user
11 What date is today?
12 <|im_end|>
13 <|im_start|>assistant
```

Listing 5.6: Output in ChatML Format

```
1 Today is April 2, 2024.
2 <|im_end|>
```

### 5.2.3 Training Data for Chat Messages

Since the model input format ChatML is different than the free text, and limit the model output in the same ChatML format with the end token `|im_end|`, it requires all the training data formatted in ChatML for training GPT-3.5 and GPT-4.

### 5.2.4 Reason for using Chat Messages

The reason to change the free text input-output to chat messages, is to make the language model interaction more like a human conversation. User could have multiple turns conversation with GPT and remember all the roles associated with the content components, using `user` role to mark the `user` inputs and `assistant` role to mark the model outputs.

This is a product direction change, which make the GPT language models more like a general purpose chatbot instead of Natural Language Processing academic purpose experiments, thus OpenAI renamed their product from GPT to ChatGPT after GPT-3.5. Also, OpenAI use reinforcement learning algorithm (InstructGPT) to make the model choices of conversation more like human.

## 5.3 Chat Completions API with Function Calling

From the advent of an update since June, 2023, OpenAI introduced the capability of **Function Calling**[42] within the chat completions API on GPT-3.5 and GPT-4 models, and also demonstrated in the article *Function calling and other API updates*[43]. This extension of the model's abilities represents a significant evolution in how users interact with language models.

When developers call the **Chat Completions API with Function Calling**, not only the message lists will send to the API, but also the function definitions. The function definitions will be translated into tokens, which are counted towards the context window size when passed into the GPT models. Therefore, we must ensure not to exceed the context window token size limit.

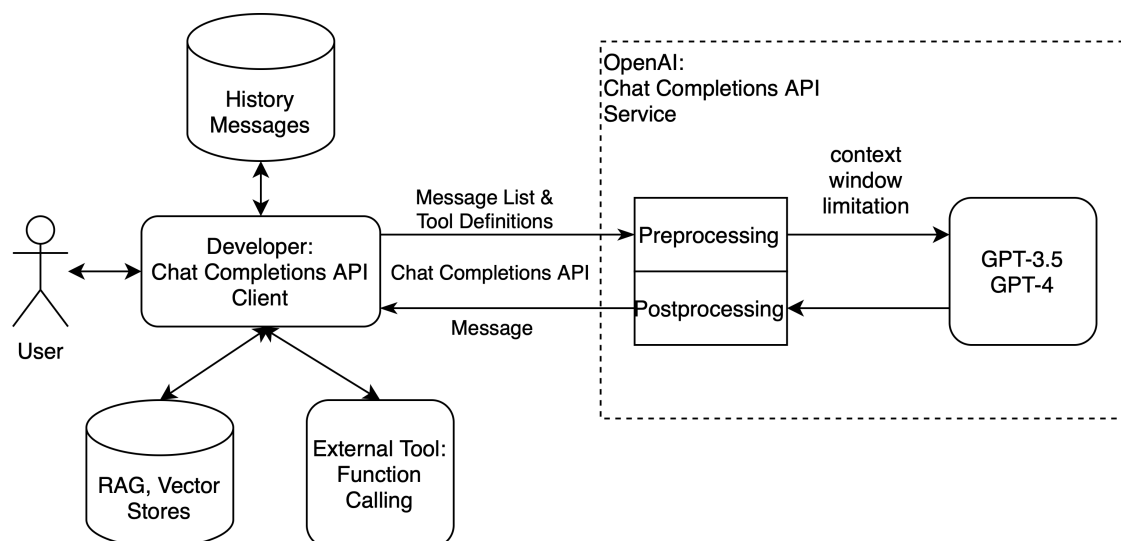


Figure 5.3: Chat Completions API with Function Calling

There are many other ways to implement language models' function calling features, we'll explain more on next chapter. In this section, we only introduced the Tool Augmented Model method, which GPT-3.5 and GPT-4 model deployed.

### 5.3.1 Example of Function Calling

Consider a conversation that user ask for the current weather of Boston. Language Model could not answer the question without querying the external weather API. Thus the GPT-3.5 and GPT-4.0 introduced the capability for calling developer prepared functions to query the necessary information.

#### Definition of a function in JSON Schema format

So first of all, developers need describing functions that the language model could call. OpenAI models ask the function descriptions defined in JSON-Schema. For example, the `get_weather` function has two arguments, one is the location and another is the unit of temperature.

```
1 {
2   "name": "get_weather",
3   "description": "Get the current weather in a given location",
4   "parameters": {
5     "type": "object",
6     "properties": {
7       "location": {
8         "type": "string",
9         "description": "The city and state, e.g. San Francisco, CA"
10      },
11      "unit": {
12        "type": "string",
13        "enum": ["celsius", "fahrenheit"]
14      }
15    },
16    "required": ["location", "unit"]
17  }
18 }
```

Listing 5.7: Function Defination in JSON Format

#### Definition of a function in ChatML format

function definitions also translate into a specific format before sending to GPT-3.5 or GPT-4 model, just like the message list data translating into ChatML format. It has been found as a TypeScript-like grammar[44].

```
1 # Tools
2
3 ## functions
4
5 namespace functions {
6
7 // Description of example function the AI will repeat back to the user
8 type get_eather = (_: {
9 // description of function property 1: string
10 location: string,
11 // description of function property 2: string w enum
12 unit: "celsius" | "fahrenheit",
13 }) => any;
14
15 } // namespace functions
```

Listing 5.8: Function Defination in ChatML Format[44]

## Function Calling Process

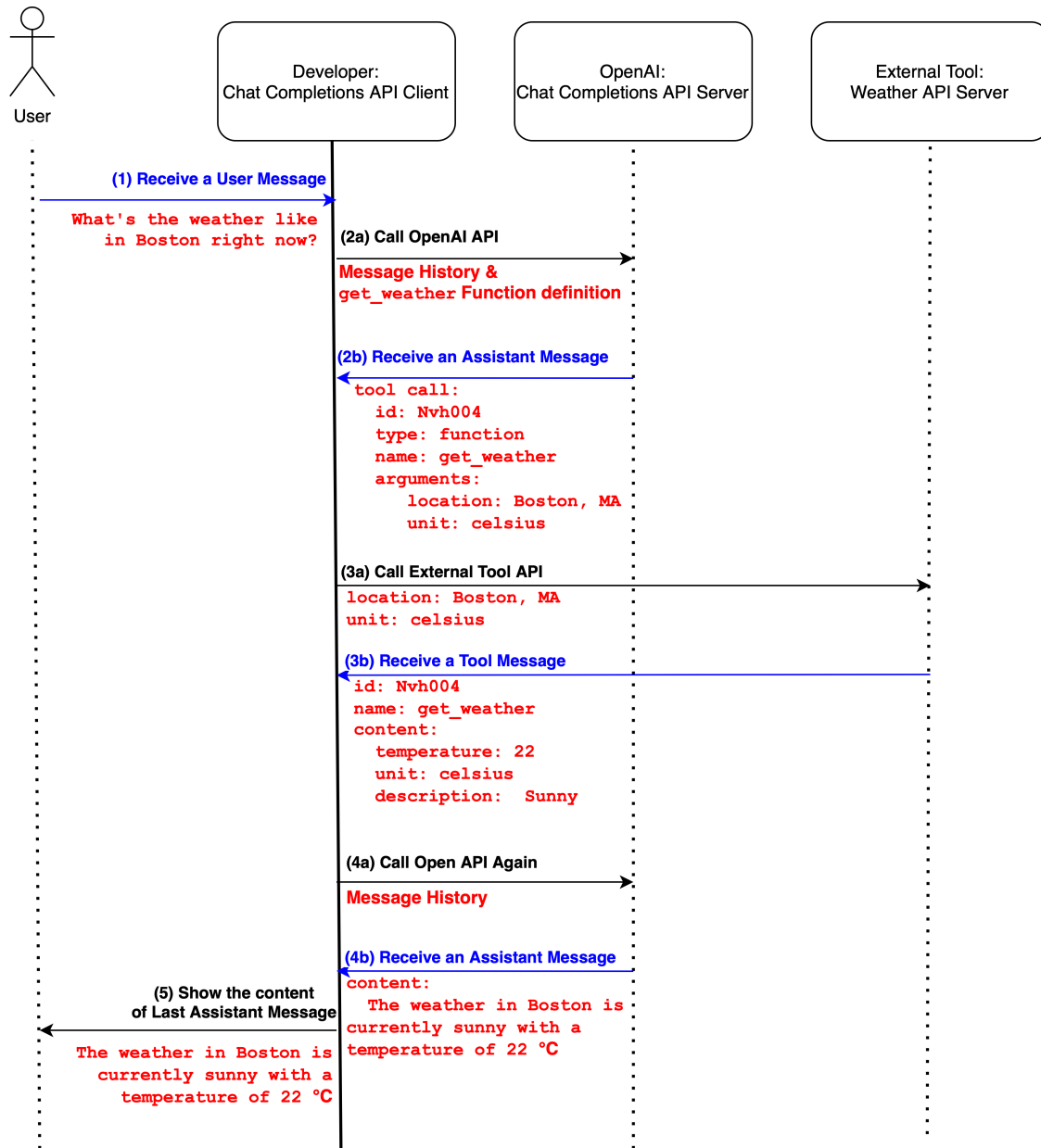


Figure 5.4: function calling Process regarding the weather in Boston

(1) The user may ask a question to the developer's application, such as "What's the weather like in Boston right now?" The developer's application will then record the user's message in its message history. In the figure, all blue arrows represent steps that record a new message in the message history.

