Hugging Face Agents Course

Table of Contents

Agents Course

(Optional) Discord 101

- The Agents course on Hugging Face’s Discord Community
- Tips for using Discord effectively
  - How to join a server
  - How to use Discord effectively

Agents Course

Introduction to Agents

Agents Course

What are LLMs?

- What is a Large Language Model?
- Understanding next token prediction.
- Attention is all you need
- Prompting the LLM is important
- How are LLMs trained?
- How can I use LLMs?
- How are LLMs used in AI Agents?

Agents Course

Messages and Special Tokens

- Messages: The Underlying System of LLMs
- Chat-Templates
  - System Messages
  - Conversations: User and Assistant Messages
  - Base Models vs. Instruct Models
  - Understanding Chat Templates
  - Messages to prompt

Agents Course

What are Tools?

- What are AI Tools?

Dummy Agent

Agents Course

Let’s Create Our First Agent Using smolagents

What is smolagents?

Let’s build our Agent!

The Tools

The Agent

The System Prompt

Agents Course

Q1: What is an Agent?

Q2: What is the Role of Planning in an Agent?

Q3: How Do Tools Enhance an Agent’s Capabilities?

Q4: How Do Actions Differ from Tools?

Q5: What Role Do Large Language Models (LLMs) Play in Agents?

Q6: Which of the Following Best Demonstrates an AI Agent?

Agents Course

Quick Self-Check (ungraded)

Q1: Which of the following best describes an AI tool?

Q2: How do AI agents use tools as a form of “acting” in an environment?

Q3: What is a Large Language Model (LLM)?

Q4: Which of the following best describes the role of special tokens in LLMs?

Q5: How do AI chat models process user messages internally?

Agents Course

Unit 1 Quiz

Quiz

Agents Course

Introduction to LlamaIndex

What Makes LlamaIndex Special?

Agents Course

Tools

Tool Creation Methods

Default Toolbox

Sharing and Importing Tools

Resources

The @tool Decorator

Defining a Tool as a Python Class

Sharing a Tool to the Hub

Importing a Tool from the Hub

Importing a Hugging Face Space as a Tool

Importing a LangChain Tool

Importing a tool collection from any MCP server

Generating a tool that retrieves the highest-rated catering

Generating a tool to generate ideas about the superhero-themed party

Agents Course

Using Tools in LlamaIndex

Creating a FunctionTool

Creating a QueryEngineTool

Creating Toolspecs

Utility Tools

Model Context Protocol (MCP) in LlamaIndex

Agents Course

Small Quiz (ungraded)

Q1: What is one of the primary advantages of choosing smolagents over other frameworks?

Q2: In which scenario would you likely benefit most from using smolagents?

Q3: smolagents offers flexibility in model integration. Which statement best reflects its approach?

Q4: How does smolagents handle the debate between code-based actions and JSON-based actions?

Q5: How does smolagents integrate with the Hugging Face Hub for added benefits?

Agents Course

Test Your Understanding of LangGraph

Q1: What is the primary purpose of LangGraph?

Q2: In the context of the “Control vs Freedom” trade-off, where does LangGraph stand?

Q3: What role does State play in LangGraph?

Q4: What is a Conditional Edge in LangGraph?

Q5: How does LangGraph help address the hallucination problem in LLMs?

Agents Course

Small Quiz (ungraded)

Q1: What is a QueryEngine?

Q2: What is the Purpose of FunctionTools?

Q3: What are Toolspecs in LlamaIndex?

Q4: What is Required to create a tool?

Agents Course

Small Quiz (ungraded)

Q1: What is the key difference between creating a tool with the @tool decorator versus creating a subclass of Tool in smolagents?

Q2: How does a CodeAgent handle multi-step tasks using the ReAct (Reason + Act) approach?

Q3: Which of the following is a primary advantage of sharing a tool on the Hugging Face Hub?

Q4: ToolCallingAgent differs from CodeAgent in how it executes actions. Which statement is correct?

Q5: What is included in the smolagents default toolbox, and why might you use it?

Agents Course

Quick Self-Check (ungraded)

Q1: What is the purpose of AgentWorkflow in LlamaIndex?

Q2: What object is used for keeping track of the state of the workflow?

Q3: Which method should be used if you want an agent to remember previous interactions?

Q4: What is a key feature of Agentic RAG?

Agents Course

Conclusion

Keep Learning, stay awesome ■

Agents Course

Conclusion

Keep Learning, and stay awesome ■

Agents Course

Conclusion

Keep Learning, Stay Awesome! ■

Agents Course

Why use smolagents

What is smolagents ?

Resources

Key Advantages of smolagents

When to use smolagents?

Code vs. JSON Actions

Resources

Agents Course

Building Agents That Use Code

Why Code Agents?

How Does a Code Agent Work?

Let’s See Some Examples

Resources

Selecting a Playlist for the Party Using smolagents

Using a Custom Tool to Prepare the Menu

Using Python Imports Inside the Agent

Sharing Our Custom Party Preparator Agent to the Hub

Inspecting Our Party Preparator Agent with OpenTelemetry and Langfuse ■

Agents Course

What is LangGraph ?

Is LangGraph different from LangChain ?

When should I use LangGraph ?

How does LangGraph work?

How is it different from regular python? Why do I need LangGraph?

Control vs freedom

Agents Course

Document Analysis Graph

The Butler’s Workflow

Setting Up the environment

Defining Agent’s State

Preparing Tools

The nodes

The ReAct Pattern: How I Assist Mr. Wayne

The Butler in Action

Step 2: Create the Retriever Tool

Step 3: Integrate the Tool with Alfred

Agents Course

Welcome to the final Unit

What's the challenge?

