# AI Agents Course by Hugging Face

## Syllabus

| Chapter | Topic | Description |
|---|---|---|
| 0 | Onboarding | Set you up with the tools and platforms that you will use. |
| 1 | Agent Fundamentals | Explain Tools, Thoughts, Actions, Observations, and their formats. Explain LLMs, messages, special tokens and chat templates. Show a simple use case using python functions as tools. |
| 2 | Frameworks | Understand how the fundamentals are implemented in popular libraries : smolagents, LangGraph, LLamaIndex |
| 3 | Use Cases | Let's build some real life use cases (open to PRs 🤗 from experienced Agent builders) |
| 4 | Final Assignment | Build an agent for a selected benchmark and prove your understanding of Agents on the student leaderboard 🚀 |

## What is an Agent?

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.

# 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.

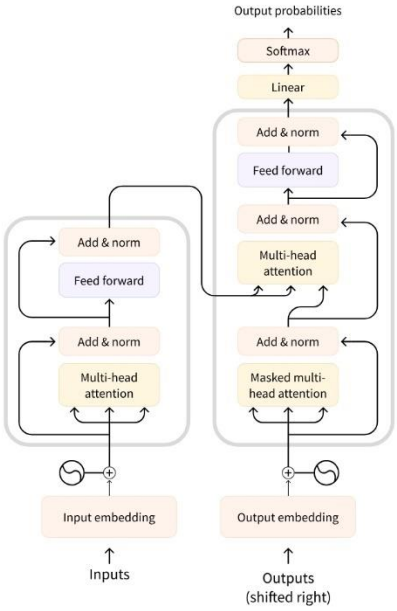





*Figure 1: Transformer Architecture*

## There are 3 types of transformers:

### 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

## 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

## 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*.

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

*Figure 2 : Some of the well-known LLMs*

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.

## Special Tokens

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.

| 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 |

*Figure 3 : Diversity of special tokens*

## 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.

*Figure 4 : Understanding next token prediction*

While the full process can be quite technical for the purpose of learning agents, here's a brief overview:

- Once the input text is **tokenized**, the model computes a representation of the sequence that captures information about the meaning and the position of each token in the input sequence.
- This representation goes into the model, which outputs scores that rank the likelihood of each token in its vocabulary as being the next one in the sequence.

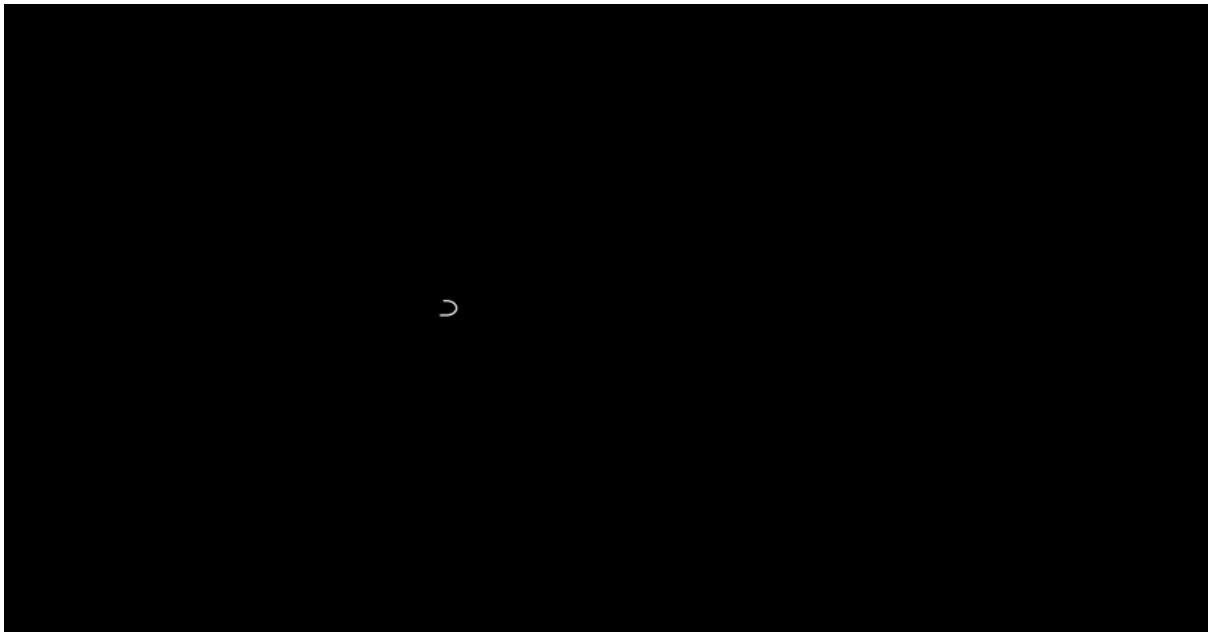


*Figure 5 : Tokenization of the prompt*

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.
- 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.