(2a) The developer's application sends the message list, including the latest user message with the content "What's the weather like in Boston right now?" to the OpenAI Chat Completions API.



(2b) When the GPT models determine it is necessary to make a tool call instead of directly returning message content to the user, in this case, the model will generate an assistant role message with a function-type tool call and notify the developer’s code to call the function with the provided arguments (location="Boston, MA", unit="Celsius").

(3a) After the developer’s application receives the assistant message with a function-typed tool call request, the application will call the weather API with the arguments (location: location="Boston, MA", unit="Celsius") and wait for the response from the weather API server.

(3b) The weather API server will return the weather result (temperature="22", unit="Celsius", description="Sunny") to the developer’s application. The developer’s application will then record a new **Tool role** message, including the tool call ID and function name from (2b), along with the weather result. This message will clearly indicate the result for the specific tool call ID, making it easy for LLMs to trace the tool call and its corresponding result.

(4a) The developer’s application sends the updated message list, including the latest tool role message described in (3b), to the Chat Completions API. The API server now has all the information it needs to generate a response to the user’s weather inquiry.

(4b) If the GPT model determines no further tool calls are needed and the information is sufficient to answer the user’s request, it will generate a natural language response based on the weather result. It will create an assistant message with content describing the weather: "The weather in Boston is currently sunny with a temperature of 22°C."

(5) The developer’s application receives the assistant message with the complete response to the weather inquiry and displays the content of the assistant message to the user.

Role	Tool Call Properties	Content
User	/	"What’s the weather like in Boston right now?"
Assistant	<pre>Tool Calls = [{   "id": "Nvh004",   "type": "function",   "function": {     "name": "get_weather",     "arguments": {       "location": "Boston, MA",       "unit": "celsius"     }   } }]</pre>	/
Tool	<pre>Tool Call ID = "Nvh004" Name = "get_weather"</pre>	<pre>{   "temperature": "22"   "unit": "celsius"   "description": "Sunny" }</pre>
Assistant	/	"The weather in Boston is currently sunny with 22 °C."

Table 5.2: Message History Regarding the Weather in Boston

Listing 5.9: Chat Completions API with Function Calling code in Python[42]

```

1 from openai import OpenAI
2 import json
3 client = OpenAI()
4 # Example dummy function hard coded to return the same weather
5 # In production, this could be your backend API or an external API
6 def get_weather(location, unit="celsius"):
7     return '{"temperature": '22', 'unit': 'celsius', 'description': 'sunny'}"
8
9 def run_conversation():
10    # Step 1: send the conversation and available functions to the model
11    messages = [{"role": "user", "content": "What's the weather like in Boston?"}]
12    tools = [{
13        "type": "function",
14        "function": {
15            "name": "get_weather",
16            "description": "Get the current weather in a given location",
17            "parameters": {
18                "type": "object",
19                "properties": {
20                    "location": {
21                        "type": "string",
22                        "description": "The city and state, e.g. San Francisco, CA",
23                    },
24                    "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
25                },
26                "required": ["location"],
27            },
28        },
29    }]
30    response = client.chat.completions.create(
31        model="gpt-4o",
32        messages=messages,
33        tools=tools,
34        tool_choice="auto", # auto is default, but we'll be explicit
35    )
36    response_message = response.choices[0].message
37    tool_calls = response_message.tool_calls
38    # Step 2: check if the model wanted to call a function
39    if tool_calls:
40        # Step 3: call the function
41        # Note: the JSON response may not always be valid; be sure to handle errors
42        available_functions = {"get_weather": get_weather} # only one function in this
43        # example, but you can have multiple
44        messages.append(response_message) # extend conversation with assistant's reply
45        # Step 4: send the info for each function call and function response to the model
46        for tool_call in tool_calls:
47            function_name = tool_call.function.name
48            function_to_call = available_functions[function_name]
49            function_args = json.loads(tool_call.function.arguments)
50            function_response = function_to_call(
51                location=function_args.get("location"),
52                unit=function_args.get("unit"),
53            )
54            messages.append( {
55                "tool_call_id": tool_call.id,
56                "role": "tool",
57                "name": function_name,
58                "content": function_response,
59            } ) # extend conversation with function response
60        second_response = client.chat.completions.create(
61            model="gpt-4o",
62            messages=messages,
63            ) # get a new response from the model where it can see the function response
64        return second_response
65    print(run_conversation()) # The weather in Boston is currently sunny with 22 degree.

```

### 5.3.2 Importance of Function Calling

In a typical scenario, Function calling is embedded within the chat as a part of the message list. The model interprets these commands, executes the specified function, and returns the result as part of the conversation. This process is seamlessly integrated into the chat, making the interaction between the user and the model more dynamic and versatile.

Function calling in GPT models allow the model to perform specific tasks beyond text prediction. These can range from accessing external databases and performing calculations to generating images and executing code. This functionality dramatically expands the potential applications of GPT models in various industries.

## 5.4 Assistants API

The **OpenAI Assistants API**[45] was launched on November 6, 2023, as announced at **OpenAI's developer conference**[46]. The Assistants API is a convenient tool for developers to create agents.

### 5.4.1 Features

The Assistants API is similar to the Chat Completions API introduced in previous sections. It has some extra features that the Chat Completions API does not have:

**Built on Chat Completions API:** The Assistants API is constructed on top of the Chat Completions API, utilizing its capabilities to enhance the service, as depicted in the figure below.

