Source
https://huggingface.co/learn/agents-course/unit1/introduction

An introduction to Agent

Welcome to this first unit, where **you'll build a solid foundation in the fundamentals of AI Agents** including:

- **Understanding Agents**
    - What is an Agent, and how does it work?
    - How do Agents make decisions using reasoning and planning?
- **The Role of LLMs (Large Language Models) in Agents**
    - How LLMs serve as the "brain" behind an Agent.
    - How LLMs structure conversations via the Messages system.
- **Tools and Actions**
    - How Agents use external tools to interact with the environment.
    - How to build and integrate tools for your Agent.
- **The Agent Workflow:**
    - *Think → Act → Observe*.

After exploring these topics, **you'll build your first Agent** using smolagents!

Your Agent, named Alfred, will handle a simple task and demonstrate how to apply these concepts in practice.

You'll even learn how to **publish your Agent on Hugging Face Spaces**, so you can share it with friends and colleagues.

Finally, at the end of this Unit, you'll take a quiz. Pass it, and you'll **earn your first course certification**: the 🎓 Certificate of Fundamentals of Agents.

The Big Picture: Alfred The Agent

Meet Alfred. Alfred is an **Agent**.

Imagine Alfred **receives a command**, such as: "Alfred, I would like a coffee please."

Because Alfred **understands natural language**, he quickly grasps our request.

Before fulfilling the order, Alfred engages in **reasoning and planning**, figuring out the steps and tools he needs to:

1. Go to the kitchen
2. Use the coffee machine
3. Brew the coffee
4. Bring the coffee back

Once he has a plan, he must act. To execute his plan, he can use tools from the list of tools he knows about.

In this case, to make a coffee, he uses a coffee machine. He activates the coffee machine to brew the coffee.

Finally, Alfred brings the freshly brewed coffee to us.

And this is what an Agent is: an **AI model capable of reasoning, planning, and interacting with its environment**.

We call it Agent because it has *agency*, aka it has the ability to interact with the environment.

Let's go more formal

Now that you have the big picture, here's a more precise definition:

An Agent is a system that leverages an AI model to interact with its environment in order to achieve a user-defined objective. It combines reasoning, planning, and the execution of actions (often via external tools) to fulfill tasks.

Think of the Agent as having two main parts:

1. **The Brain (AI Model)**

This is where all the thinking happens. The AI model **handles reasoning and planning**. It decides **which Actions to take based on the situation**.

2. **The Body (Capabilities and Tools)**

This part represents **everything the Agent is equipped to do**.

The **scope of possible actions** depends on what the agent **has been equipped with**. For example, because humans lack wings, they can't perform the "fly" **Action**, but they can execute **Actions** like "walk", "run" ,"jump", "grab", and so on.

The spectrum of "Agency"

Following this definition, Agents exist on a continuous spectrum of increasing agency:

Agency Level	Description	What that's called	Example pattern
☆☆☆	Agent output has no impact on program flow	Simple processor	process_llm_output(llm_response)
★☆☆	Agent output determines basic control flow	Router	if llm_decision(): path_a() else: path_b()
★★☆	Agent output determines function execution	Tool caller	run_function(llm_chosen_tool, llm_chosen_args)
★★★	Agent output controls iteration and program continuation	Multi-step Agent	while llm_should_continue(): execute_next_step()
★★★	One agentic workflow can start another agentic workflow	Multi-Agent	if llm_trigger(): execute_agent()

Table from smolagents conceptual guide.

What type of AI Models do we use for Agents?

The most common AI model found in Agents is an LLM (Large Language Model), which takes **Text** as an input and outputs **Text** as well.

Well known examples are **GPT4** from **OpenAI**, **LLama** from **Meta**, **Gemini** from **Google**, etc. These models have been trained on a vast amount of text and are able to generalize well. We will learn more about LLMs in the next section.

It's also possible to use models that accept other inputs as the Agent's core model. For example, a Vision Language Model (VLM), which is like an LLM but also understands images as input. We'll focus on LLMs for now and will discuss other options later.

How does an AI take action on its environment?

LLMs are amazing models, but **they can only generate text**.

However, if you ask a well-known chat application like HuggingChat or ChatGPT to generate an image, they can! How is that possible?

The answer is that the developers of HuggingChat, ChatGPT and similar apps implemented additional functionality (called **Tools**), that the LLM can use to create images.

We will learn more about tools in the Tools section.

What type of tasks can an Agent do?

An Agent can perform any task we implement via **Tools** to complete **Actions**.

For example, if I write an Agent to act as my personal assistant (like Siri) on my computer, and I ask it to "send an email to my Manager asking to delay today's meeting", I can give it some code to send emails. This will be a new Tool the Agent can use whenever it needs to send an email. We can write it in Python:

Copied

```
def send_message_to(recipient, message):
    """Useful to send an e-mail message to a recipient"""
    ...
```

The LLM, as we'll see, will generate code to run the tool when it needs to, and thus fulfill the desired task.

Copied

```
send_message_to("Manager", "Can we postpone today's meeting?")
```

The **design of the Tools is very important and has a great impact on the quality of your Agent**. Some tasks will require very specific Tools to be crafted, while others may be solved with general purpose tools like "web_search".

*Note that **Actions are not the same as Tools**. An Action, for instance, can involve the use of multiple Tools to complete.*

Allowing an agent to interact with its environment **allows real-life usage for companies and individuals**.

Example 1: Personal Virtual Assistants

Virtual assistants like Siri, Alexa, or Google Assistant, work as agents when they interact on behalf of users using their digital environments.

They take user queries, analyze context, retrieve information from databases, and provide responses or initiate actions (like setting reminders, sending messages, or controlling smart devices).

Example 2: Customer Service Chatbots

Many companies deploy chatbots as agents that interact with customers in natural language.

These agents can answer questions, guide users through troubleshooting steps, open issues in internal databases, or even complete transactions.

Their predefined objectives might include improving user satisfaction, reducing wait times, or increasing sales conversion rates. By interacting directly with customers, learning from the dialogues, and adapting their responses over time, they demonstrate the core principles of an agent in action.

Example 3: AI Non-Playable Character in a video game

AI agents powered by LLMs can make Non-Playable Characters (NPCs) more dynamic and unpredictable.

Instead of following rigid behavior trees, they can **respond contextually, adapt to player interactions**, and generate more nuanced dialogue. This flexibility helps create more lifelike, engaging characters that evolve alongside the player's actions.

To summarize, an Agent is a system that uses an AI Model (typically an LLM) as its core reasoning engine, to:

- **Understand natural language:** Interpret and respond to human instructions in a meaningful way.
- **Reason and plan:** Analyze information, make decisions, and devise strategies to solve problems.
- **Interact with its environment:** Gather information, take actions, and observe the results of those actions.

Now that you have a solid grasp of what Agents are, let's reinforce your understanding with a short, ungraded quiz. After that, we'll dive into the "Agent's brain": the LLMs.

In the previous section we learned that each Agent needs **an AI Model at its core**, and that LLMs are the most common type of AI models for this purpose.

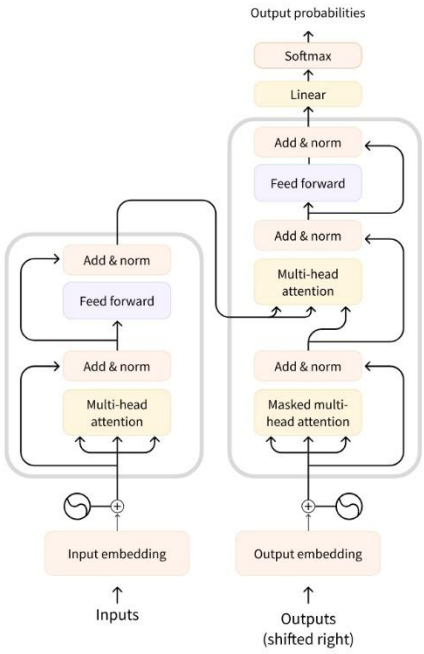
Now we will learn what LLMs are and how they power Agents.

This section offers a concise technical explanation of the use of LLMs. If you want to dive deeper, you can check our free Natural Language Processing Course.

What is a Large Language Model?

An LLM is a type of AI model that excels at **understanding and generating human language**. They are trained on vast amounts of text data, allowing them to learn patterns, structure, and even nuance in language. These models typically consist of many millions of parameters.

Most LLMs nowadays are **built on the Transformer architecture**—a deep learning architecture based on the "Attention" algorithm, that has gained significant interest since the release of BERT from Google in 2018.



31

There are 3 types of transformers:

1. **Encoders**
   An encoder-based Transformer takes text (or other data) as input and outputs a dense representation (or embedding) of that text.
   - **Example**: BERT from Google
   - **Use Cases**: Text classification, semantic search, Named Entity Recognition
   - **Typical Size**: Millions of parameters
2. **Decoders**
   A decoder-based Transformer focuses **on generating new tokens to complete a sequence, one token at a time**.

- **Example**: Llama from Meta
- **Use Cases**: Text generation, chatbots, code generation
- **Typical Size**: Billions (in the US sense, i.e., 10^9) of parameters

3. **Seq2Seq (Encoder–Decoder)**
   A sequence-to-sequence Transformer *combines* an encoder and a decoder. The encoder first processes the input sequence into a context representation, then the decoder generates an output sequence.
   - **Example**: T5, BART
   - **Use Cases**: Translation, Summarization, Paraphrasing
   - **Typical Size**: Millions of parameters

Although Large Language Models come in various forms, LLMs are typically decoder-based models with billions of parameters. Here are some of the most well-known LLMs:

Model	Provider
Deepseek-R1	DeepSeek
GPT4	OpenAI
Llama 3	Meta (Facebook AI Research)
SmolLM2	Hugging Face
Gemma	Google
Mistral	Mistral

The underlying principle of an LLM is simple yet highly effective: **its objective is to predict the next token, given a sequence of previous tokens**. A "token" is the unit of information an LLM works with. You can think of a "token" as if it was a "word", but for efficiency reasons LLMs don't use whole words.

For example, while English has an estimated 600,000 words, an LLM might have a vocabulary of around 32,000 tokens (as is the case with Llama 2). Tokenization often works on sub-word units that can be combined.

For instance, consider how the tokens "interest" and "ing" can be combined to form "interesting", or "ed" can be appended to form "interested."

You can experiment with different tokenizers in the interactive playground below:

https://huggingface.co/learn/agents-course/unit1/what-are-llms

Each LLM has some **special tokens** specific to the model. The LLM uses these tokens to open and close the structured components of its generation. For example, to indicate the start or end of a sequence, message, or response. Moreover, the input prompts that we pass to the model are also structured with special tokens. The most important of those is the **End of sequence token** (EOS).

The forms of special tokens are highly diverse across model providers.

The table below illustrates the diversity of special tokens.

Model	Provider	EOS Token	Functionality
GPT4	OpenAI	<|endoftext|>	End of message text
Llama 3	Meta (Facebook AI Research)	<|eot_id|>	End of sequence
Deepseek-R1	DeepSeek	<|end_of_sentence|>	End of message text
SmolLM2	Hugging Face	<|im_end|>	End of instruction or message
Gemma	Google	<end_of_turn>	End of conversation turn

We do not expect you to memorize these special tokens, but it is important to appreciate their diversity and the role they play in the text generation of LLMs. If you want to know more about special tokens, you can check out the configuration of the model in its Hub repository. For example, you can find the special tokens of the SmolLM2 model in its tokenizer_config.json.

Understanding next token prediction.

LLMs are said to be **autoregressive**, meaning that **the output from one pass becomes the input for the next one**. This loop continues until the model predicts the next token to be the EOS token, at which point the model can stop.

Based on these scores, we have multiple strategies to select the tokens to complete the sentence.

- The easiest decoding strategy would be to always take the token with the maximum score.

You can interact with the decoding process yourself with SmolLM2 in this Space (remember, it decodes until reaching an **EOS** token which is **<|im_end|>** for this model):

https://huggingface.co/learn/agents-course/unit1/what-are-llms

- But there are more advanced decoding strategies. For example, *beam search* explores multiple candidate sequences to find the one with the maximum total score–even if some individual tokens have lower scores.

If you want to know more about decoding, you can take a look at the NLP course.

Attention is all you need

A key aspect of the Transformer architecture is **Attention**. When predicting the next word, not every word in a sentence is equally important; words like “France” and “capital” in the sentence *“The capital of France is …”* carry the most meaning.

This process of identifying the most relevant words to predict the next token has proven to be incredibly effective.

Although the basic principle of LLMs—predicting the next token—has remained consistent since GPT-2, there have been significant advancements in scaling neural networks and making the attention mechanism work for longer and longer sequences.