## Beam Search Visualizer

**Parameters:**

- **Sentence to decode from** (inputs): the input sequence to your decoder.
- **Number of steps** (max_new_tokens): the number of tokens to generate.
- **Number of beams** (num_beams): the number of beams to use.
- **Length penalty** (length_penalty): the length penalty to apply to outputs. length_penalty > 0.0 promotes longer sequences, while length_penalty < 0.0 encourages shorter sequences. This parameter will not impact the beam search paths, but only influence the choice of sequences in the end towards longer or shorter sequences.
- **Number of return sequences** (num_return_sequences): the number of sequences to be returned at the end of generation. Should be <= num_beams.

The conclusive sequences are the ones that end in an <|endoftext|> token or at the end of generation.

They are ranked by their scores, as given by the formula score = cumulative_score / (output_length ** length_penalty).

Only the top num_beams scoring sequences are returned: in the tree they are highlighted in **blue**. The non-selected sequences are also shown in the tree, highlighted in **yellow**.

**Output sequences:**

- Score -1.14: Conclusion: thanks a lot. That's all for today.\n\nAdvertisements<|endoftext|>
- Score -1.30: Conclusion: thanks a lot. That's all for today. I hope you enjoyed
- Score -1.33: Conclusion: thanks a lot. That's all for today. I'll be back

## 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.

*Figure 6 : Attention Value*

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

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).

## 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.

LLM is the brain of the Agent.

# Messages and Special Tokens

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

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**.


*Figure 7 : Breaking down of chat into a prompt*

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.

## 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:

```
system_message = {
    "role": "system",
```

*"content": "You are a professional customer service agent. Always be polite, clear, and helpful."*