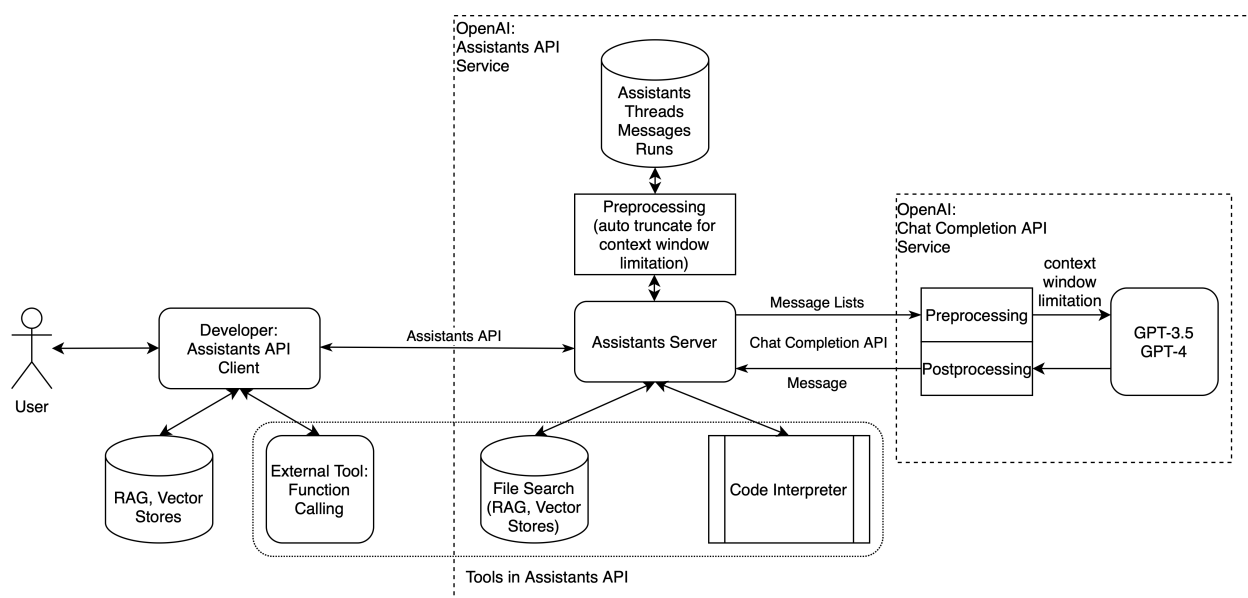


Figure 5.5: Chat Completions API with Function Calling

**More Agent-like Experiences:** Unlike the LLM API, the Assistants API functions more like an agent framework, providing additional services (especially RAG Service) beyond the LLM to simplify the development of agent applications.

**Internal System Instructions:** The Assistants API includes internal system instructions, which developers cannot control. These instructions utilize models, tools, and files to respond to user queries.

**Extra Tool Support:** The Assistants API supports three types of tools: Sandboxed Code Interpreter, File Search, and Function Calling. In contrast, the Chat Completions API only supports Function Calling. The Code Interpreter eliminates the need for developers to build their own calculators or code execution tools, facilitating accurate mathematical calculations and code execution results. The File Search tool (supporting up to 10,000 files per assistant) allows developers to avoid creating their own retrieval-augmented generation (RAG) system

for specific documents.

**Persistent Online Data Management for Assistants, Threads, Messages and Runs:**

The Assistants API automatically saves message history and provides APIs for managing Assistants, Threads, Messages, and Runs. With the Assistants API, developers can create persistent and infinitely long threads, enabling them to delegate thread state management to OpenAI and bypass context window constraints. Developers can create, edit, delete, and retrieve these entities for better control. The message history can be maintained for an extended period. The Assistants API automatically handles the context window limitation, using the most relevant context if the limit is reached. However, developers cannot change the order of the message list sent to the LLM.

**Image File Input Support:** The Assistants API supports image file input as a user message.

**Integration with GPT Store and Playground:** The GPTs[47] available in the GPT Store and OpenAI's online Playground for Assistants[48] are built on the Assistants API. GPTs allow users to create and configure their own GPT-based assistants directly through their web browser without writing any code. Users can upload files, configure system prompts, and customize the behavior of these assistants to suit their specific needs.

## 5.4.2 Choosing Between the Chat Completions API and the Assistants API

**Chat Completions API:** This API provides developers with more freedom. Developers can implement their own Retrieval-Augmented Generation (RAG) systems and manage message lists.

**Assistants API:** This API handles more of the heavy lifting for developers. It offers two additional tools (File Search and Code Interpreter) and automatically manages the context limitations of message histories. It eliminates the need for developers to maintain message history.

Developers can use the Chat Completions API to recreate an assistants API if they prefer more control and flexibility. However, if developers do not want such freedom, but want to store threads on cloud, and write less code and have the Assistants API assist in building the RAG system and Code Interpreter while managing context limitations, they should opt for the Assistants API.

## 5.5 Conclusion

The evolution from simple text-based models to those capable of handling structured chats and executing functional commands illustrates the significant advancements in the field of artificial intelligence. These developments not only improve user experience but also broaden the scope of applications for GPT models in real-world scenarios.

As we have seen, the evolution from GPT-1 through to GPT-4 and beyond involves not only improvements in model architecture and training techniques but also significant changes in how models handle inputs and outputs. These changes enhance the flexibility and utility of the models in various applications, from simple text prediction to complex interaction and data manipulation tasks.

## Chapter 6

# How to Gain Function Calling Capabilities

If we do not have the OpenAI Function Calling API, how can we achieve Function Callings with existing models? Especially with open-source models that lack built-in Function Calling capabilities or with GPT models released before the Function Calling feature.

Before OpenAI released the Function Calling feature in GPT-3.5 and GPT-4, many researchers had already devised various methods for enabling LLMs to call APIs.

Function Calling feature could be gained by three methods: (1) Prompt Engineering, (2) Fine-tuning, and (3) Large Language Model + Planner.

## 6.1 Prompt Engineering

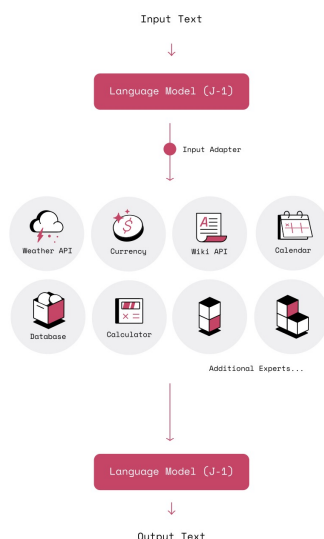


Figure 6.1: MRKL System [49].

If we already have an LLM, but it lacks the capability to call tools automatically, we still could use prompt engineering to gain the Function Calling feature. In the method of Prompt Engineering, the API's description and the work to implement Function Callings are embedded directly into the prompt. The MRKL System [49] is a typical example of this category.

Most implementations using this method also leverage techniques such as Chain of Thought (CoT) or Self-reflection to facilitate the Function Calling process. This approach is particularly useful given the capabilities of LLMs for zero-shot or few-shot learning. However, while these models can handle simple queries effectively, their performance degrades with complex tasks, and they exhibit a high error rate.



## 6.2 Fine-tuning an LLM

Fine-tuning is a more tailored approach where a pretrained model is further trained on a specific dataset that includes examples of Function Callings. This method allows the model to adapt to particular API structures and domain-specific language, enhancing its ability to execute Function Callings accurately.

For instance, a model could be fine-tuned on datasets that simulate banking transactions or weather information retrieval. This adaptation makes the model more robust in handling the nuances of such specialized tasks, thereby reducing error rates and improving response relevance.

Two typical solutions in this category include Toolformer [50] and TALM (Tool Augmented Language Models) [51]. These two solutions differ in their approach to the Function Calling process.