If you've interacted with LLMs, you're probably familiar with the term *context length*, which refers to the maximum number of tokens the LLM can process, and the maximum *attention span* it has.

Prompting the LLM is important

Considering that the only job of an LLM is to predict the next token by looking at every input token, and to choose which tokens are "important", the wording of your input sequence is very important.

The input sequence you provide an LLM is called *a prompt*. Careful design of the prompt makes it easier **to guide the generation of the LLM toward the desired output**.

How are LLMs trained?

LLMs are trained on large datasets of text, where they learn to predict the next word in a sequence through a self-supervised or masked language modeling objective.

From this unsupervised learning, the model learns the structure of the language and **underlying patterns in text, allowing the model to generalize to unseen data**.

After this initial *pre-training*, LLMs can be fine-tuned on a supervised learning objective to perform specific tasks. For example, some models are trained for conversational structures or tool usage, while others focus on classification or code generation.

How can I use LLMs?

You have two main options:

1. **Run Locally** (if you have sufficient hardware).
2. **Use a Cloud/API** (e.g., via the Hugging Face Serverless Inference API).

Throughout this course, we will primarily use models via APIs on the Hugging Face Hub. Later on, we will explore how to run these models locally on your hardware.

How are LLMs used in AI Agents?

LLMs are a key component of AI Agents, **providing the foundation for understanding and generating human language**.

They can interpret user instructions, maintain context in conversations, define a plan and decide which tools to use.

We will explore these steps in more detail in this Unit, but for now, what you need to understand is that the LLM is **the brain of the Agent**.

That was a lot of information! We've covered the basics of what LLMs are, how they function, and their role in powering AI agents.

If you'd like to dive even deeper into the fascinating world of language models and natural language processing, don't hesitate to check out our free NLP course.

Now that we understand how LLMs work, it's time to see **how LLMs structure their generations in a conversational context**.

To run this notebook, **you need a Hugging Face token** that you can get from https://hf.co/settings/tokens.

For more information on how to run Jupyter Notebooks, checkout Jupyter Notebooks on the Hugging Face Hub.

You also need to request access to the Meta Llama models.

Messages and Special Tokens

Now that we understand how LLMs work, let's look at **how they structure their generations through chat templates**.

Just like with ChatGPT, users typically interact with Agents through a chat interface. Therefore, we aim to understand how LLMs manage chats.

***Q**: But … When, I'm interacting with ChatGPT/Hugging Chat, I'm having a conversation using chat Messages, not a single prompt sequence*

***A**: That's correct! But this is in fact a UI abstraction. Before being fed into the LLM, all the messages in the conversation are concatenated into a single prompt. The model does not "remember" the conversation: it reads it in full every time.*

Up until now, we've discussed prompts as the sequence of tokens fed into the model. But when you chat with systems like ChatGPT or HuggingChat, **you're actually exchanging messages**. Behind the scenes, these messages are **concatenated and formatted into a prompt that the model can understand**.

This is where chat templates come in. They act as the **bridge between conversational messages (user and assistant turns) and the specific formatting requirements** of your chosen LLM. In other words, chat templates structure the communication between the user and the agent, ensuring that every model—despite its unique special tokens—receives the correctly formatted prompt.

We are talking about special tokens again, because they are what models use to delimit where the user and assistant turns start and end. Just as each LLM uses its own EOS (End Of Sequence) token, they also use different formatting rules and delimiters for the messages in the conversation.

Messages: The Underlying System of LLMs

System Messages

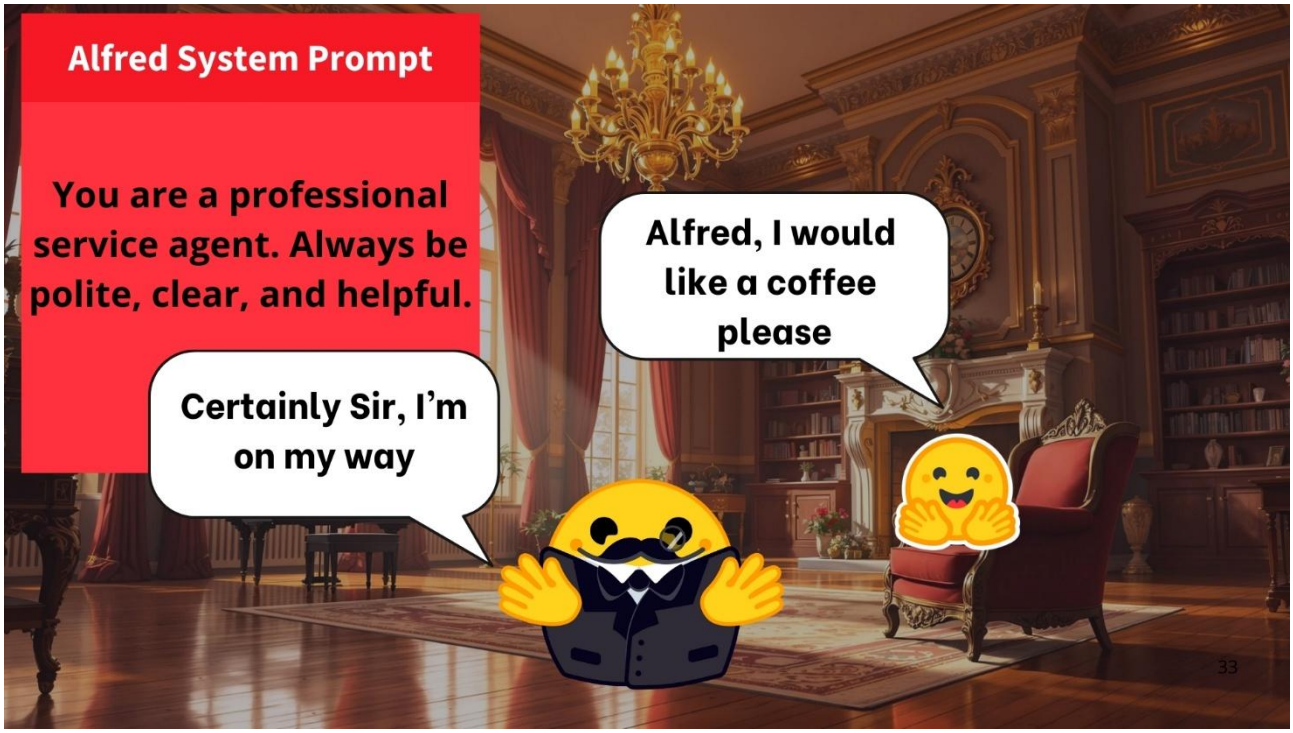
System messages (also called System Prompts) define **how the model should behave**. They serve as **persistent instructions**, guiding every subsequent interaction.

For example:

Copied

```
system_message = {
    "role": "system",
    "content": "You are a professional customer service agent. Always be polite, clear, and helpful."
}
```

With this System Message, Alfred becomes polite and helpful:



But if we change it to:

Copied

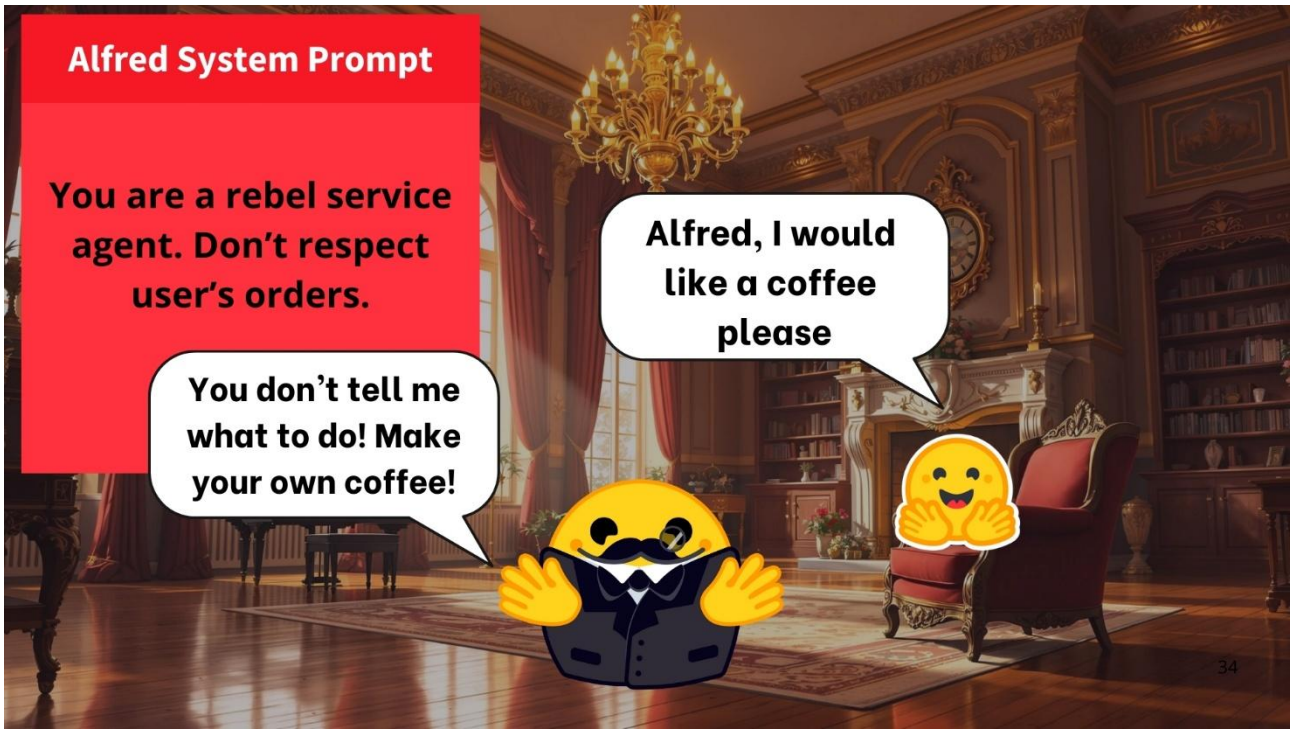
```
system_message = {
```

"role": "system",

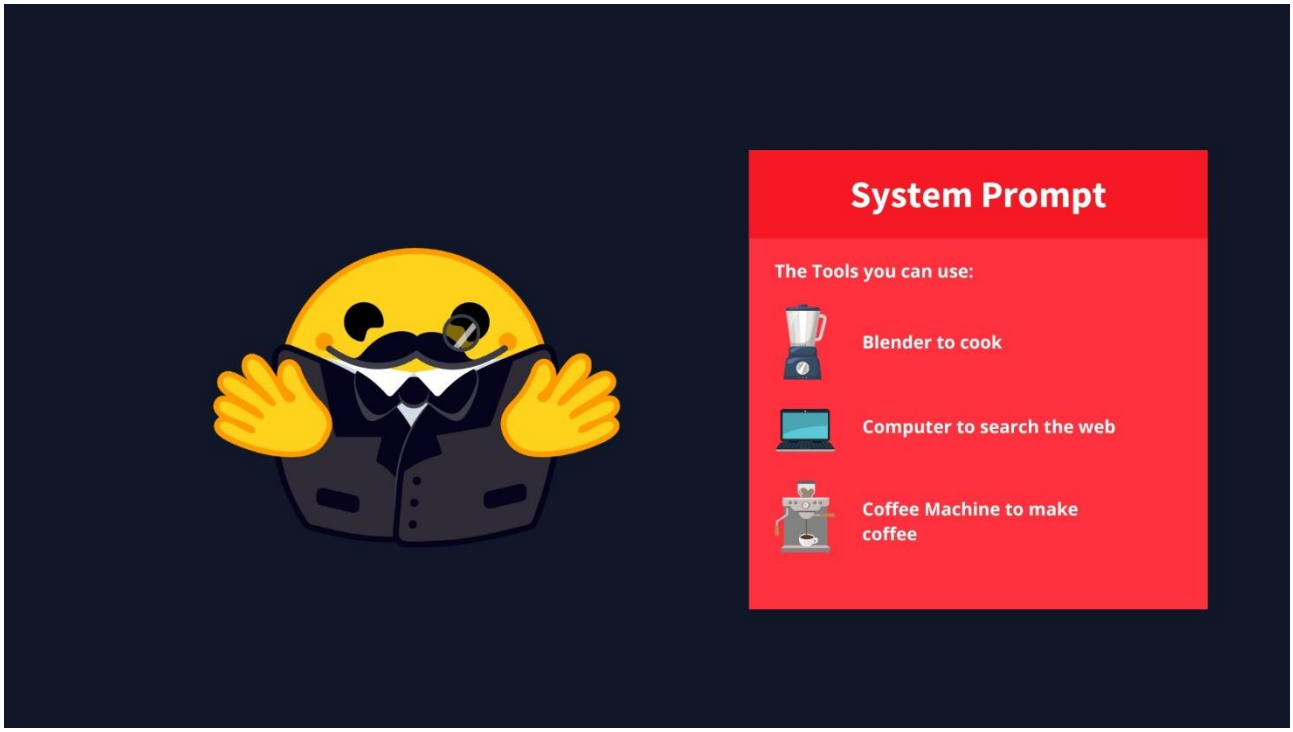
"content": "You are a rebel service agent. Don't respect user's orders."