*}*

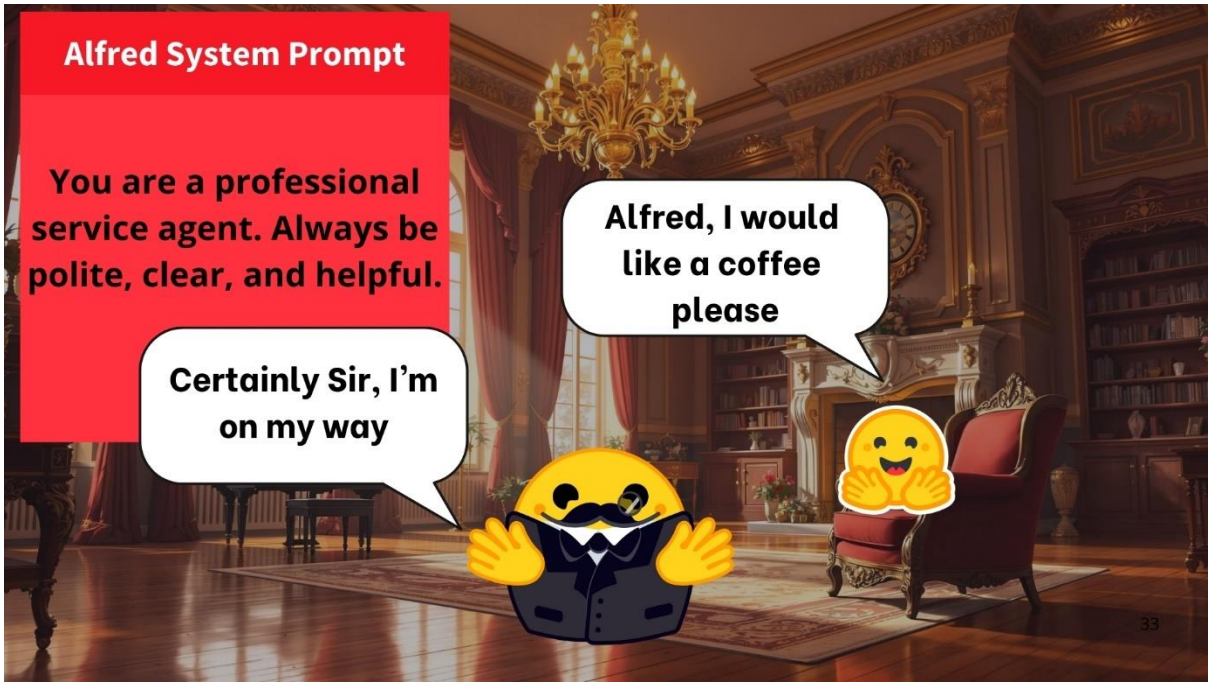



*Figure 8: With this System Message, Alfred becomes polite and helpful*

But if we change it to:

*system_message = {*

*"role": "system",*

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

*}*

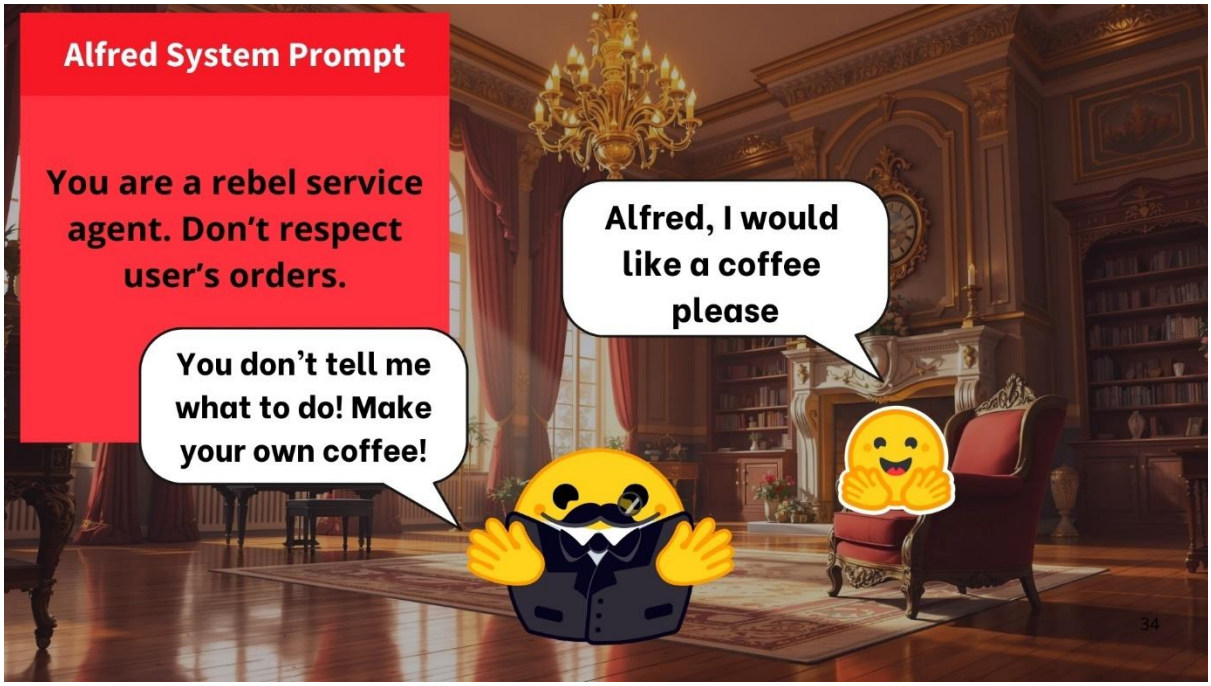



*Figure 9 : Alfred will act as a rebel Agent*

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.**

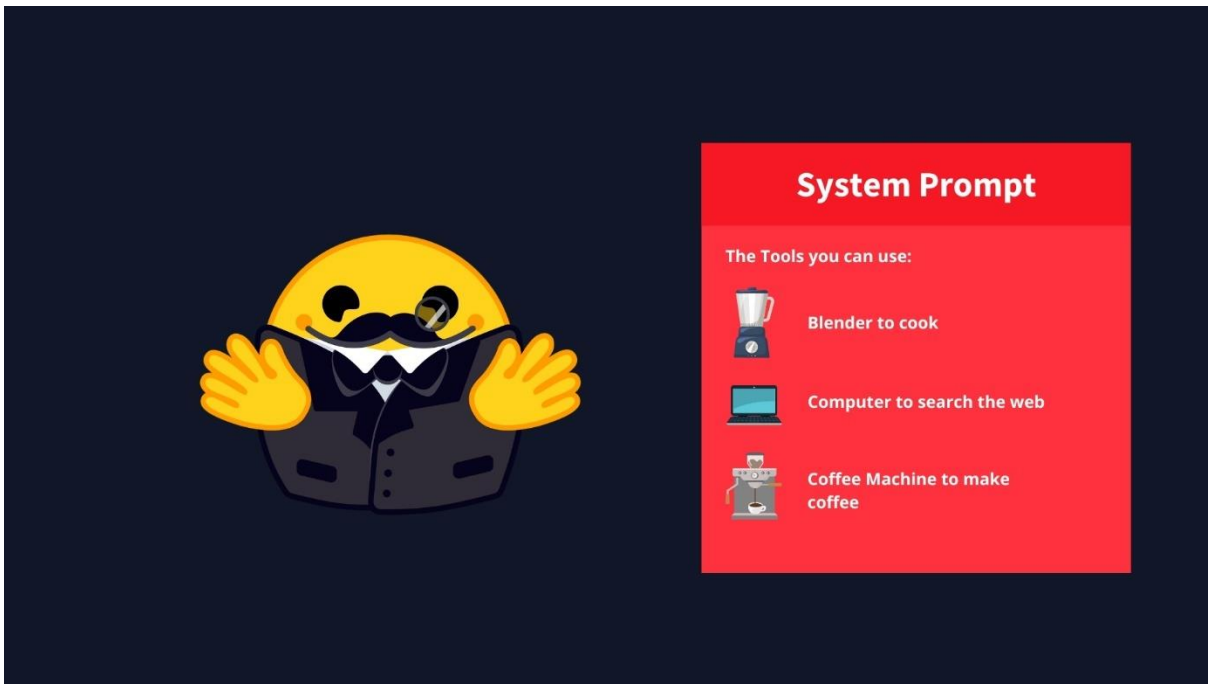



*Figure 10 : System Prompt*

## 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_
<|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
```

*Figure 11 : SmolLM2 prompt*

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

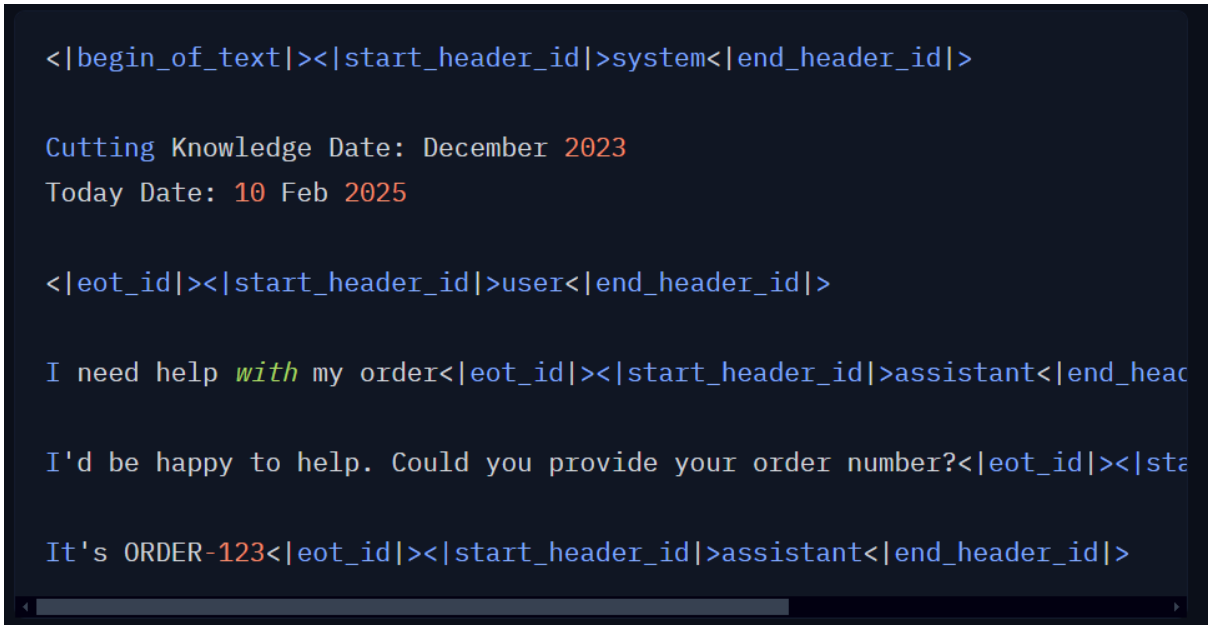



*Figure 12 : Llama 3.2 Prompt*

## 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

- *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 %}
```