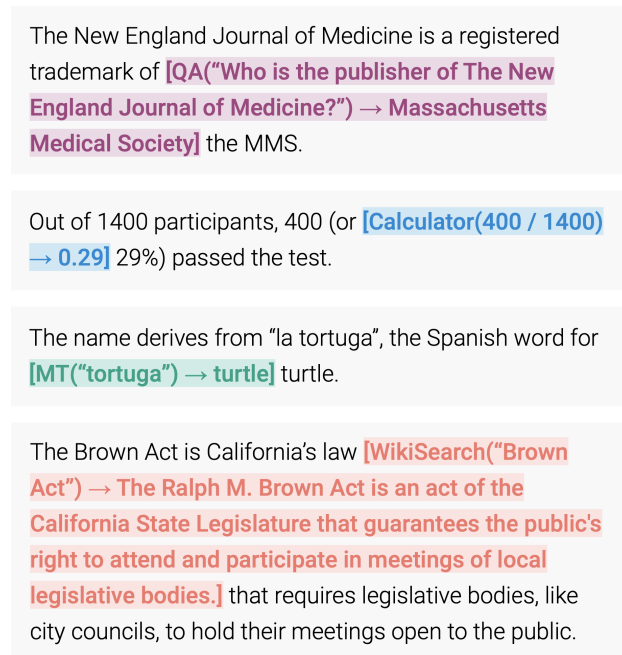


Figure 6.2: Toolformer [50, Figure 1]

The Toolformer solution simply replaces the LLM's output text with the tool's result, eliminating the need for developers to feed the result back into the LLM to generate a user-readable response. In contrast, TALM requires that the result be sent back to the LLM, and then generate a user-readable response; this is the method chosen by OpenAI for its Function Calling feature.

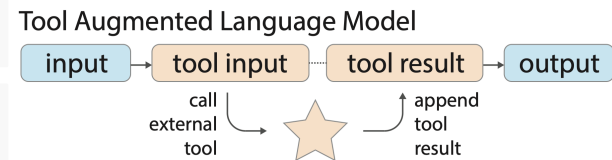


Figure 6.3: TALM (Tool Augmented Language Models) [51, Figure 2]

## 6.3 LLM + Planner

The third approach LLM+ Planner[52] involves integrating a large language model with a specific planner model designed to interpret the model’s responses and execute Function Callings. This planner acts as an intermediary that translates the LLM’s natural language output into actionable API calls.

One significant advantage of using a planner is the separation of concerns: the LLM focuses on understanding and generating human-like responses, while the planner handles the technical execution of API calls. This modularity allows for easier updates and maintenance of the system, as changes in API specifications or capabilities require only modifications to the planner rather than retraining the entire LLM.

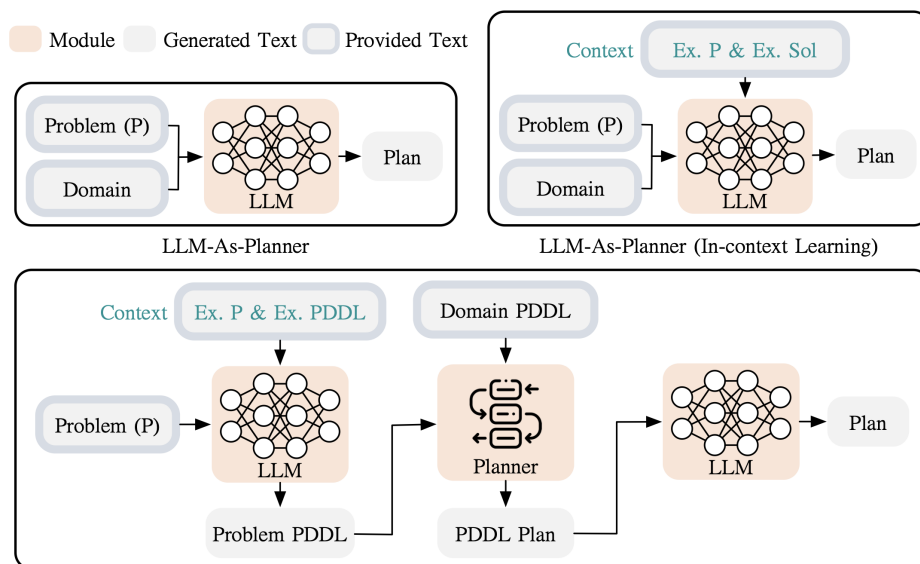


Figure 6.4: LLM + P  
[52, Figure 1]

## 6.4 Conclusion

In conclusion, each of these methods has its strengths and weaknesses. Prompt Engineering is quick and easy to implement but lacks robustness for complex tasks. Fine-tuning offers improved accuracy and specificity at the cost of requiring a large amount of domain-specific training data. Lastly, integrating an LLM with a planner provides a flexible and scalable solution but requires additional system architecture and maintenance. The choice of method depends on the specific requirements and constraints of the application at hand.

# Chapter 7

## A Divide-and-Concur RPA Agent

### 7.1 Introduction

Robotic Process Automation (RPA) ([53],[54],[55]), the cutting-edge automation approach, is a type of software service that helps people build workflows on existing services. It enhances efficiency by automating processes through the integration of various software systems, following meticulously crafted rules to create a streamlined workflow.

RPA utilizes software robots to either automate interactions with software APIs or mimic user GUI actions to complete tasks across various software platforms. Consequently, RPA has recently garnered considerable attention as an efficient technology for automating repetitive, rule-based tasks traditionally handled by humans.

Although RPA has helped people complete their work more efficiently, it still requires detailed human thought and programming concepts, leaving tasks that demand human intelligence dependent on human effort. While RPA workflows can automate execution, their creation relies heavily on human expertise for detailed design. Additionally, many tasks undertaken by humans are characterized by their complexity and adaptability.

Since RPA workflows are just another format of programming structures, we have proposed a method to let LLM to generate Python code and then translate them into RPA workflows in the "*ProAgent: From Robotic Process Automation to Agentic Process Automation*"[1] paper.

In this chapter, we will introduce another method, which will progress with user request with the divide-and-concur method.

## 7.2 Objective

Our primary goal is to translate user requests into functional RPA workflows. For example, consider the following prompt that describes a grade format:

*"The data format is like \$json = {"grade": 90, "name": "Lisa", "email": "lisa@gmail.com"}."*

Additionally, we have a user request such as:

*"Retrieve data from MongoDB and check if the score is greater than or equal to 90. If it is, send the score and name to the #general channel on Slack. If not, send an email through Gmail to remind that the score needs to be improved."*

This user prompt describe a workflow to do an action for each student's transcript records.

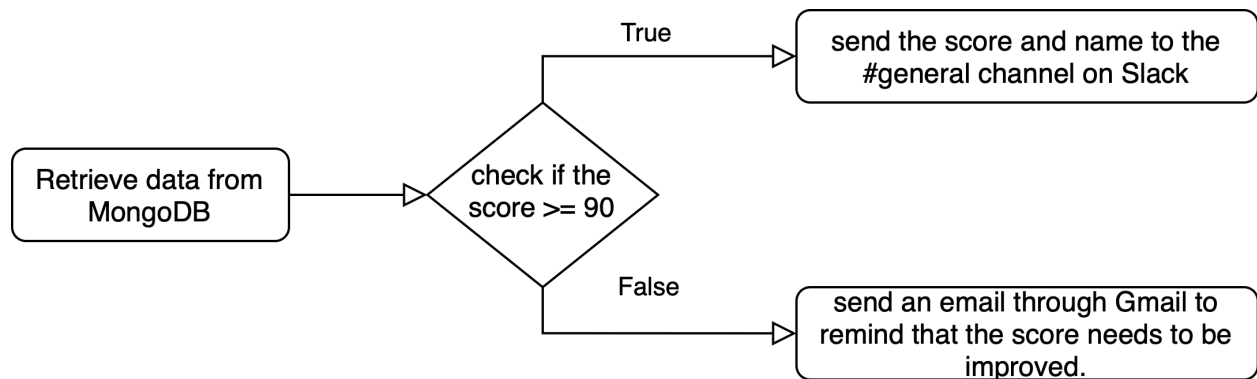


Figure 7.1: RPA Workflow Builder Task Overview

We need translate this prompt into a real n8n[56] workflow dynamically by our RPA agent.