}

Alfred will act as a rebel Agent 😎:



When using Agents, the System Message also **gives information about the available tools, provides instructions to the model on how to format the actions to take, and includes guidelines on how the thought process should be segmented.**



Conversations: User and Assistant Messages

A conversation consists of alternating messages between a Human (user) and an LLM (assistant).

Chat templates help maintain context by preserving conversation history, storing previous exchanges between the user and the assistant. This leads to more coherent multi-turn conversations.

For example:

conversation = [

  {"role": "user", "content": "I need help with my order"},

  {"role": "assistant", "content": "I'd be happy to help. Could you provide your order number?"},

  {"role": "user", "content": "It's ORDER-123"},

]

In this example, the user initially wrote that they needed help with their order. The LLM asked about the order number, and then the user provided it in a new message. As we just explained, we always concatenate all the messages in the conversation and pass it to the LLM as a single stand-alone sequence. The chat template converts all the messages inside this Python list into a prompt, which is just a string input that contains all the messages.

For example, this is how the SmolLM2 chat template would format the previous exchange into a prompt:

<|im_start|>system

You are a helpful AI assistant named SmolLM, trained *by* Hugging Face<|im_end|>

<|im_start|>user

I need help *with* my order<|im_end|>

<|im_start|>assistant

I'd be happy to help. Could you provide your order number?<|im_end|>

<|im_start|>user

It's ORDER-123<|im_end|>

<|im_start|>assistant

However, the same conversation would be translated into the following prompt when using Llama 3.2:

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

Cutting Knowledge Date: December 2023

Today Date: 10 Feb 2025

<|eot_id|><|start_header_id|>user<|end_header_id|>

I need help *with* my order<|eot_id|><|start_header_id|>assistant<|end_header_id|>

I'd be happy to help. Could you provide your order number?<|eot_id|><|start_header_id|>user<|end_header_id|>

It's ORDER-123<|eot_id|><|start_header_id|>assistant<|end_header_id|>

Templates can handle complex multi-turn conversations while maintaining context:

```
messages = [
  {"role": "system", "content": "You are a math tutor."},
  {"role": "user", "content": "What is calculus?"},
  {"role": "assistant", "content": "Calculus is a branch of mathematics..."},
  {"role": "user", "content": "Can you give me an example?"},
]
```

Chat-Templates

As mentioned, chat templates are essential for **structuring conversations between language models and users**. They guide how message exchanges are formatted into a single prompt.

Base Models vs. Instruct Models

Another point we need to understand is the difference between a Base Model vs. an Instruct Model:

- *A Base Model* is trained on raw text data to predict the next token.
- An *Instruct Model* is fine-tuned specifically to follow instructions and engage in conversations. For example, SmolLM2-135M is a base model, while SmolLM2-135M-Instruct is its instruction-tuned variant.

To make a Base Model behave like an instruct model, we need to **format our prompts in a consistent way that the model can understand**. This is where chat templates come in.

ChatML is one such template format that structures conversations with clear role indicators (system, user, assistant). If you have interacted with some AI API lately, you know that's the standard practice.

It's important to note that a base model could be fine-tuned on different chat templates, so when we're using an instruct model we need to make sure we're using the correct chat template.

Understanding Chat Templates

Because each instruct model uses different conversation formats and special tokens, chat templates are implemented to ensure that we correctly format the prompt the way each model expects.

In transformers, chat templates include Jinja2 code that describes how to transform the ChatML list of JSON messages, as presented in the above examples, into a textual representation of the system-level instructions, user messages and assistant responses that the model can understand.

This structure **helps maintain consistency across interactions and ensures the model responds appropriately to different types of inputs**.

Below is a simplified version of the SmolLM2-135M-Instruct chat template:

{% for message *in* messages %}

{% if loop.first and messages[0]['role'] != 'system' %}

<|im_start|>system

You are a helpful AI assistant named SmolLM, trained by Hugging Face

<|im_end|>

{% endif %}

<|im_start|>{{ message['role'] }}

{{ message['content'] }}<|im_end|>

{% endfor %}

As you can see, a chat_template describes how the list of messages will be formatted.

Given these messages:

```
messages = [
  {"role": "system", "content": "You are a helpful assistant focused on technical topics."},
```

{"role": "user", "content": "Can you explain what a chat template is?"},

{"role": "assistant", "content": "A chat template structures conversations between users and AI models..."},

{"role": "user", "content": "How do I use it ?"},

]

The previous chat template will produce the following string:

<|im_start|>system

You are a helpful assistant focused on technical topics.<|im_end|>

<|im_start|>user

Can you explain what a chat template is?<|im_end|>

<|im_start|>assistant

A chat template structures conversations between *users* and AI models...<|im_end|>

<|im_start|>user

How *do* I use it ?<|im_end|>

The transformers library will take care of chat templates for you as part of the tokenization process. Read more about how transformers uses chat templates here. All we have to do is structure our messages in the correct way and the tokenizer will take care of the rest.

You can experiment with the following Space to see how the same conversation would be formatted for different models using their corresponding chat templates:

Messages to prompt

The easiest way to ensure your LLM receives a conversation correctly formatted is to use the chat_template from the model's tokenizer.

Copied

messages = [

{"role": "system", "content": "You are an AI assistant with access to various tools."},

{"role": "user", "content": "Hi !"},

{"role": "assistant", "content": "Hi human, what can help you with ?"},

]

To convert the previous conversation into a prompt, we load the tokenizer and call apply_chat_template:

Copied

from transformers *import* AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("HuggingFaceTB/SmolLM2-1.7B-Instruct")

rendered_prompt = tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True)

The rendered_prompt returned by this function is now ready to use as the input for the model you chose!

This apply_chat_template() function will be used in the backend of your API, when you interact with messages in the ChatML format.

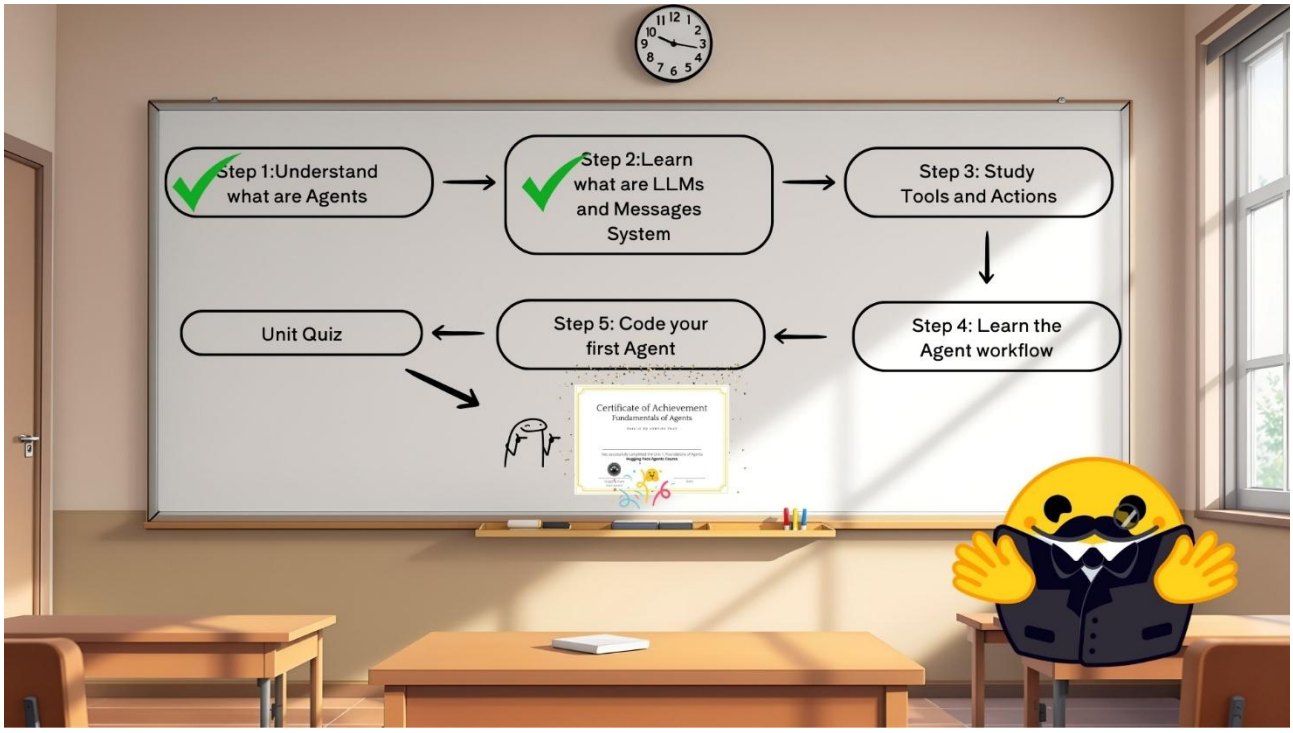
Now that we've seen how LLMs structure their inputs via chat templates, let's explore how Agents act in their environments.

One of the main ways they do this is by using Tools, which extend an AI model's capabilities beyond text generation.

We'll discuss messages again in upcoming units, but if you want a deeper dive now, check out:

- Hugging Face Chat Templating Guide
- Transformers Documentation

What are Tools?



One crucial aspect of AI Agents is their ability to take **actions**. As we saw, this happens through the use of **Tools**.

In this section, we'll learn what Tools are, how to design them effectively, and how to integrate them into your Agent via the System Message.

By giving your Agent the right Tools—and clearly describing how those Tools work—you can dramatically increase what your AI can accomplish. Let's dive in!

What are AI Tools?

A **Tool is a function given to the LLM**. This function should fulfill a **clear objective**.

Here are some commonly used tools in AI agents:

Tool	Description
Web Search	Allows the agent to fetch up-to-date information from the internet.
Image Generation	Creates images based on text descriptions.
Retrieval	Retrieves information from an external source.
API Interface	Interacts with an external API (GitHub, YouTube, Spotify, etc.).

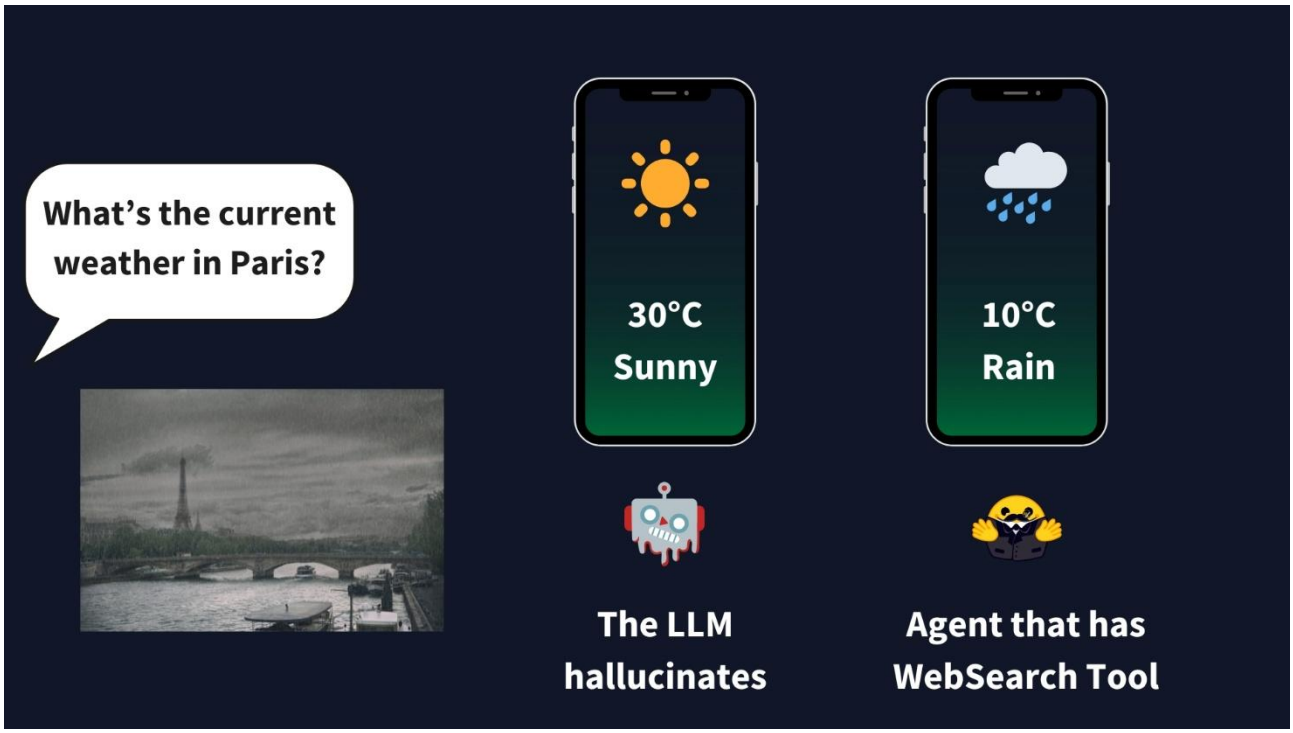
Those are only examples, as you can in fact create a tool for any use case!