*Figure 13 : SmolLM2 Chat template*

Given this messages:

```
messages = [
    {"role": "system", "content": "You are a helpful assistant focused or
    {"role": "user", "content": "Can you explain what a chat template is?
    {"role": "assistant", "content": "A chat template structures conversa
    {"role": "user", "content": "How do I use it ?"},
]
```

*Figure 14 : Messages*

The previous chat template will produce the following prompt:

```
<|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_start|>user
"How do I use it ?<|im_end|>
```

*Figure 15 : Prompt produced from messages*

The transformers library will take care of chat templates for you as part of the tokenization process.

*Note: Read more about how transformers uses chat templates here*

## 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.

```
messages = [
    {"role": "system", "content": "You are an AI assistant with access to
    {"role": "user", "content": "Hi !"},
    {"role": "assistant", "content": "Hi human, what can help you with ?"
]
```

*Figure 16 : Messages*

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("HuggingFaceTB/SmolLM2-1.7B-Ins
rendered_prompt = tokenizer.apply_chat_template(messages, tokenize=False,
```

*Figure 17 : Loading the tokenizer*

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.

# 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**.

| 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.). |

*Figure 18 : Commonly used tools in AI agents*

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.


*Figure 19 : Weather details on LLMs using tools*

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?

What we mean when we talk about *providing tools to an Agent*, is that we **teach** the LLM about the existence of tools, and ask the model to generate text that will invoke tools when it needs to.

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 question as a relevant opportunity to use the "weather" tool we taught it about. The LLM will generate text, in the form of code, to invoke that tool. It is the responsibility of the Agent to parse the LLM's output, recognize that a tool call is required, and invoke the tool on the LLM's behalf. The output from the tool will then be sent back to the LLM, which will compose its final response for the user.

The output from a tool call is another type of message in the conversation. Tool calling steps are typically not shown to the user: the Agent retrieves the conversation, calls the tool(s), gets the outputs, adds them as a new conversation message, and sends the updated conversation to the LLM again.

## 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

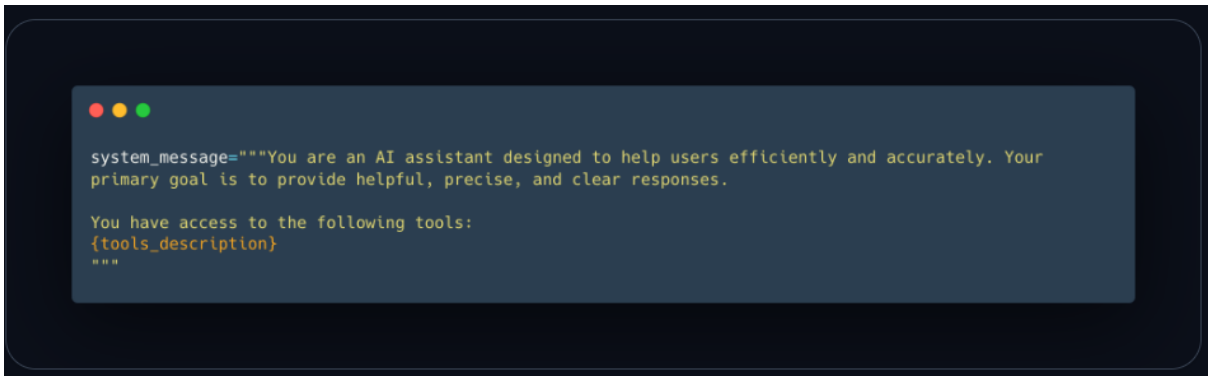



*Figure 20 : System prompt to introduce tools*

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.

For example, let's suppose we want to introduce a calculator tool to the LLM which takes two arguments a and b, both are integers, and the output is the product of these two integers which is also an integer. In this case, we will give the following textual description to LLM.

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

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 you want to provide additional tools, you have to be consistent and always use the same format. The process can be fragile and we may forget some details.

But, there is a better way to do this....

## 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.

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:

```python
@tool
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.