### 7.2.1 Workflow Nodes

The n8n workflows are built with nodes, which is a fixed and single feature units in the RPA system. We listed all the nodes we used in this example.

To address this specific user request, we can utilize a combination of nodes with types including "MongoDB", "Slack", "Gmail", and "if". Here's a detailed breakdown of how this can be achieved:

#### MongoDB Node



**Function:** Retrieve data from MongoDB.

**Operation:** Extract the relevant grade and user details.

### If Node



**Function:** Evaluate the condition.

**Operation:** Check if the score is greater than or equal to 90.

### Slack Node



**Function:** Send a message to Slack.

**Operation:** If the score meets the criteria, send the score and name to the #general channel.

### Gmail Node



**Function:** Send an email.

**Operation:** If the score does not meet the criteria, send an email to remind the user that the score needs improvement.

## 7.3 Methodology

We propose a novel divide-and-conquer method leveraging our LLM (Large Language Model) agent to build a Robotic Process Automation (RPA) workflow based on user requests. The key concept of this methodology is to systematically decompose the user’s request into smaller, manageable components. This step-by-step division ensures that each component can be easily translated into a functional unit, such as an n8n node. Once all the components have been identified and translated into their corresponding nodes, we halt the decomposition process and proceed to assemble these nodes into a cohesive n8n workflow. This approach not only simplifies the complexity of translating user requests into automated workflows but also enhances the accuracy and efficiency of the RPA development process.

### Detailed Process

**Initial User Request:** The process begins with the user submitting a request for a specific RPA workflow. This request is usually a high-level description of the tasks they want to automate.

**Decomposition:** The LLM agent then analyzes the user request and divides it into smaller, more manageable components. Each step of the decomposition is designed to break down complex tasks into simpler, discrete actions.

**Translation into Nodes:** Each of these smaller components is then translated into a functional unit. In the context of n8n, these functional units are represented as nodes, each responsible for a specific task or operation.

**Node Assembly:** After all the components have been translated into nodes, the LLM agent assembles these nodes into a coherent n8n workflow. This involves connecting the nodes in a logical sequence to ensure that the workflow performs the desired automation tasks correctly.

**Verification and Testing:** Once the workflow is built, it undergoes verification and testing to ensure it meets the user’s requirements and performs as expected. Any necessary adjustments are made to refine the workflow.

**Deployment:** Finally, the verified and tested workflow is deployed, allowing the user to benefit from the automated process.

### Advantages

**Efficiency:** By breaking down complex user requests into simpler components, the process becomes more manageable and less prone to errors.

**Scalability:** The divide-and-conquer approach allows for easy scaling, as additional components can be seamlessly integrated into the workflow.

**Flexibility:** Users can request changes or additions to the workflow, which can be accommodated by adjusting the relevant nodes without overhauling the entire system.

**Accuracy:** The systematic translation of user requests into functional units ensures that the final workflow closely aligns with the user’s intentions. This innovative approach leverages the advanced capabilities of our LLM agent to streamline the creation of RPA workflows, making it easier for users to automate their processes efficiently and accurately.

### 7.3.1 Steps

#### 1. Initial Node Determination and Divide into First Part and Rest Part

When the builder encounters a full request from a user, it first determines the type of the initial node. Given that the initial step of the user's request involves using MongoDB to query the transcript, the LLM will return the initial node type as "MongoDB". Using the **choose node type prompt** ("Use function call to choose node type for the user query."), we can determine the appropriate block type for the user query.

Using the **divide into first part and rest part prompt** ("Divide a user query into a sequence first sub query with rest. You just need separate the first step as first\_part, and rest as rest\_part. "), we can divide the full request into two parts:

**the first part** (the MongoDB node: "Retrieve data from MongoDB") and

**the rest part** ("check if the score is greater than or equal to 90. If it is, send the score and name to the #general channel on Slack. If not, send an email through Gmail to remind that the score needs to be improved.").

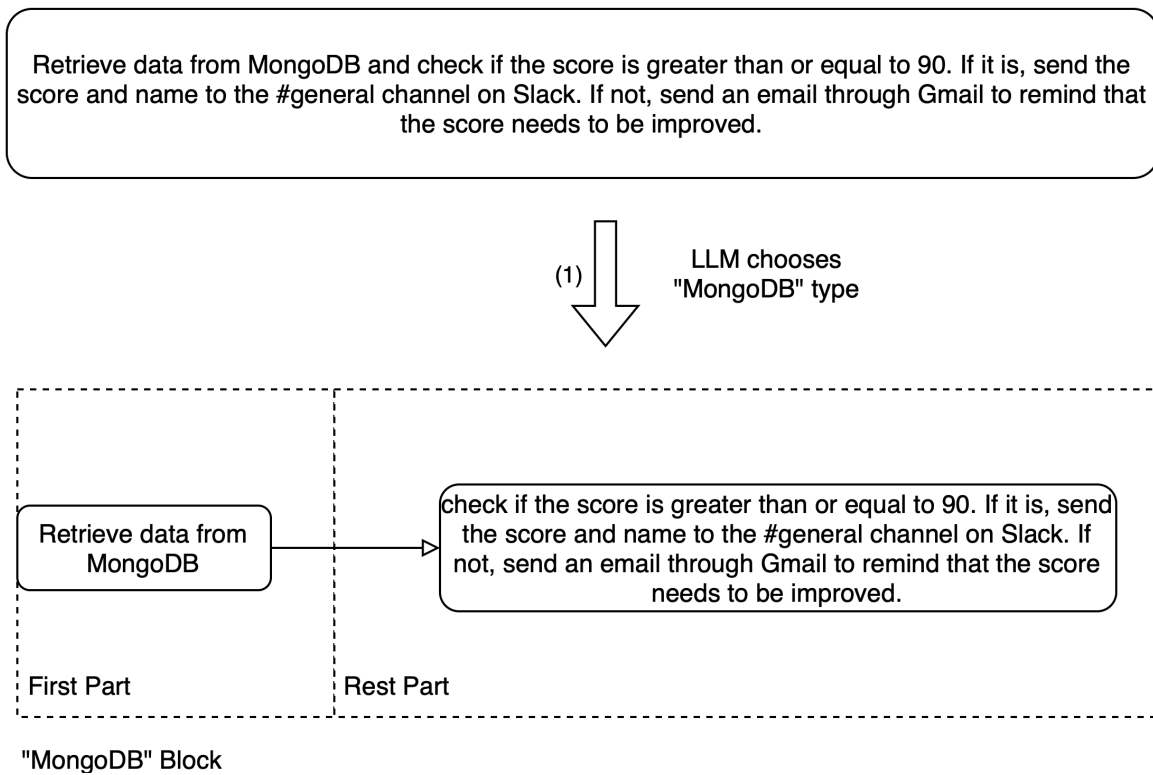


Figure 7.2: RPA Workflow Builder Step (1)



## Listing 7.1: Sequence Query Builder Code

```
1 def query_builder_sequence(raw_query: str, hint: str, llm: OpenAILLM):
2     messages = []
3     FUNCTION_NAME = "divide_into_first_rest"
4     messages += [
5         {"role": "system", "content": f"We need to divide a complex query into first sub
6             query with rest. {hint}"}
7     ]
8     messages += [
9         {"role": "user", "content": f"# User Query:\n{raw_query}"}
10    ]
11    functions = [
12        {
13            "name": FUNCTION_NAME,
14            "description": "Divide a user query into a sequence first sub query with rest.