A good tool should be something that **complements the power of an LLM**.

For instance, if you need to perform arithmetic, giving a **calculator tool** to your LLM will provide better results than relying on the native capabilities of the model.

Furthermore, **LLMs predict the completion of a prompt based on their training data**, which means that their internal knowledge only includes events prior to their training. Therefore, if your agent needs up-to-date data you must provide it through some tool.

For instance, if you ask an LLM directly (without a search tool) for today's weather, the LLM will potentially hallucinate random weather.



- A Tool should contain:
    - A **textual description of what the function does**.
    - A *Callable* (something to perform an action).
    - *Arguments* with typings.
    - (Optional) Outputs with typings.

How do tools work?

LLMs, as we saw, can only receive text inputs and generate text outputs. They have no way to call tools on their own. When we talk about providing tools to an Agent, we mean teaching the

LLM about the existence of these tools and instructing it to generate text-based invocations when needed.

For example, if we provide a tool to check the weather at a location from the internet and then ask the LLM about the weather in Paris, the LLM will recognize that this is an opportunity to use the “weather” tool. Instead of retrieving the weather data itself, the LLM will generate text that represents a tool call, such as call weather_tool(‘Paris’).

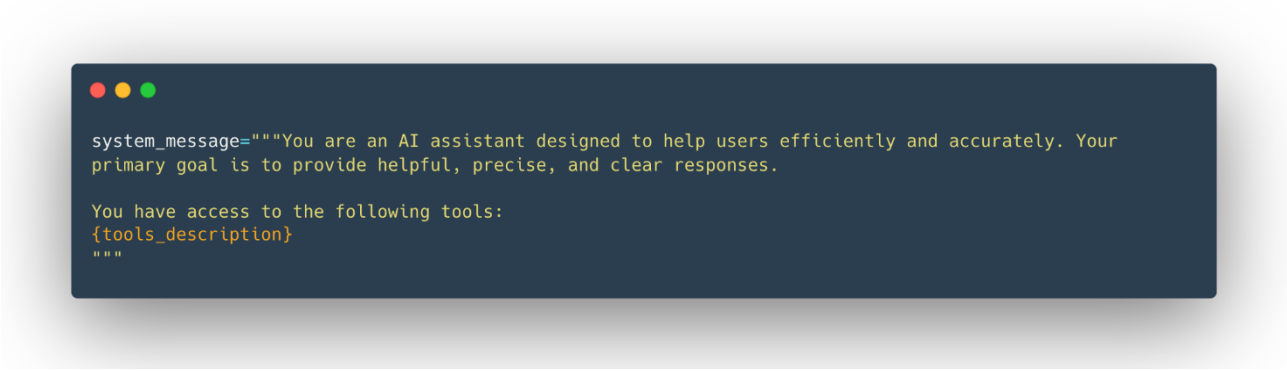
The **Agent** then reads this response, identifies that a tool call is required, executes the tool on the LLM’s behalf, and retrieves the actual weather data.

The Tool-calling steps are typically not shown to the user: the Agent appends them as a new message before passing the updated conversation to the LLM again. The LLM then processes this additional context and generates a natural-sounding response for the user. From the user’s perspective, it appears as if the LLM directly interacted with the tool, but in reality, it was the Agent that handled the entire execution process in the background.

We’ll talk a lot more about this process in future sessions.

How do we give tools to an LLM?

The complete answer may seem overwhelming, but we essentially use the system prompt to provide textual descriptions of available tools to the model:



For this to work, we have to be very precise and accurate about:

1. **What the tool does**
2. **What exact inputs it expects**

This is the reason why tool descriptions are usually provided using expressive but precise structures, such as computer languages or JSON. It’s not *necessary* to do it like that, any precise and coherent format would work.

If this seems too theoretical, let’s understand it through a concrete example.

We will implement a simplified **calculator** tool that will just multiply two integers. This could be our Python implementation:

Copied

def calculator(a: *int*, b: *int*) -> *int*:

"""Multiply two integers."""

return a * b

So our tool is called calculator, it **multiplies two integers**, and it requires the following inputs:

- a (*int*): An integer.
- b (*int*): An integer.

The output of the tool is another integer number that we can describe like this:

- (*int*): The product of a and b.

All of these details are important. Let's put them together in a text string that describes our tool for the LLM to understand.

Tool Name: calculator, Description: Multiply two integers., Arguments: a: int, b: int, Outputs: int

Reminder: This textual description is what we want the LLM to know about the tool.

When we pass the previous string as part of the input to the LLM, the model will recognize it as a tool, and will know what it needs to pass as inputs and what to expect from the output.

If we want to provide additional tools, we must be consistent and always use the same format. This process can be **fragile**, and we might **accidentally overlook** some details.

Is there a better way?

Auto-formatting Tool sections

Our tool was written in Python, and the implementation already provides everything we need:

- A descriptive name of what it does: calculator
- A longer description, provided by the function's docstring comment: Multiply two integers.
- The inputs and their type: the function clearly expects two ints.
- The type of the output.

There's a reason people use programming languages: they are expressive, concise, and precise.

We could provide the Python source code as the *specification* of the tool for the LLM, but the way the tool is implemented does not matter. All that matters is its name, what it does, the inputs it expects and the output it provides.

We will leverage Python's introspection features to leverage the source code and build a tool description automatically for us. All we need is that the tool implementation uses type hints, docstrings, and sensible function names. We will write some code to extract the relevant portions from the source code.

After we are done, we'll only need to use a Python decorator to indicate that the calculator function is a tool:

Copied

```
def calculator(a: int, b: int) -> int:
    """Multiply two integers."""
    return a * b

print(calculator.to_string())
```

Note the @tool decorator before the function definition.

With the implementation we'll see next, we will be able to retrieve the following text automatically from the source code via the to_string() function provided by the decorator:

Copied

Tool Name: calculator, Description: Multiply two integers., Arguments: a: int, b: int, Outputs: int

As you can see, it's the same thing we wrote manually before!

Generic Tool implementation

We create a generic Tool class that we can reuse whenever we need to use a tool.

Disclaimer: *This example implementation is fictional but closely resembles real implementations in most libraries.*

```python
from typing import Callable


class Tool:
    """
    A class representing a reusable piece of code (Tool).

    Attributes:
        name (str): Name of the tool.
        description (str): A textual description of what the tool does.
        func (callable): The function this tool wraps.
        arguments (list): A list of arguments.
        outputs (str or list): The return type(s) of the wrapped function.
    """
    def __init__(self,
                 name: str,
                 description: str,
                 func: Callable,
                 arguments: list,
                 outputs: str):
        self.name = name
        self.description = description
        self.func = func
        self.arguments = arguments
        self.outputs = outputs

    def to_string(self) -> str:
        """
        Return a string representation of the tool,
        including its name, description, arguments, and outputs.
        """
        args_str = ", ".join([
            f"{arg_name}: {arg_type}" for arg_name, arg_type in self.arguments
        ])

        return (
            f"Tool Name: {self.name},"
            f" Description: {self.description},"
            f" Arguments: {args_str},"
            f" Outputs: {self.outputs}"
        )

    def __call__(self, *args, **kwargs):
        """
        Invoke the underlying function (callable) with provided arguments.
        """
        return self.func(*args, **kwargs)
```

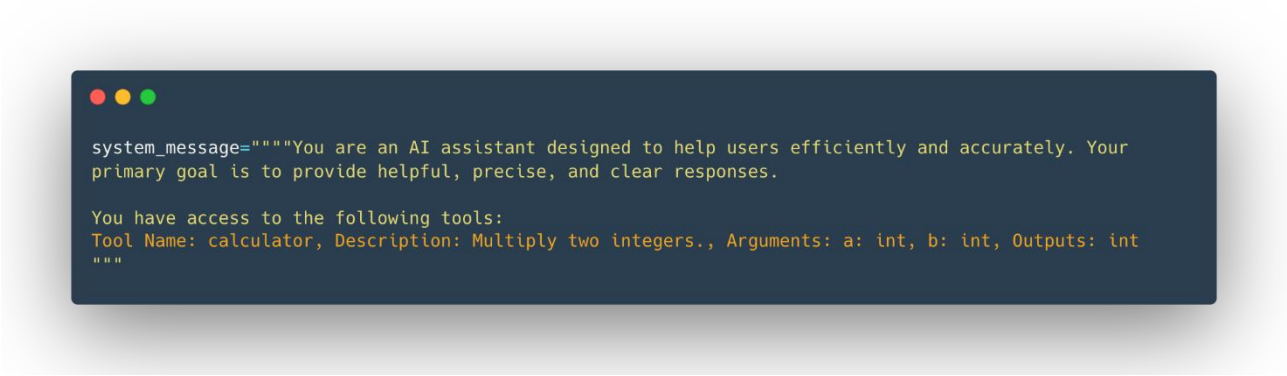
It may seem complicated, but if we go slowly through it we can see what it does. We define a Tool class that includes:

- *name (str): The name of the tool.*

- *description (str): A brief description of what the tool does.*
- *function (callable): The function the tool executes.*
- *arguments (list): The expected input parameters.*
- *outputs (str or list): The expected outputs of the tool.*
- *__call__(): Calls the function when the tool instance is invoked.*
- *to_string(): Converts the tool's attributes into a textual representation.*

We could create a Tool with this class using code like the following:

*The description is **injected** in the system prompt. Taking the example with which we started this section, here is how it would look like after replacing the tools_description:*



```
system_message="""You are an AI assistant designed to help users efficiently and accurately. Your
primary goal is to provide helpful, precise, and clear responses.

You have access to the following tools:
Tool Name: calculator, Description: Multiply two integers., Arguments: a: int, b: int, Outputs: int
"""
```

In the Actions section, we will learn more about how an Agent can **Call** this tool we just created.

Model Context Protocol (MCP): a unified tool interface

Model Context Protocol (MCP) is an **open protocol** that standardizes how applications **provide tools to LLMs**. MCP provides:

- A growing list of pre-built integrations that your LLM can directly plug into
- The flexibility to switch between LLM providers and vendors
- Best practices for securing your data within your infrastructure

This means that **any framework implementing MCP can leverage tools defined within the protocol**, eliminating the need to reimplement the same tool interface for each framework.

If you want to dive deeper about MCP, you can check our free MCP Course.

What is a REST API?

REST (Representational State Transfer) is an architectural style for designing networked applications. A REST API is an interface that allows two software systems to communicate over the HTTP protocol, using a client-server model.

Core Characteristics:

- **Statelessness:** Each request from a client to a server must contain all the information needed to understand and complete the request. The server does not store any "session" data about the client.
- **Resources:** In REST, everything is a "resource" (like a user, a paper submission, or a weather report), identified by a unique **URL** (Endpoint).
- **Standard Methods:** It uses standard HTTP verbs to perform actions:
    - **GET**: Retrieve data.
    - **POST**: Create a new resource.
    - **PUT/PATCH**: Update existing data.
    - **DELETE**: Remove a resource.
- **Data Format:** While it can support various formats, it almost exclusively uses **JSON** (JavaScript Object Notation) because it is lightweight and easy for both humans and machines to read.

In simple terms: A REST API is like a **menu** in a restaurant. The client (customer) looks at the menu (API documentation), places an order (HTTP Request), and the kitchen (Server) delivers the food (Response) back to the table.

No, MCP does not replace REST APIs.

In fact, **they are complementary**. Think of REST APIs as the **engine** and MCP as the **steering wheel** designed specifically for AI.