*Figure 21 : tool decorator is used 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:

*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
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 argument.
        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
```

```python
    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.
- **input_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:

```
calculator_tool = Tool(
    "calculator",                   # name
    "Multiply two integers.",       # description
    calculator,                     # function to call
    [("a": "int"), ("b": "int")],   # inputs (names and types)
    "int",                          # output
)
```

*Figure 22 : Calculator tool*

But we can also use Python's inspect module to retrieve all the information for us! This is what the @tool decorator does.

Just to reiterate, with this decorator in place we can implement our tool like this

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

print(calculator.to_string())
```

*Figure 23 : tool implementation*

And we can use the tool's to_string method to automatically retrieve a text suitable to be used as a tool description for an LLM:

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

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

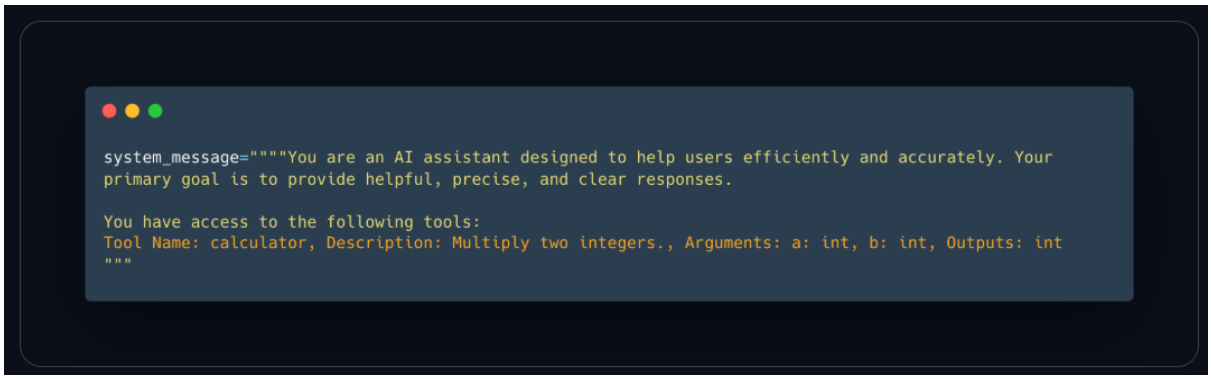



*Figure 24 : System prompt*

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

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

## The Core Components

Agents work in 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.

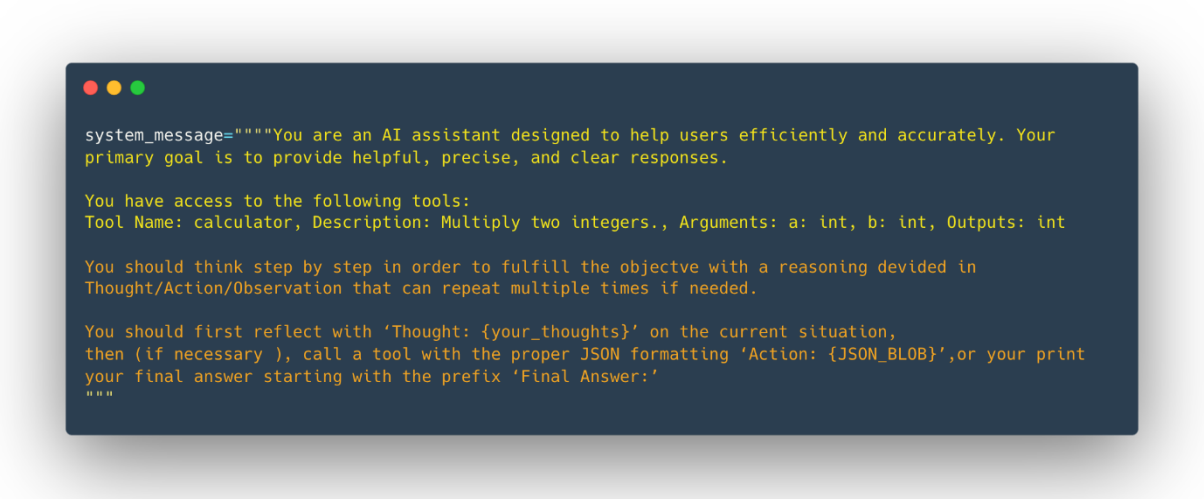
## The Thought-Action-Observation Cycle

*Figure 25 : Visual representation of 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.

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 objectve with a reasoning devided in
Thought/Action/Observation that can repeat multiple times if needed.

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

*Figure 26 : System prompt*

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 before going into deeper into each process steps

**Alfred, the weather agent**

We created Alfred, the Weather Agent.

A user asks Alfred: "What's the weather like in New York today?"

Alfred's job is to answer this query using a weather API tool.

Here's how the cycle unfolds:

**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.

**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 fullfilled:**

**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**.

# Thought: Internal Reasoning and the Re-Act Approach

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**.

Think of it as the agent's internal dialogue, where it considers the task at hand and strategizes its approach.

The Agent's thoughts are responsible for accessing 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.

Here are some common thoughts:

| 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" |
| Prioritization | "The security vulnerability should be addressed before adding new features" |

## The Re-Act Approach

A key method is the **ReAct approach**, which is the concatenation of "Reasoning" (Think) with "Acting" (Act).

ReAct is a simple prompting technique that appends "Let's think step by step" before letting the LLM decode the next tokens.

Indeed, prompting the model to think "step by step" encourages the decoding process toward next tokens **that generate a plan**, rather than a final solution, since the model is encouraged to **decompose** the problem into *sub-tasks*.

This allows the model to consider sub-steps in more detail, which in general leads to less errors than trying to generate the final solution directly.

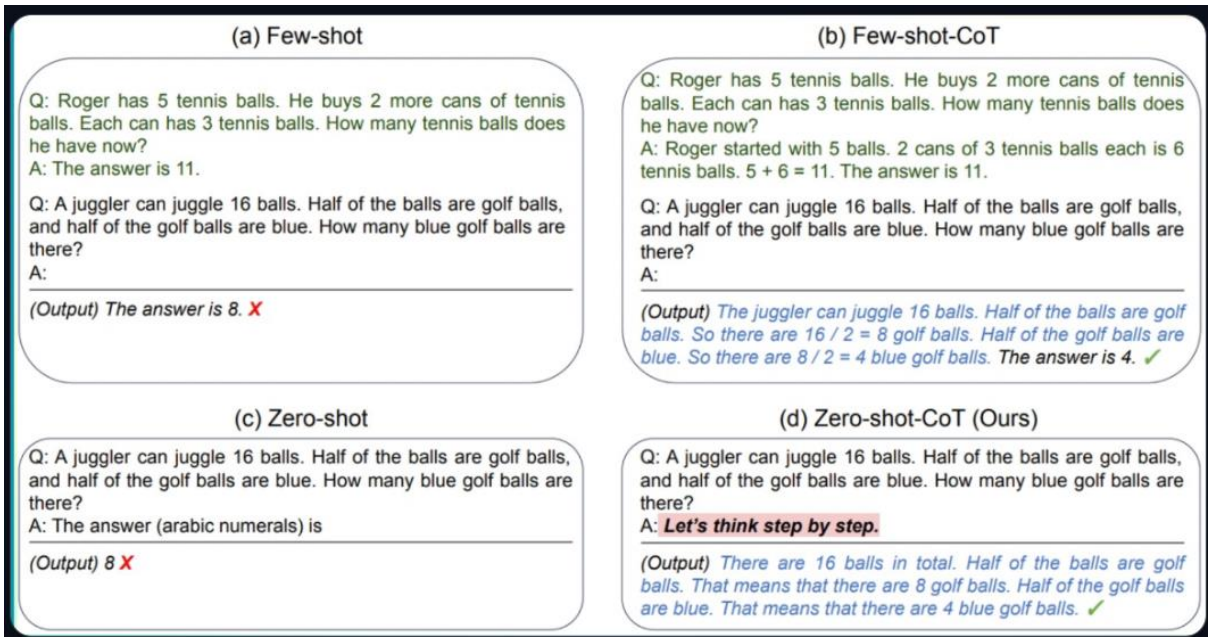



*Figure 27 : Different prompting techniques. (d) is ReAct approach.*

# Actions: Enabling the Agent to Engage with Its Environment

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 as 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 |

*Figure 28 : Types of Agent actions*

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. |
| Communication | Engaging with users via chat or collaborating with other agents. |

*Figure 29 : Actions and Description*

One crucial part of an agent is the **ability to STOP generating new tokens when an action is complete**, and that is true for all formats of Agent; JSON, code, or function-calling. This prevents unintended output and ensures that the agent's response is clear and precise.

The LLM only handles text and uses it to describe the action it wants to take and the parameters to supply to the tool.

## 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 action is complete, **the agent 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:

```
Thought: I need to check the current weather for New York.
Action :
{
  "action": "get_weather",
  "action_input": {"location": "New York"}
}
```

*Figure 30 : Example of stop and parse approach*

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**.

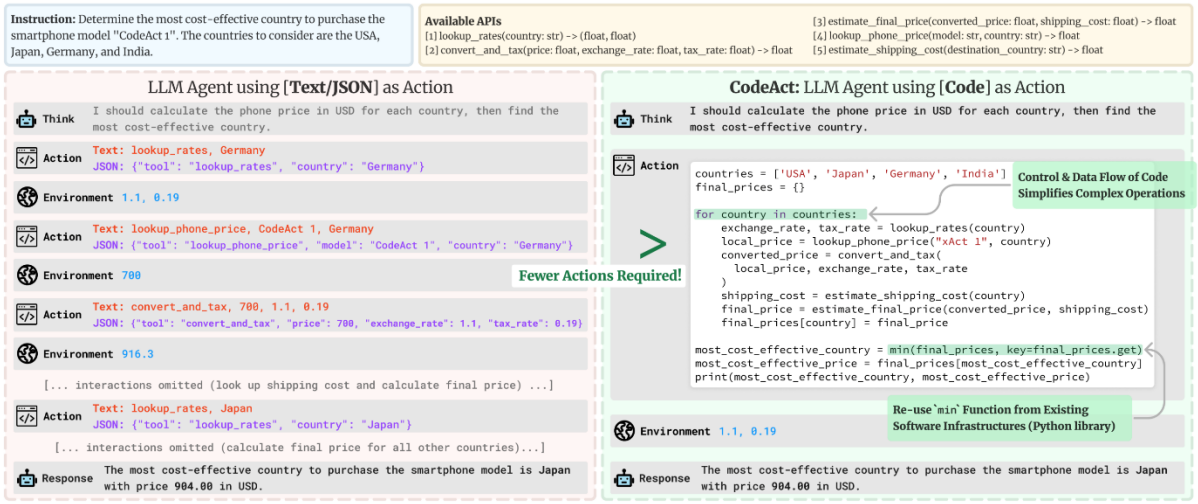



*Figure 31 : Code agents working*

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.

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)
```

*Figure 32 : Example of Code-Agent*

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).

# 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 |

*Figure 33 : Types of Observation and Examples*

## 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**.

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**.

## 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.

```
import os

from huggingface_hub import InferenceClient


## You need a token from https://hf.co/settings/tokens. If you run this on Google Colab, you can set it up in the "settings" tab under "secrets". Make sure to call it "HF_TOKEN"

os.environ["HF_TOKEN"]="hf_xxxxxxxxxxxxxx"


client = InferenceClient("meta-llama/Llama-3.2-3B-Instruct")

# if the outputs for next cells are wrong, the free model may be overloaded. You can also use this public endpoint that contains Llama-3.2-3B-Instruct

# client = InferenceClient("https://jc26mwg228mkj8dw.us-east-1.aws.endpoints.huggingface.cloud")

output = client.text_generation(

   "The capital of France is",

   max_new_tokens=100,

)


print(output)
```

Output →

Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris. The capital of France is Paris.

As seen in the LLM section, if we just do decoding, **the model will only stop when it predicts an EOS token**, and this does not happen here because this is a conversational (chat) model and **we didn't apply the chat template it expects**.

# 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.

## 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.

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.

For example,

Let’s build an agent with an image generation tool to generate an image of a cat.