15                You just need separate the first step as first_block, and rest as rest_block
16                . If it is an unit action, including the action's parameters, e.g. message
17                to send, just leave it as first_block, and rest_block as empty string. You
18                must call this function to response.",
19            "parameters": {
20                "type": "object",
21                "properties": {
22                    "first_block": {
23                        "type": "string",
24                        "description": "first step to do"
25                    },
26                    "rest_block": {
27                        "type": "string",
28                        "description": "after first step. Optional."
29                    },
30                },
31            },
32        },
33    ]
34    content, function_name, function_arguments, message = llm.chat_completion(
35        messages=messages,
36        functions=functions,
37        function_call={"name": FUNCTION_NAME}
38    )
39    if not function_name or not function_arguments:
40        raise KeyError("function_call")
41    assert function_name == FUNCTION_NAME
42
43    first_block = function_arguments.get("first_block")
44    rest_block = function_arguments.get("rest_block") or None
45
46    return first_block, rest_block
```

## 2. If Node Breakdown

For the remaining part of the request, the LLM uses the `choose node type` prompt to determine the node types and divide them into smaller parts. It identifies the current node as an "if" node.

The LLM then constructs an "if" node, dividing the current request text into four parts:

(2a) **If expression part:** "check if the score is greater than or equal to 90",

(2b) **True branch part:** "send the score and name to the #general channel on Slack",

(2c) **False branch part:** "send an email through Gmail to remind that the score needs to be improved."

(2d) **Remainder part:** which is empty in this case).

For the If expression part, the LLM translates it into the expression `$json['grade'] >= 90`. For the remaining parts (True branch, False branch, Remainder), the LLM continues to determine their block types.

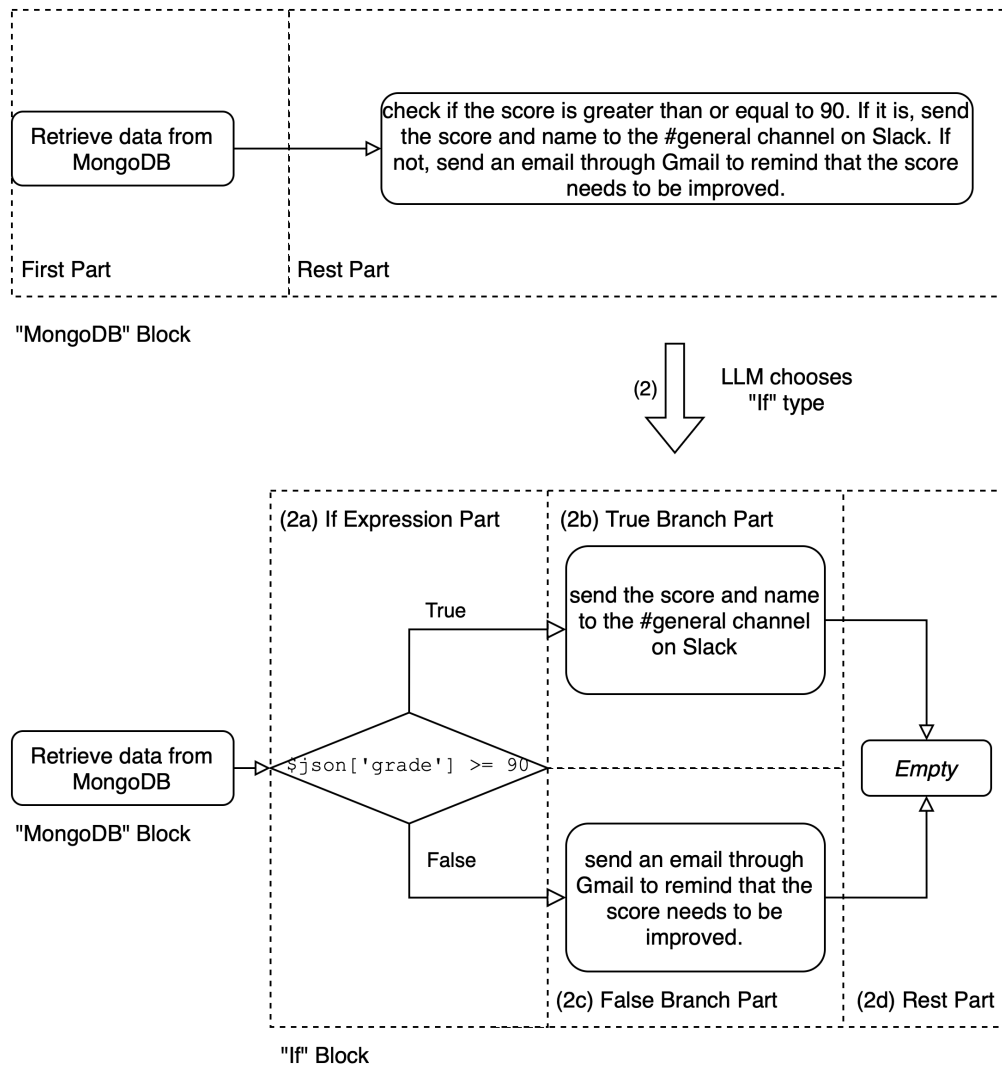


Figure 7.3: RPA Workflow Builder Step (2)

## Listing 7.2: If Branch Query Builder

```
1 def query_builder_if_branch(raw_query: str, llm: OpenAILLM):
2     messages = []
3     messages += [
4         {
5             "role": "system",
6             "content": """
7             We need to divide a complex query into block queries.
8             You should only reply a json object, has a json schema:
9             "if_expression_block": {
10                "type": "string",
11                "description": "if expression part. If a block has nothing to do just
12                    return empty string."
13            },
14            "true_branch_block": {
15                "type": "string",
16                "description": "if true what we should do. If a block has nothing to do
17                    just return empty string."
18            },
19            "false_branch_block": {
20                "type": "string",
21                "description": "if false wha we should do. If a block has nothing to do
22                    just return empty string."
23            },
24            "rest_block": {
25                "type": "string",
26                "description": "after what we should do"
27            },
28            """
29        }
30    ]
31    messages += [
32        {"role": "user", "content": f"### User Query:\n{raw_query}"}
33    ]
34    while True:
35        content, function_name, function_arguments, message = llm.chat_completion(
36            messages=messages
37        )
38        try:
39            content_json = json.loads(content)
40            break
41        except json.JSONDecodeError:
42            continue
43
44    if_expression=str(content_json.get("if_expression_block")) or None
45    true_branch_block=str(content_json.get("true_branch_block")) or None
46    false_branch_block=str(content_json.get("false_branch_block")) or None
47    rest_block=str(content_json.get("rest_block")) or None
48    return if_expression, true_branch_block, false_branch_block, rest_block
```

### 3. True Branch Processing

For the (2b) True branch part, the LLM determines it as a "Slack" block type. The text request "send the score and name to the #general channel on Slack" is translated into an Slack node, as listed in Table 7.1.

<b>Node Type</b>	Slack
<b>Node Sub-Type</b>	send_message
<b>Mode</b>	channel
<b>Channel Id</b>	#general
<b>Content</b>	"Student " + \$json.name + " has received a grade of " + \$json.grade + " in the recent examination."

Table 7.1: LLM translates (2b) into Slack Node

### 4. False Branch Processing

For the (2c) False branch part, the LLM determines it as a "Gmail" block type. The text request "send an email through Gmail to remind that the score needs to be improved." is translated into a Gmail node, as listed in Table 7.2.




<b>Node Type</b>	Gmail
<b>Slack Node Type</b>	send_email
<b>To</b>	\$json.email
<b>Title</b>	"Reminder for " + \$json.name + ", Your grade needs to be improved"
<b>Content</b>	"Dear " + \$json.name + ",  Your current grade is " + \$json.grade + ". There is always room for improvement. Keep up the good work!  Best,  Your Teacher"

Table 7.2: LLM translates (2c) into Gmail Node

## 7.4 Result

The proposed divide-and-conquer methodology, leveraging our Large Language Model (LLM) agent, successfully translated the user request into a functional n8n workflow. The systematic decomposition of the user request into manageable components ensured that each step was clearly defined and accurately executed. The resulting workflow included the necessary nodes for MongoDB data retrieval, conditional evaluation using an If node, and subsequent actions via Slack and Gmail nodes.