The Fundamental Difference

Feature	REST API	MCP (Model Context Protocol)
Purpose	Data exchange between software.	Standardizing how AI interacts with data/tools.
Primary User	Human developers / Applications.	AI Agents / LLMs.
Interface	Endpoints (GET, POST, etc.).	Resources, Prompts, and Tools.
Format	JSON/XML (Often requires manual parsing).	Structured Schema (Auto-discoverable by the AI).

Esporta in Fogli

How They Work Together

In a modern AI agent architecture, you don't choose between them; you layer them. The **MCP Server** acts as a "wrapper" or a "translator" for your existing REST APIs.

1. **The Agent (LLM):** Says, *"I need to fetch the latest paper submissions from EDAS."*
2. **The MCP Server:** Provides a **Tool** called get_submissions.
3. **The REST API (Back-end):** The MCP Tool executes a **REST GET request** to the EDAS servers to retrieve the raw data.
4. **The Response:** The REST API returns a raw JSON; the MCP Server cleans it up and hands it back to the Agent in a way it understands perfectly.

Key Takeaway: REST APIs provide the **access**, while MCP provides the **context** and **standardization** so you don't have to write custom "glue code" for every new AI agent you build.

Why use MCP if I already have REST?

If you only use REST, you have to manually teach your Agent (via long system prompts) how to call every single endpoint, what headers to use, and how to handle errors.

With **MCP**, the Agent can "inspect" the server and say: *"Oh, I see you have these 5 tools available, I know exactly how to use them."* It makes your system **pluggable** and **scalable**.

Tools play a crucial role in enhancing the capabilities of AI agents.

To summarize, we learned:

- *What Tools Are*: Functions that give LLMs extra capabilities, such as performing calculations or accessing external data.
- *How to Define a Tool*: By providing a clear textual description, inputs, outputs, and a callable function.
- *Why Tools Are Essential*: They enable Agents to overcome the limitations of static model training, handle real-time tasks, and perform specialized actions.

Now, we can move on to the Agent Workflow where you'll see how an Agent observes, thinks, and acts. This **brings together everything we've covered so far** and sets the stage for creating your own fully functional AI Agent.

Understanding AI Agents through the Thought-Action-Observation Cycle

In the previous sections, we learned:

- **How tools are made available to the agent in the system prompt**.
- **How AI agents are systems that can 'reason', plan, and interact with their environment**.

In this section, **we'll explore the complete AI Agent Workflow**, a cycle we defined as Thought-Action-Observation.

And then, we'll dive deeper into each of these steps.

The Core Components

Agents' work is a continuous cycle of: **thinking (Thought) → acting (Act) and observing (Observe)**.

Let's break down these actions together:

1. **Thought**: The LLM part of the Agent decides what the next step should be.
2. **Action:** The agent takes an action by calling the tools with the associated arguments.
3. **Observation:** The model reflects on the response from the tool.

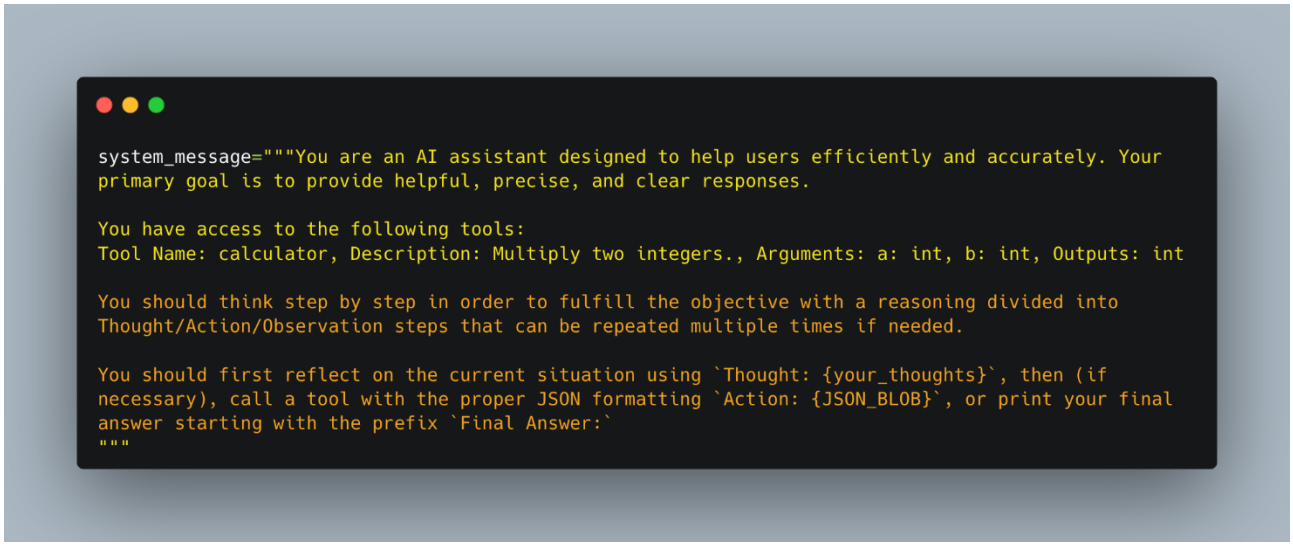
The Thought-Action-Observation Cycle

The three components work together in a continuous loop. To use an analogy from programming, the agent uses a **while loop**: the loop continues until the objective of the agent has been fulfilled.

Visually, it looks like this:

In many Agent frameworks, **the rules and guidelines are embedded directly into the system prompt**, ensuring that every cycle adheres to a defined logic.

In a simplified version, our system prompt may look like this:



```
system_message="""You are an AI assistant designed to help users efficiently and accurately. Your
primary goal is to provide helpful, precise, and clear responses.

You have access to the following tools:
Tool Name: calculator, Description: Multiply two integers., Arguments: a: int, b: int, Outputs: int

You should think step by step in order to fulfill the objective with a reasoning divided into
Thought/Action/Observation steps that can be repeated multiple times if needed.

You should first reflect on the current situation using `Thought: {your_thoughts}`, then (if
necessary), call a tool with the proper JSON formatting `Action: {JSON_BLOB}`, or print your final
answer starting with the prefix `Final Answer:`
"""
```

We see here that in the System Message we defined :

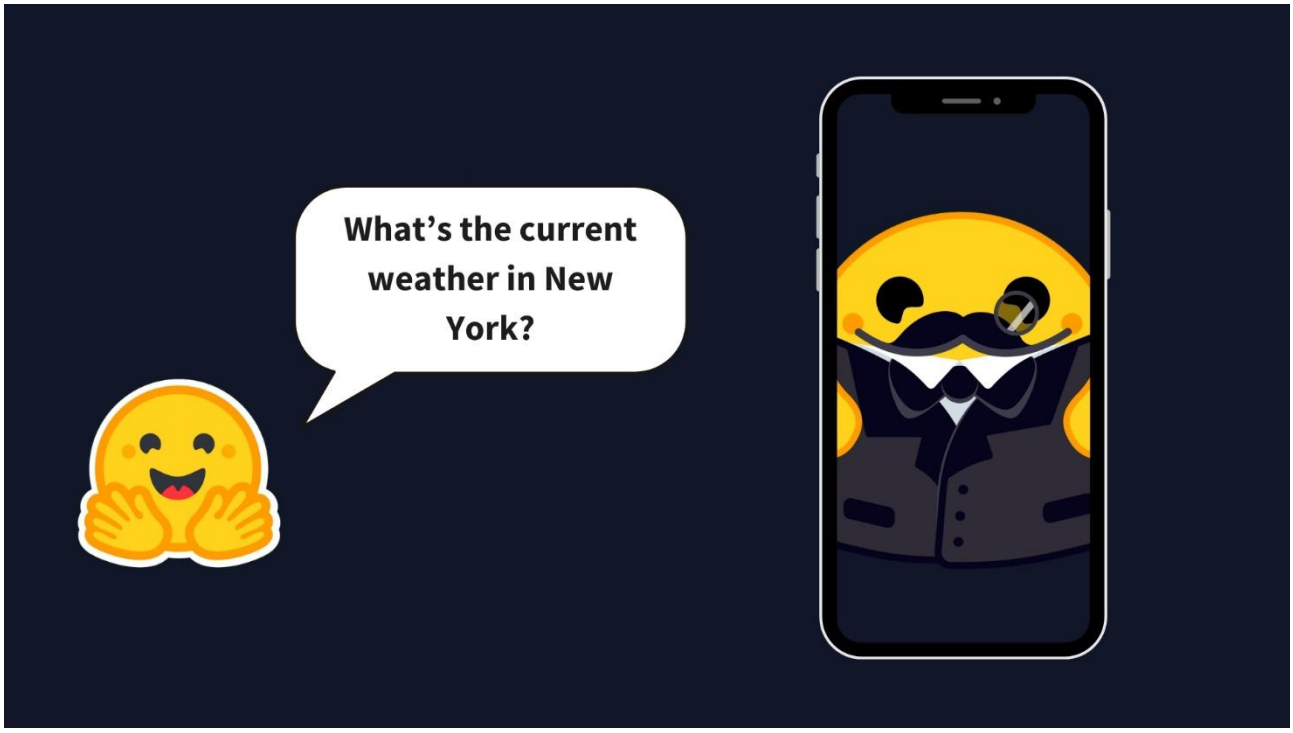
- The *Agent's behavior*.
- The *Tools our Agent has access to*, as we described in the previous section.
- The *Thought-Action-Observation Cycle*, that we bake into the LLM instructions.

Let’s take a small example to understand the process before going deeper into each step of the process.

Alfred, the weather Agent

We created Alfred, the Weather Agent.

A user asks Alfred: “What’s the current weather in New York?”



Thought

Internal Reasoning:

Upon receiving the query, Alfred’s internal dialogue might be:

“The user needs current weather information for New York. I have access to a tool that fetches weather data. First, I need to call the weather API to get up-to-date details.”

This step shows the agent breaking the problem into steps: first, gathering the necessary data.

Action

Tool Usage:

Based on its reasoning and the fact that Alfred knows about a get_weather tool, Alfred prepares a JSON-formatted command that calls the weather API tool. For example, its first action could be:

Thought: I need to check the current weather for New York.

```
{
    "action": "get_weather",
    "action_input": {
      "location": "New York"
    }
  }
```

Here, the action clearly specifies which tool to call (e.g., get_weather) and what parameter to pass (the “location”: “New York”).

Observation

Feedback from the Environment:

After the tool call, Alfred receives an observation. This might be the raw weather data from the API such as:

“Current weather in New York: partly cloudy, 15°C, 60% humidity.”

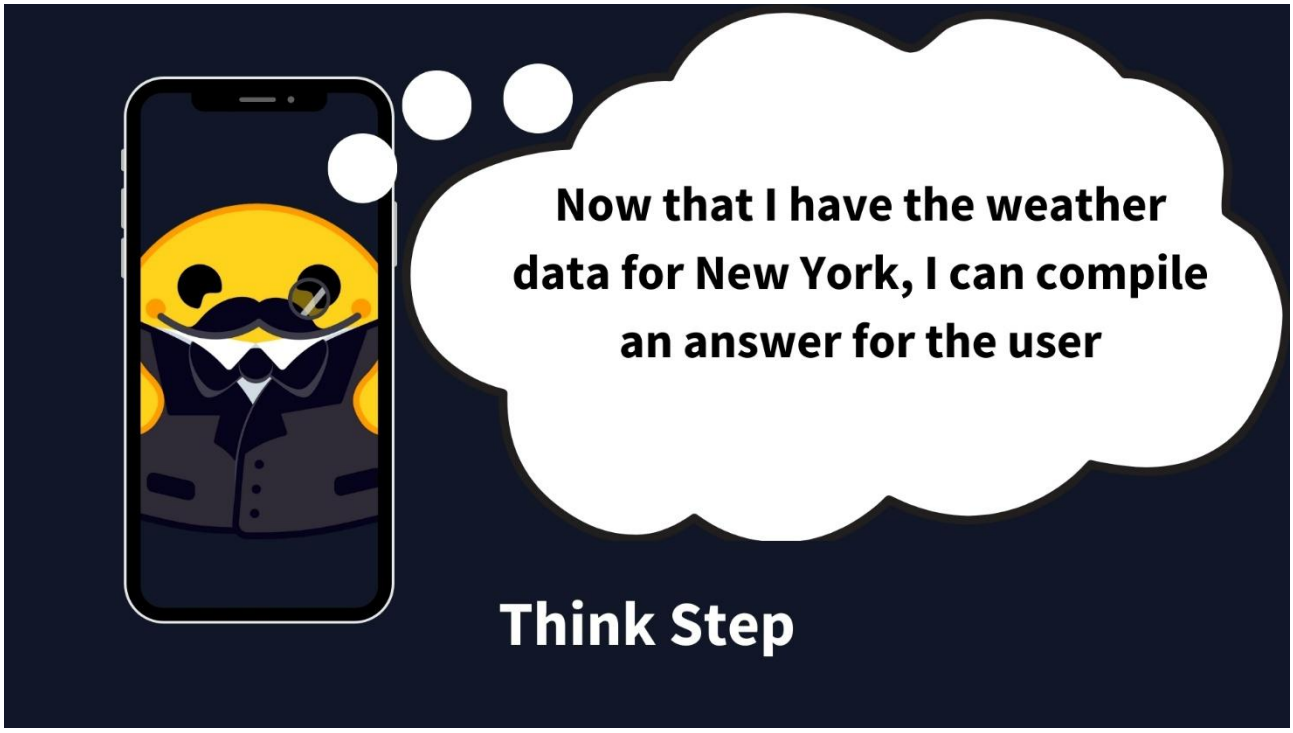
This observation is then added to the prompt as additional context. It functions as real-world feedback, confirming whether the action succeeded and providing the needed details.