The automated workflow was verified and tested, demonstrating its ability to handle the specified tasks accurately and efficiently.

### 7.4.1 Visual Results

**n8n Workflow:** The assembled workflow in n8n, as depicted in 7.4, shows the interconnected nodes representing the various steps of the process.

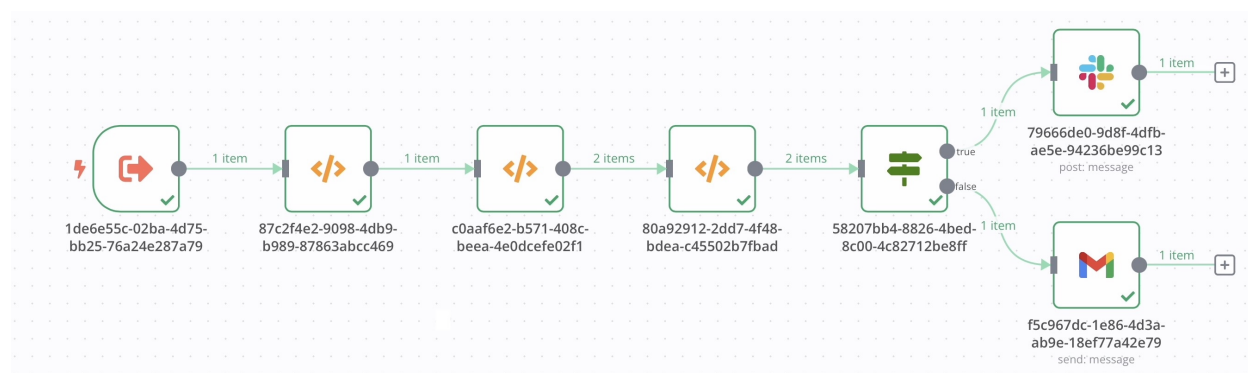


Figure 7.4: RPA Result n8n Workflow

**Gmail Notification:** An example of the email sent through Gmail, shown in 7.5, illustrates the reminder message to students whose scores were below the threshold.

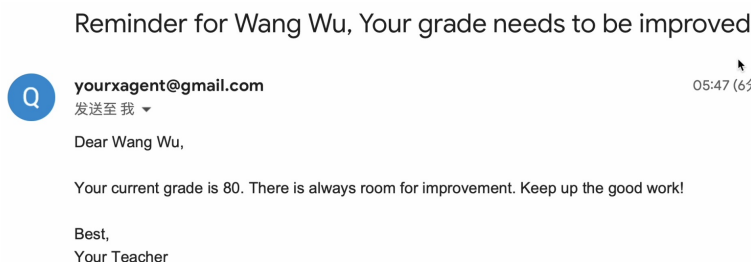


Figure 7.5: RPA Result Gmail

**Slack Notification:** The message posted in the #general Slack channel, as shown in 7.6,

confirms the successful notification of students with satisfactory grades.

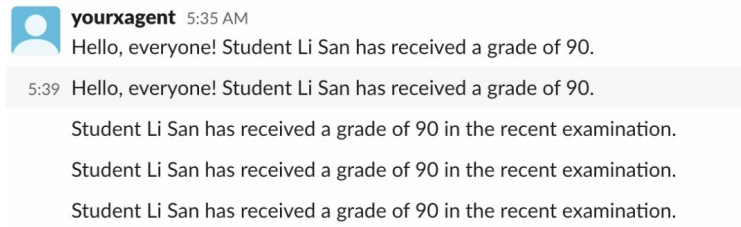


Figure 7.6: RPA Result Slack

## 7.5 Conclusion

The divide-and-conquer methodology, powered by the LLM agent, has proven to be a robust approach for translating complex user requests into functional RPA workflows. The systematic decomposition of tasks ensured that each component was manageable and accurately translated into n8n nodes. The resulting workflow not only met the user's requirements but also demonstrated the following key advantages:

**Efficiency:** The process was streamlined, reducing the likelihood of errors and ensuring quick development.

**Scalability:** Additional components can be easily integrated into the workflow, allowing for future enhancements.

**Flexibility:** The workflow can be adjusted based on user feedback or changing requirements without significant rework.

**Accuracy:** The final workflow closely aligned with the user's intentions, ensuring the desired outcomes were achieved.

Overall, this innovative approach leverages the advanced capabilities of our LLM agent to simplify and enhance the creation of RPA workflows, making it easier for users to automate their processes effectively.

# Chapter 8

## Conclusion

The research presented in this thesis highlights the significant advancements in the field of Large Language Models (LLMs) and their deployment as agents for various applications. By exploring the historical development of language models and their transition to sophisticated neural network architectures, we have established a solid foundation for understanding the capabilities and limitations of LLMs. The study's focus on LLM agents has provided insights into their operational frameworks, including user interactions, environmental considerations, and task planning, which are crucial for effective deployment.

Addressing the limitations of LLM inputs, particularly the context window, the thesis introduces Retrieval-Augmented Generation (RAG) as a viable solution, enhancing the models' ability to handle extensive background knowledge. The exploration of OpenAI's APIs for GPT models has further emphasized the practical applications and versatility of these models in real-world scenarios.

The practical implementation of LLMs in Robotic Process Automation (RPA) through a divide-and-conquer methodology has demonstrated the potential of these models to streamline complex workflows, ensuring efficiency, scalability, flexibility, and accuracy. This innovative approach simplifies the creation of RPA workflows, making automation more accessible and effective for users.

Overall, this thesis confirms the transformative potential of LLMs in automating processes and enhancing user interactions. Future research should continue to explore the evolving capabilities of LLMs, focusing on improving their efficiency, expanding their applications, and addressing emerging challenges in their deployment as intelligent agents.

# References

- [1] Y. Ye, X. Cong, S. Tian, *et al.*, *Proagent: From robotic process automation to agentic process automation*, 2023. arXiv: [2311.10751](#) [cs.R0].
- [2] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” 2018.
- [3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019. URL: <https://api.semanticscholar.org/CorpusID:160025533>.
- [4] T. B. Brown, B. Mann, N. Ryder, *et al.*, *Language models are few-shot learners*, 2020. arXiv: [2005.14165](#) [cs.CL].
- [5] OpenAI, J. Achiam, S. Adler, *et al.*, *Gpt-4 technical report*, 2024. arXiv: [2303.08774](#) [cs.CL].
- [6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, *Bert: Pre-training of deep bidirectional transformers for language understanding*, 2019. arXiv: [1810.04805](#) [cs.CL].
- [7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, *Attention is all you need*, 2023. arXiv: [1706.03762](#) [cs.CL].
- [8] C. R. Jones and B. K. Bergen, *People cannot distinguish gpt-4 from a human in a turing test*, 2024. arXiv: [2405.08007](#) [cs.HC].
- [9] G. Yenduri, R. M, C. S. G, *et al.*, *Generative pre-trained transformer: A comprehensive review on enabling technologies, potential applications, emerging challenges, and future directions*, 2023. arXiv: [2305.10435](#) [cs.CL].
- [10] H. Chen, F. Jiao, X. Li, C. Qin, M. Ravaut, R. Zhao, C. Xiong, and S. Joty, *Chatgpt’s one-year anniversary: Are open-source large language models catching up?* 2024. arXiv: [2311.16989](#) [cs.CL].
- [11] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, *Gorilla: Large language model connected with massive apis*, 2023. arXiv: [2305.15334](#) [cs.CL].
- [12] Y. Qin, S. Liang, Y. Ye, *et al.*, *Toollm: Facilitating large language models to master 16000+ real-world apis*, 2023. arXiv: [2307.16789](#) [cs.AI].
- [13] R. Rosenfeld, “Two decades of statistical language modeling: Where do we go from here?” *Proceedings of the IEEE*, vol. 88, no. 8, pp. 1270–1278, 2000. DOI: [10.1109/5.880083](#).



- [14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, *Efficient estimation of word representations in vector space*, 2013. arXiv: [1301.3781 \[cs.CL\]](#).
- [15] L. R. Medsker, L. Jain, *et al.*, “Recurrent neural networks,” *Design and Applications*, vol. 5, no. 64-67, p. 2, 2001.
- [16] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997. DOI: [10.1162/neco.1997.9.8.1735](#).
- [17] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, *Learning phrase representations using rnn encoder-decoder for statistical machine translation*, 2014. arXiv: [1406.1078 \[cs.CL\]](#).
- [18] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, *Zero-shot text-to-image generation*, 2021. arXiv: [2102.12092 \[cs.CV\]](#).
- [19] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, *High-resolution image synthesis with latent diffusion models*, 2022. arXiv: [2112.10752 \[cs.CV\]](#).
- [20] OpenAI. “Sora.” (2024), URL: <https://openai.com/index/sora/> (visited on 02/15/2024).
- [21] Suno. “Suno.” (2023), URL: <https://suno.com> (visited on 12/20/2023).
- [22] OpenAI. “Hello gpt-4o.” (2024), URL: <https://openai.com/index/hello-gpt-4o/> (visited on 05/13/2024).
- [23] L. Weng, “Llm-powered autonomous agents,” *lilianweng.github.io*, Jun. 2023. URL: <https://lilianweng.github.io/posts/2023-06-23-agent/>.
- [24] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, *Chain-of-thought prompting elicits reasoning in large language models*, 2023. arXiv: [2201.11903 \[cs.CL\]](#).
- [25] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, *Tree of thoughts: Deliberate problem solving with large language models*, 2023. arXiv: [2305.10601 \[cs.CL\]](#).
- [26] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone, *Llm+p: Empowering large language models with optimal planning proficiency*, 2023. arXiv: [2304.11477 \[cs.AI\]](#).
- [27] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, *React: Synergizing reasoning and acting in language models*, 2023. arXiv: [2210.03629 \[cs.CL\]](#).
- [28] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao, *Reflexion: Language agents with verbal reinforcement learning*, 2023. arXiv: [2303.11366 \[cs.AI\]](#).
- [29] H. Liu, C. Sferrazza, and P. Abbeel, *Chain of hindsight aligns language models with feedback*, 2023. arXiv: [2302.02676 \[cs.LG\]](#).
- [30] OpenAI. “How openai assistants work.” (2023), URL: <https://platform.openai.com/docs/assistants/how-it-works> (visited on 06/19/2024).
- [31] Microsoft. “Gpt assistant agent.” (2023), URL: [https://microsoft.github.io/autogen/docs/topics/openai-assistant/gpt\\_assistant\\_agent/](https://microsoft.github.io/autogen/docs/topics/openai-assistant/gpt_assistant_agent/) (visited on 06/19/2024).

- [32] O. Community. “Assistants api: Multi-assistant agentic workflow.” (2023), URL: <https://community.openai.com/t/assistants-api-multi-assistant-agentic-workflow/707742> (visited on 06/19/2024).
- [33] NVIDIA. “Building your first llm agent application.” (2023), URL: <https://developer.nvidia.com/blog/building-your-first-llm-agent-application/> (visited on 06/19/2024).
- [34] P. Guide. “Llm agents research.” (2023), URL: <https://www.promptingguide.ai/research/llm-agents> (visited on 06/19/2024).
- [35] ProjectPro. “Llm agents.” (2023), URL: <https://www.projectpro.io/article/llm-agents/1013> (visited on 06/19/2024).
- [36] OpenAI. “Models overview.” (2024), URL: <https://platform.openai.com/docs/models/gpt-4-turbo-and-gpt-4> (visited on 06/19/2024).
- [37] P. Lewis, E. Perez, A. Piktus, *et al.*, *Retrieval-augmented generation for knowledge-intensive nlp tasks*, 2021. arXiv: [2005.11401](https://arxiv.org/abs/2005.11401) [cs.CL].
- [38] OpenAI. “Text completions api guide.” (2024), URL: <https://platform.openai.com/docs/guides/text-generation/completions-api> (visited on 06/19/2024).
- [39] OpenAI. “Text completions api playground.” (2024), URL: <https://platform.openai.com/playground/complete> (visited on 06/19/2024).
- [40] OpenAI. “Chat completions api guide.” (2024), URL: <https://platform.openai.com/docs/guides/text-generation/chat-completions-api> (visited on 06/19/2024).
- [41] Microsoft. “How to work with the chat markup language (preview).” (2024), URL: <https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/chat-markup-language> (visited on 06/19/2024).
- [42] OpenAI. “Function calling api guide.” (2024), URL: <https://platform.openai.com/docs/guides/function-calling> (visited on 06/19/2024).
- [43] OpenAI. “Function calling and other api updates.” (2024), URL: <https://www.openai.com/index/function-calling-and-other-api-updates> (visited on 06/19/2024).
- [44] AI Moda. “Function calling to system prompts [openai internals].” (2024), URL: <https://www.ai.moda/en/blog/openai-function-calling-to-system-prompt> (visited on 06/19/2024).
- [45] OpenAI. “Assistants api guide.” (2024), URL: <https://platform.openai.com/docs/assistants/overview> (visited on 06/19/2024).
- [46] OpenAI. “New models and developer products announced at devday.” (2024), URL: [https://openai.com/index/new-models-and-developer-products-announced-at-devday/#\\_u4705USJmf1PSClw60386](https://openai.com/index/new-models-and-developer-products-announced-at-devday/#_u4705USJmf1PSClw60386) (visited on 06/19/2024).
- [47] OpenAI. “Introducing gpts.” (2024), URL: <https://openai.com/index/introducing-gpts/> (visited on 06/19/2024).
- [48] OpenAI. “Assistants api playground.” (2024), URL: <https://platform.openai.com/playground/assistants> (visited on 06/19/2024).

- [49] E. Karpas, O. Abend, Y. Belinkov, *et al.*, *Mrkl systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning*, 2022. arXiv: [2205.00445](#) [[cs.CL](#)].
- [50] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom, *Toolformer: Language models can teach themselves to use tools*, 2023. arXiv: [2302.04761](#) [[cs.CL](#)].
- [51] A. Parisi, Y. Zhao, and N. Fiedel, *Talm: Tool augmented language models*, 2022. arXiv: [2205.12255](#) [[cs.CL](#)].
- [52] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone, *Llm+p: Empowering large language models with optimal planning proficiency*, 2023. arXiv: [2304.11477](#) [[cs.AI](#)].
- [53] L. Ivančić, D. Suša Vugec, and V. Vuksic, “Robotic process automation: Systematic literature review,” in Aug. 2019, pp. 280–295, ISBN: 978-3-030-30428-7. DOI: [10.1007/978-3-030-30429-4\\_19](#).
- [54] J. Wewerka and M. Reichert, *Robotic process automation – a systematic literature review and assessment framework*, 2020. arXiv: [2012.11951](#) [[cs.R0](#)].
- [55] A. Tiwari, C. Turner, and B. Majeed, “A review of business process mining: State-of-the-art and future trends,” *Business Process Management Journal*, vol. 14, pp. 5–22, Feb. 2008. DOI: [10.1108/14637150810849373](#).
- [56] n8n. “N8n - a powerful workflow automation tool.” (2024), URL: <https://n8n.io> (visited on 06/19/2024).