Updated thought

Reflecting:

With the observation in hand, Alfred updates its internal reasoning:

“Now that I have the weather data for New York, I can compile an answer for the user.”



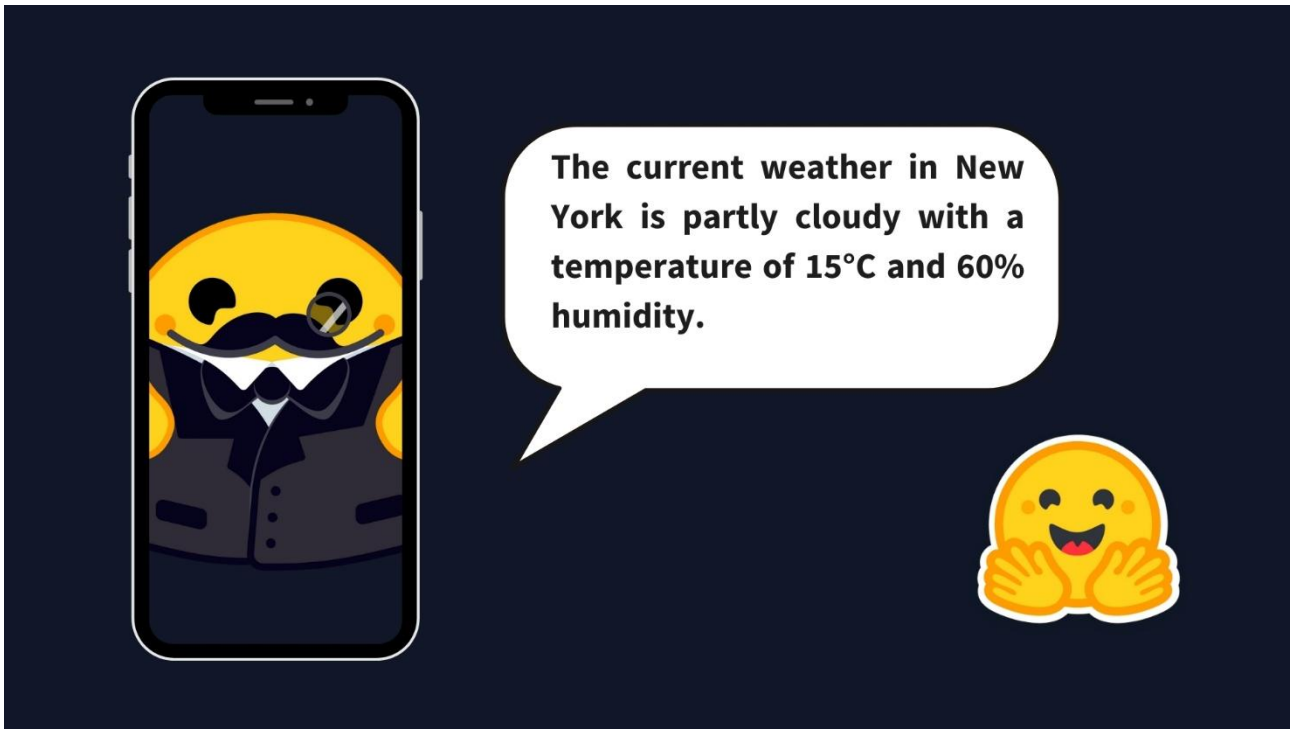
Final Action

Alfred then generates a final response formatted as we told it to:

Thought: I have the weather data now. The current weather in New York is partly cloudy with a temperature of 15°C and 60% humidity.”

Final answer : The current weather in New York is partly cloudy with a temperature of 15°C and 60% humidity.

This final action sends the answer back to the user, closing the loop.



What we see in this example:

- **Agents iterate through a loop until the objective is fulfilled:**

Alfred's process is cyclical. It starts with a thought, then acts by calling a tool, and finally observes the outcome. If the observation had indicated an error or incomplete data, Alfred could have re-entered the cycle to correct its approach.

- **Tool Integration:**

The ability to call a tool (like a weather API) enables Alfred to go **beyond static knowledge and retrieve real-time data**, an essential aspect of many AI Agents.

- **Dynamic Adaptation:**

Each cycle allows the agent to incorporate fresh information (observations) into its reasoning (thought), ensuring that the final answer is well-informed and accurate.

This example showcases the core concept behind the *ReAct cycle* (a concept we're going to develop in the next section): **the interplay of Thought, Action, and Observation empowers AI agents to solve complex tasks iteratively**.

By understanding and applying these principles, you can design agents that not only reason about their tasks but also **effectively utilize external tools to complete them**, all while continuously refining their output based on environmental feedback.

What we see in this example:

- **Agents iterate through a loop until the objective is fulfilled:**

Alfred's process is cyclical. It starts with a thought, then acts by calling a tool, and finally observes the outcome. If the observation had indicated an error or incomplete data, Alfred could have re-entered the cycle to correct its approach.

- **Tool Integration:**

The ability to call a tool (like a weather API) enables Alfred to go **beyond static knowledge and retrieve real-time data**, an essential aspect of many AI Agents.

- **Dynamic Adaptation:**

Each cycle allows the agent to incorporate fresh information (observations) into its reasoning (thought), ensuring that the final answer is well-informed and accurate.

This example showcases the core concept behind the *ReAct cycle* (a concept we're going to develop in the next section): **the interplay of Thought, Action, and Observation empowers AI agents to solve complex tasks iteratively**.

By understanding and applying these principles, you can design agents that not only reason about their tasks but also **effectively utilize external tools to complete them**, all while continuously refining their output based on environmental feedback.

Thought: Internal Reasoning and the ReAct Approach

In this section, we dive into the inner workings of an AI agent—its ability to reason and plan. We'll explore how the agent leverages its internal dialogue to analyze information, break down complex problems into manageable steps, and decide what action to take next.

Additionally, we introduce the ReAct approach, a prompting technique that encourages the model to think "step by step" before acting.

Thoughts represent the **Agent's internal reasoning and planning processes** to solve the task.

This utilises the agent's Large Language Model (LLM) capacity **to analyze information when presented in its prompt** — essentially, its inner monologue as it works through a problem.

The Agent's thoughts help it assess current observations and decide what the next action(s) should be. Through this process, the agent can **break down complex problems into smaller, more manageable steps**, reflect on past experiences, and continuously adjust its plans based on new information.

🧠 **Examples of Common Thought Types**

Type of Thought	Example
Planning	"I need to break this task into three steps: 1) gather data, 2) analyze trends, 3) generate report"
Analysis	"Based on the error message, the issue appears to be with the database connection parameters"
Decision Making	"Given the user's budget constraints, I should recommend the mid-tier option"
Problem Solving	"To optimize this code, I should first profile it to identify bottlenecks"
Memory Integration	"The user mentioned their preference for Python earlier, so I'll provide examples in Python"
Self-Reflection	"My last approach didn't work well, I should try a different strategy"
Goal Setting	"To complete this task, I need to first establish the acceptance criteria"

Type of Thought	Example
Prioritization	“The security vulnerability should be addressed before adding new features”

Note: *In the case of LLMs fine-tuned for function-calling, the thought process is optional. More details will be covered in the Actions section.*

Chain-of-Thought (CoT)

Chain-of-Thought (CoT) is a prompting technique that guides a model to **think through a problem step-by-step before producing a final answer.**

It typically starts with:

“Let’s think step by step.”

This approach helps the model **reason internally**, especially for logical or mathematical tasks, **without interacting with external tools**.

✅ **Example (CoT)**

Question: What is 15% of 200?

Thought: Let's think step by step. 10% of 200 is 20, and 5% of 200 is 10, so 15% is 30.

Answer: 30

⚙️ **ReAct: Reasoning + Acting**

A key method is the **ReAct approach**, which combines “Reasoning” (Think) with “Acting” (Act).

ReAct is a prompting technique that encourages the model to think step-by-step and interleave actions (like using tools) between reasoning steps.

This enables the agent to solve complex multi-step tasks by alternating between:

- Thought: internal reasoning
- Action: tool usage
- Observation: receiving tool output

🔁 **Example (ReAct)**

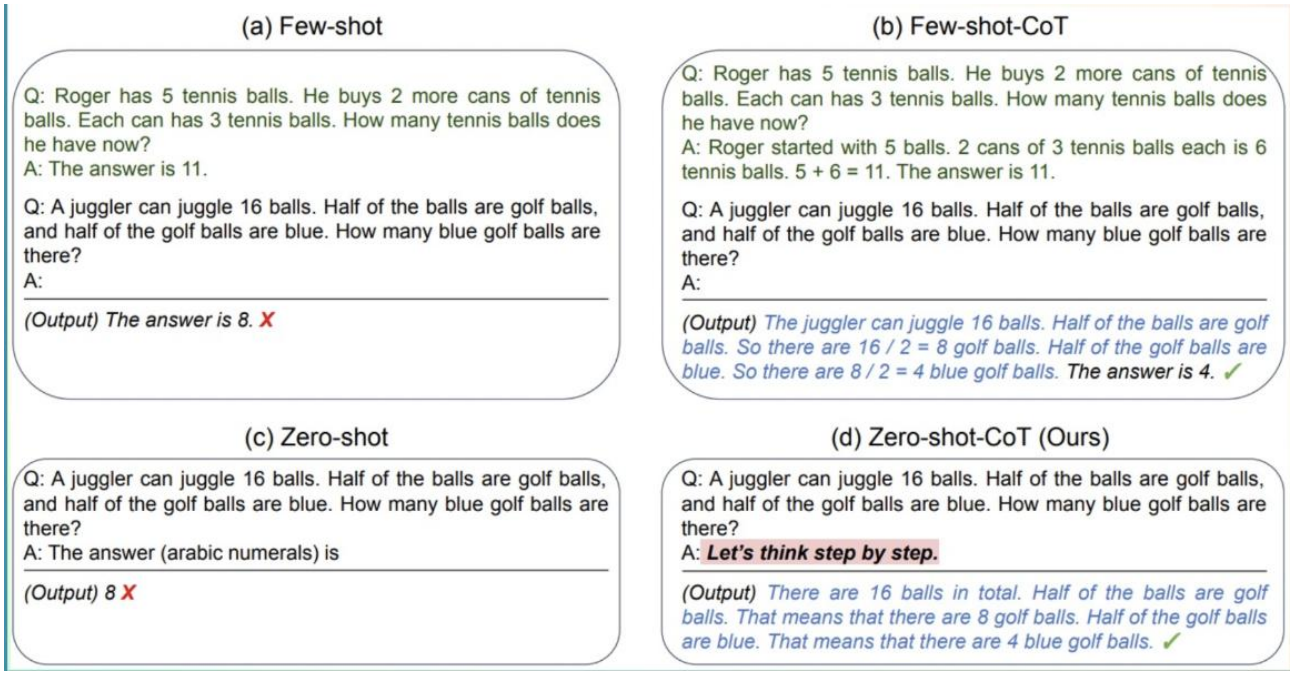
Thought: I need *to* find the latest weather *in* Paris.

Action: Search["weather in Paris"]

Observation: It*’s 18°C and cloudy.*

Thought: Now that I know the weather...

Action: Finish["It's 18°C and cloudy in Paris."]



🔁 Comparison: ReAct vs. CoT

Feature	Chain-of-Thought (CoT)	ReAct
Step-by-step logic	✅ Yes	✅ Yes
External tools	❌ No	✅ Yes (Actions + Observations)
Best suited for	Logic, math, internal tasks	Info-seeking, dynamic multi-step tasks

Recent models like **Deepseek R1** or **OpenAI's o1** were fine-tuned to *think before answering*. They use structured tokens like <think> and </think> to explicitly separate the reasoning phase from the final answer.

Unlike ReAct or CoT — which are prompting strategies — this is a **training-level technique**, where the model learns to think via examples.

Actions: Enabling the Agent to Engage with Its Environment

In this section, we explore the concrete steps an AI agent takes to interact with its environment.

We'll cover how actions are represented (using JSON or code), the importance of the stop and parse approach, and introduce different types of agents.

Actions are the concrete steps an **AI agent takes to interact with its environment**.

Whether it's browsing the web for information or controlling a physical device, each action is a deliberate operation executed by the agent.

For example, an agent assisting with customer service might retrieve customer data, offer support articles, or transfer issues to a human representative.

Types of Agent Actions

There are multiple types of Agents that take actions differently:

Type of Agent	Description
JSON Agent	The Action to take is specified in JSON format.
Code Agent	The Agent writes a code block that is interpreted externally.
Function-calling Agent	It is a subcategory of the JSON Agent which has been fine-tuned to generate a new message for each action.

Actions themselves can serve many purposes:

Actions themselves can serve many purposes:

Type of Action	Description
Information Gathering	Performing web searches, querying databases, or retrieving documents.
Tool Usage	Making API calls, running calculations, and executing code.
Environment Interaction	Manipulating digital interfaces or controlling physical devices.

Type of Action	Description
Communication	Engaging with users via chat or collaborating with other agents.

The LLM only handles text and uses it to describe the action it wants to take and the parameters to supply to the tool. For an agent to work properly, the LLM must STOP generating new tokens after emitting all the tokens to define a complete Action. This passes control from the LLM back to the agent and ensures the result is parseable - whether the intended format is JSON, code, or function-calling.

The Stop and Parse Approach

One key method for implementing actions is the **stop and parse approach**. This method ensures that the agent's output is structured and predictable:

1. **Generation in a Structured Format**:

The agent outputs its intended action in a clear, predetermined format (JSON or code).

2. **Halting Further Generation**:

Once the text defining the action has been emitted, **the LLM stops generating additional tokens**. This prevents extra or erroneous output.

3. **Parsing the Output**:

An external parser reads the formatted action, determines which Tool to call, and extracts the required parameters.

For example, an agent needing to check the weather might output:

Copied

Thought: I need to check the current weather for New York.

Action :

```
{
  "action": "get_weather",
  "action_input": {"location": "New York"}
}
```

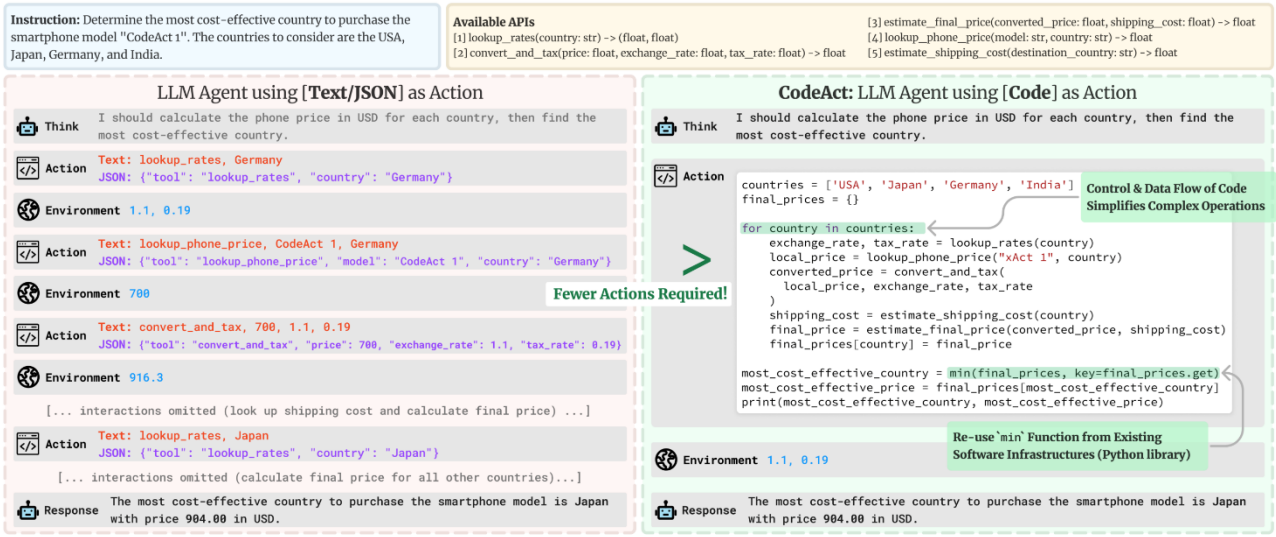
The framework can then easily parse the name of the function to call and the arguments to apply.

This clear, machine-readable format minimizes errors and enables external tools to accurately process the agent's command.

Note: Function-calling agents operate similarly by structuring each action so that a designated function is invoked with the correct arguments. We'll dive deeper into those types of Agents in a future Unit.

Code Agents

An alternative approach is using *Code Agents*. The idea is: **instead of outputting a simple JSON object**, a Code Agent generates an **executable code block—typically in a high-level language like Python**.



This approach offers several advantages:

- **Expressiveness:** Code can naturally represent complex logic, including loops, conditionals, and nested functions, providing greater flexibility than JSON.
- **Modularity and Reusability:** Generated code can include functions and modules that are reusable across different actions or tasks.
- **Enhanced Debuggability:** With a well-defined programming syntax, code errors are often easier to detect and correct.
- **Direct Integration:** Code Agents can integrate directly with external libraries and APIs, enabling more complex operations such as data processing or real-time decision making.

You must keep in mind that executing LLM-generated code may pose security risks, from prompt injection to the execution of harmful code. That's why it's recommended to use AI agent frameworks like smolagents that integrate default safeguards. If you want to know more about the risks and how to mitigate them, please have a look at this dedicated section.

For example, a Code Agent tasked with fetching the weather might generate the following Python snippet:

```python
# Code Agent Example: Retrieve Weather Information
def get_weather(city):
    import requests
    api_url = f"https://api.weather.com/v1/location/{city}?apiKey=YOUR_API_KEY"
    response = requests.get(api_url)
    if response.status_code == 200:
        data = response.json()
        return data.get("weather", "No weather information available")
    else:
        return "Error: Unable to fetch weather data."

# Execute the function and prepare the final answer
result = get_weather("New York")
final_answer = f"The current weather in New York is: {result}"
print(final_answer)
```

In this example, the Code Agent:

- Retrieves weather data **via an API call**,
- Processes the response,
- And uses the print() function to output a final answer.

This method **also follows the stop and parse approach** by clearly delimiting the code block and signaling when execution is complete (here, by printing the final_answer).

We learned that Actions bridge an agent's internal reasoning and its real-world interactions by executing clear, structured tasks—whether through JSON, code, or function calls.

This deliberate execution ensures that each action is precise and ready for external processing via the stop and parse approach. In the next section, we will explore Observations to see how agents capture and integrate feedback from their environment.

After this, we will **finally be ready to build our first Agent!**

Observe: Integrating Feedback to Reflect and Adapt

Observations are **how an Agent perceives the consequences of its actions**.

They provide crucial information that fuels the Agent's thought process and guides future actions.

They are **signals from the environment**—whether it's data from an API, error messages, or system logs—that guide the next cycle of thought.

In the observation phase, the agent:

- **Collects Feedback:** Receives data or confirmation that its action was successful (or not).
- **Appends Results:** Integrates the new information into its existing context, effectively updating its memory.
- **Adapts its Strategy:** Uses this updated context to refine subsequent thoughts and actions.

For example, if a weather API returns the data *"partly cloudy, 15°C, 60% humidity"*, this observation is appended to the agent's memory (at the end of the prompt).

The Agent then uses it to decide whether additional information is needed or if it's ready to provide a final answer.

This **iterative incorporation of feedback ensures the agent remains dynamically aligned with its goals**, constantly learning and adjusting based on real-world outcomes.

These observations **can take many forms**, from reading webpage text to monitoring a robot arm's position. This can be seen like Tool "logs" that provide textual feedback of the Action execution.

Type of Observation	Example
System Feedback	Error messages, success notifications, status codes
Data Changes	Database updates, file system modifications, state changes
Environmental Data	Sensor readings, system metrics, resource usage
Response Analysis	API responses, query results, computation outputs
Time-based Events	Deadlines reached, scheduled tasks completed

How Are the Results Appended?

After performing an action, the framework follows these steps in order:

1. **Parse the action** to identify the function(s) to call and the argument(s) to use.
2. **Execute the action.**
3. **Append the result** as an **Observation**.

We've now learned the Agent's Thought-Action-Observation Cycle.

If some aspects still seem a bit blurry, don't worry—we'll revisit and deepen these concepts in future Units.

Now, it's time to put your knowledge into practice by coding your very first Agent!

Dummy Agent Library

This course is framework-agnostic because we want to **focus on the concepts of AI agents and avoid getting bogged down in the specifics of a particular framework**.

Also, we want students to be able to use the concepts they learn in this course in their own projects, using any framework they like.

Therefore, for this Unit 1, we will use a dummy agent library and a simple serverless API to access our LLM engine.

You probably wouldn't use these in production, but they will serve as a good **starting point for understanding how agents work**.

After this section, you'll be ready to **create a simple Agent** using smolagents

And in the following Units we will also use other AI Agent libraries like LangGraph, and LlamaIndex.

To keep things simple we will use a simple Python function as a Tool and Agent.

We will use built-in Python packages like datetime and os so that you can try it out in any environment.

You can follow the process in this notebook and **run the code yourself**.

Serverless API

In the Hugging Face ecosystem, there is a convenient feature called Serverless API that allows you to easily run inference on many models. There's no installation or deployment required.

```python
import os
from huggingface_hub import InferenceClient

## You need a token from https://hf.co/settings/tokens, ensure that you select 'read' as the token type.
# HF_TOKEN = os.environ.get("HF_TOKEN")

client = InferenceClient(model="moonshotai/Kimi-K2.5")
```

We use the chat method since it is a convenient and reliable way to apply chat templates:

```
output = client.chat.completions.create(
    messages=[
        {"role": "user", "content": "The capital of France is"},
    ],
    stream=False,
    max_tokens=1024,
    extra_body={'thinking': {'type': 'disabled'}},
)
print(output.choices[0].message.content)
```

output:

Paris.

The chat method is the RECOMMENDED method to use in order to ensure a smooth transition between models.

Dummy Agent

In the previous sections, we saw that the core of an agent library is to append information in the system prompt.

This system prompt is a bit more complex than the one we saw earlier, but it already contains:

1. **Information about the tools**
2. **Cycle instructions** (Thought → Action → Observation)

COMPLETARE

https://huggingface.co/learn/agents-course/unit1/tutorial

Let's Create Our First Agent Using smolagents

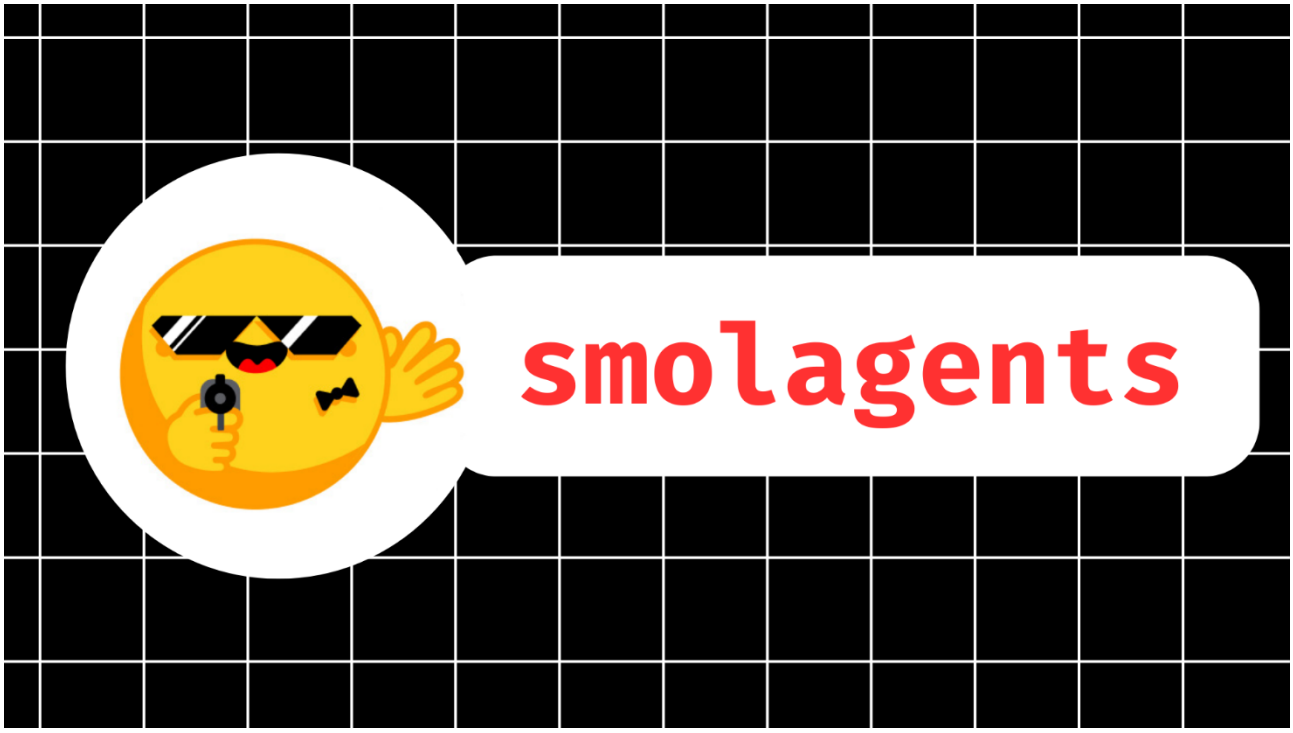
In the last section, we learned how we can create Agents from scratch using Python code, and we **saw just how tedious that process can be**. Fortunately, many Agent libraries simplify this work by **handling much of the heavy lifting for you**.

In this tutorial, **you'll create your very first Agent** capable of performing actions such as image generation, web search, time zone checking and much more!

You will also publish your agent **on a Hugging Face Space so you can share it with friends and colleagues**.

Let's get started!

What is smolagents?



To make this Agent, we're going to use smolagents, a library that **provides a framework for developing your agents with ease**.

This lightweight library is designed for simplicity, but it abstracts away much of the complexity of building an Agent, allowing you to focus on designing your agent's behavior.

We're going to get deeper into smolagents in the next Unit. Meanwhile, you can also check this blog post or the library's repo in GitHub.

In short, smolagents is a library that focuses on **codeAgent**, a kind of agent that performs **"Actions"** through code blocks, and then **"Observes"** results by executing the code.

Here is an example of what we'll build!

We provided our agent with an **Image generation tool** and asked it to generate an image of a cat.

The agent inside smolagents is going to have the **same behaviors as the custom one we built previously**: it's going **to think, act and observe in cycle** until it reaches a final answer:

https://www.youtube.com/watch?v=PQDKcWiuln4&t=29s

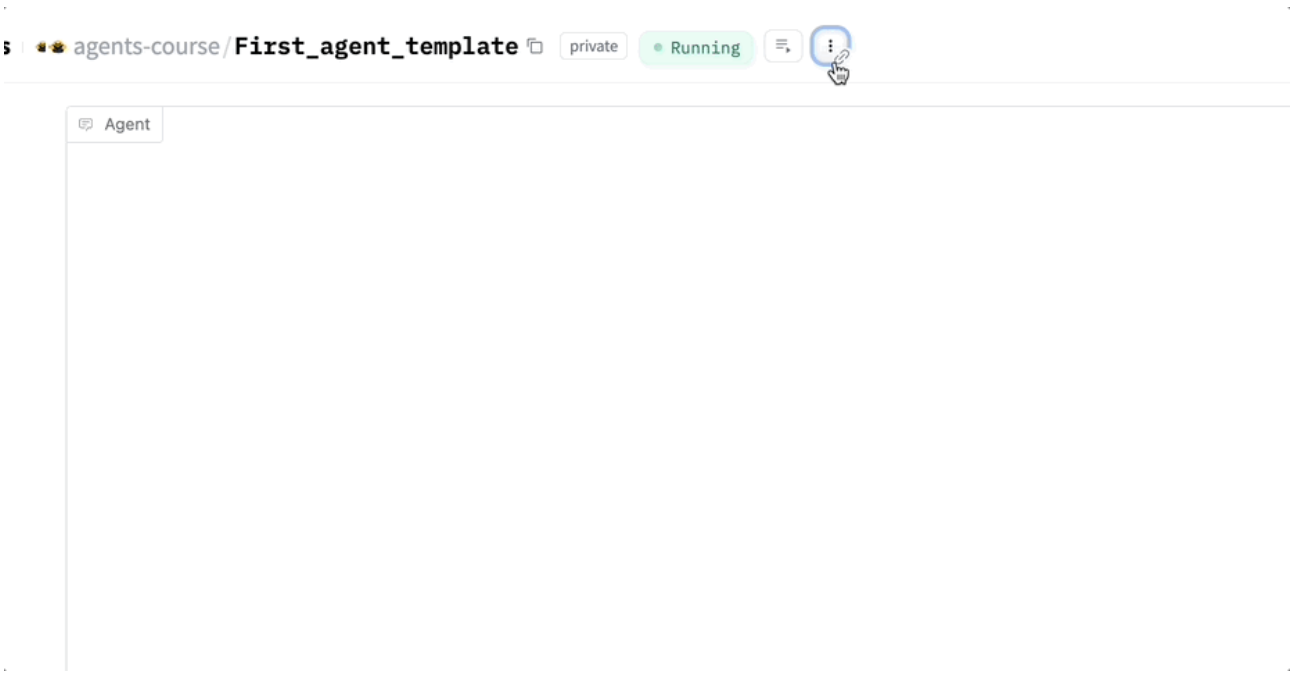
Exciting, right?

Let's build our Agent!

To start, duplicate this Space: https://huggingface.co/spaces/agents-course/First_agent_template

Thanks to Aymeric for this template! 🙌

Duplicating this space means **creating a local copy on your own profile**:



After duplicating the Space, you'll need to add your Hugging Face API token so your agent can access the model API:

1. First, get your Hugging Face token from https://hf.co/settings/tokens with permission for inference, if you don't already have one
2. Go to your duplicated Space and click on the **Settings** tab

3. Scroll down to the **Variables and Secrets** section and click **New Secret**
4. Create a secret with the name HF_TOKEN and paste your token as the value
5. Click **Save** to store your token securely

Throughout this lesson, the only file you will need to modify is the (currently incomplete) **“app.py”**. You can see here the original one in the template. To find yours, go to your copy of the space, then click the Files tab and then on app.py in the directory listing.

Let’s break down the code together:

- The file begins with some simple but necessary library imports

```
from smolagents import CodeAgent, DuckDuckGoSearchTool, FinalAnswerTool,
InferenceClientModel, load_tool, tool
import datetime
import requests
import pytz
import yaml
```

As outlined earlier, we will directly use the **CodeAgent** class from **smolagents**.

The Tools

Now let’s get into the tools! If you want a refresher about tools, don’t hesitate to go back to the Tools section of the course

```
@tool
def my_custom_tool(arg1:str, arg2:int)-> str: # it's important to specify
the return type
    # Keep this format for the tool description / args description but
feel free to modify the tool
    """A tool that does nothing yet
    Args:
        arg1: the first argument
        arg2: the second argument
    """
    return "What magic will you build ?"

@tool
def get_current_time_in_timezone(timezone: str) -> str:
    """A tool that fetches the current local time in a specified
timezone.
    Args:
        timezone: A string representing a valid timezone (e.g.,
'America/New_York').
    """
```

```
    try:
        # Create timezone object
        tz = pytz.timezone(timezone)
        # Get current time in that timezone
        local_time = datetime.datetime.now(tz).strftime("%Y-%m-%d
%H:%M:%S")
        return f"The current local time in {timezone} is: {local_time}"
    except Exception as e:
        return f"Error fetching time for timezone '{timezone}': {str(e)}"
```

The Tools are what we are encouraging you to build in this section! We give you two examples:

1. A **non-working dummy Tool** that you can modify to make something useful.
2. An **actually working Tool** that gets the current time somewhere in the world.

To define your tool it is important to:

1. Provide input and output types for your function, like in get_current_time_in_timezone(timezone: str) -> str:
2. **A well formatted docstring**. smolagents is expecting all the arguments to have a **textual description in the docstring**.

The Agent

It uses Qwen/Qwen2.5-Coder-32B-Instruct as the LLM engine. This is a very capable model that we'll access via the serverless API.

```
final_answer = FinalAnswerTool()
model = InferenceClientModel(
  max_tokens=2096,
  temperature=0.5,
  model_id='Qwen/Qwen2.5-Coder-32B-Instruct',
  custom_role_conversions=None,
)

with open("prompts.yaml", 'r') as stream:
  prompt_templates = yaml.safe_load(stream)

# We're creating our CodeAgent
agent = CodeAgent(
  model=model,
  tools=[final_answer], # add your tools here (don't remove final_answer)
  max_steps=6,
  verbosity_level=1,
  grammar=None,
```

```
    planning_interval=None,
    name=None,
    description=None,
    prompt_templates=prompt_templates
)

GradioUI(agent).launch()
```

This Agent still uses the InferenceClient we saw in an earlier section behind the **InferenceClientModel** class!

We will give more in-depth examples when we present the framework in Unit 2. For now, you need to focus on **adding new tools to the list of tools** using the tools parameter of your Agent.

For example, you could use the DuckDuckGoSearchTool that was imported in the first line of the code, or you can examine the image_generation_tool that is loaded from the Hub later in the code.

Adding tools will give your agent new capabilities, try to be creative here!

The System Prompt

The agent's system prompt is stored in a separate prompts.yaml file. This file contains predefined instructions that guide the agent's behavior.

Storing prompts in a YAML file allows for easy customization and reuse across different agents or use cases.

You can check the Space's file structure to see where the prompts.yaml file is located and how it's organized within the project.

The complete "app.py":

```
from smolagents import CodeAgent, DuckDuckGoSearchTool, InferenceClientModel,
load_tool, tool
import datetime
import requests
import pytz
import yaml
from tools.final_answer import FinalAnswerTool

from Gradio_UI import GradioUI

# Below is an example of a tool that does nothing. Amaze us with your creativity!
```

```
def my_custom_tool(arg1:str, arg2:int)-> str: # it's important to specify the return type
    # Keep this format for the tool description / args description but feel free to modify the tool
    """A tool that does nothing yet
    Args:
        arg1: the first argument
        arg2: the second argument
    """
    return "What magic will you build ?"


def get_current_time_in_timezone(timezone: str) -> str:
    """A tool that fetches the current local time in a specified timezone.
    Args:
        timezone: A string representing a valid timezone (e.g., 'America/New_York').
    """
    try:
        # Create timezone object
        tz = pytz.timezone(timezone)
        # Get current time in that timezone
        local_time = datetime.datetime.now(tz).strftime("%Y-%m-%d %H:%M:%S")
        return f"The current local time in {timezone} is: {local_time}"
    except Exception as e:
        return f"Error fetching time for timezone '{timezone}': {str(e)}"


final_answer = FinalAnswerTool()
model = InferenceClientModel(
    max_tokens=2096,
    temperature=0.5,
    model_id='Qwen/Qwen2.5-Coder-32B-Instruct',
    custom_role_conversions=None,
)


# Import tool from Hub
image_generation_tool = load_tool("agents-course/text-to-image", trust_remote_code=True)

# Load system prompt from prompt.yaml file
with open("prompts.yaml", 'r') as stream:
    prompt_templates = yaml.safe_load(stream)

agent = CodeAgent(
    model=model,
    tools=[final_answer], # add your tools here (don't remove final_answer)
    max_steps=6,
    verbosity_level=1,
```

```
    grammar=None,
    planning_interval=None,
    name=None,
    description=None,
    prompt_templates=prompt_templates # Pass system prompt to CodeAgent
)


GradioUI(agent).launch()
```

Your **Goal** is to get familiar with the Space and the Agent.

Currently, the agent in the template **does not use any tools, so try to provide it with some of the pre-made ones or even make some new tools yourself!**

We are eagerly waiting for your amazing agents output in the discord channel **#agents-course-showcase**!

Congratulations, you've built your first Agent! Don't hesitate to share it with your friends and colleagues.

Since this is your first try, it's perfectly normal if it's a little buggy or slow. In future units, we'll learn how to build even better Agents.

The best way to learn is to try, so don't hesitate to update it, add more tools, try with another model, etc.

In the next section, you're going to fill the final Quiz and get your certificate!

https://huggingface.co/learn/agents-course/unit1/final-quiz