Agents Course

Conclusion

Agents Course

What is GAIA?

■ GAIA's Core Principles

Difficulty Levels

Example of a Hard GAIA Question

Live Evaluation

Agents Course

And now? What topics I should learn?

■ Model Context Protocol (MCP)

■ Agent-to-Agent (A2A) Protocol

Agents Course

Claim Your Certificate ■

Agents Course

Hands-On

The Dataset

The process

Unit 0

Welcome to the course ■

Source: https://huggingface.co/learn/agents-course/unit0/introduction

Agents Course

Welcome to the ■ AI Agents Course

Welcome to the most exciting topic in AI today: Agents !
This free course will take you on a journey, from beginner to expert , in understanding, using and building AI agents.
This first unit will help you onboard:

- Discover the course's syllabus .
- Choose the path you're going to take (either self-audit or certification process).
- Get more information about the certification process and the deadlines .
- Get to know the team behind the course.
- Create your Hugging Face account .
- Sign-up to our Discord server , and meet your classmates and us.

Let's get started!

What to expect from this course?

In this course, you will:

- ■ Study AI Agents in theory, design, and practice.
- ■■■ Learn to use established AI Agent libraries such as smolagents , LlamaIndex , and LangGraph .
- ■ Share your agents on the Hugging Face Hub and explore agents created by the community.
- ■ Participate in challenges where you will evaluate your agents against other students'.
- ■ Earn a certificate of completion by completing assignments.

And more!
At the end of this course you'll understand how Agents work and how to build your own Agents using the latest libraries and tools .
Don't forget to sign up to the course!

(We are respectful of your privacy. We collect your email address to be able to send you the links when each Unit is published and give you information about the challenges and updates).

What does the course look like?

The course is composed of:

- Foundational Units : where you learn Agents concepts in theory .
- Hands-on : where you'll learn to use established AI Agent libraries to train your agents in unique environments. These hands-on sections will be Hugging Face Spaces with a pre-configured environment.
- Use case assignments : where you'll apply the concepts you've learned to solve a real-world problem that you'll choose.
- The Challenge : you'll get to put your agent to compete against other agents in a challenge. There will also be a leaderboard (not available yet) for you to compare the agents' performance.

This course is a living project, evolving with your feedback and contributions! Feel free to open issues and PRs in GitHub , and engage in discussions in our Discord server.
After you have gone through the course, you can also send your feedback ■ using this form

What's the syllabus?

Here is the general syllabus for the course . A more detailed list of topics will be released with each unit.
We are also planning to release some bonus units, stay tuned!

What are the prerequisites?

To be able to follow this course you should have a:

- Basic knowledge of Python
- Basic knowledge of LLMs (we have a section in Unit 1 to recap what they are)

What tools do I need?

You only need 2 things:

- A computer with an internet connection.
- A Hugging Face Account : to push and load models, agents, and create Spaces. If you don't have an account yet, you can create one here (it's free).

The Certification Process

You can choose to follow this course in audit mode , or do the activities and get one of the two certificates we'll issue .
If you audit the course, you can participate in all the challenges and do assignments if you want, and you don't need to notify us .
The certification process is completely free :

- To get a certification for fundamentals : you need to complete Unit 1 of the course. This is intended for students that want to get up to date with the latest trends in Agents.
- To get a certificate of completion : you need to complete Unit 1, one of the use case assignments we'll propose during the course, and the final challenge.

There's a deadline for the certification process: all the assignments must be finished before July 1st 2025 .

What is the recommended pace?

Each chapter in this course is designed to be completed in 1 week, with approximately 3-4 hours of work per week .
Since there's a deadline, we provide you a recommended pace:

How to get the most out of the course?

To get the most out of the course, we have some advice:

- Join study groups in Discord : studying in groups is always easier. To do that, you need to join our discord server and verify your Hugging Face account.
- Do the quizzes and assignments : the best way to learn is through hands-on practice and self-assessment.
- Define a schedule to stay in sync : you can use our recommended pace schedule below or create yours.

Who are we

About the authors:

Acknowledgments

We would like to extend our gratitude to the following individuals for their invaluable contributions to this course:

- Pedro Cuenca – For his guidance and expertise in reviewing the materials.
- Aymeric Roucher – For his amazing demo spaces (decoding and final agent) as well as his help on the smolagents parts.
- Joshua Lochner – For his amazing demo space on tokenization.
- Quentin Gallouédec – For his help on the course content.
- David Berenstein – For his help on the course content and moderation.
- XiaXiao (ShawnSiao) – Chinese translator for the course.
- Jiaming Huang – Chinese translator for the course.

I found a bug, or I want to improve the course

Contributions are welcome ■

- If you found a bug ■ in a notebook , please open an issue and describe the problem .
- If you want to improve the course , you can open a Pull Request.
- If you want to add a full section or a new unit , the best is to open an issue and describe what content you want to add before starting to write it so that we can guide you .

I still have questions

Please ask your question in our discord server #ai-agents-discussions.
Now that you have all the information, let’s get on board ■

Joffrey Thomas

Joffrey is a machine learning engineer at Hugging Face and has built and deployed AI Agents in production. Joffrey will be your main instructor for this course.

- Follow Joffrey on Hugging Face
- Follow Joffrey on X
- Follow Joffrey on Linkedin

Ben Burtenshaw

Ben is a machine learning engineer at Hugging Face and has delivered multiple courses across various platforms. Ben’s goal is to make the course accessible to everyone.

- Follow Ben on Hugging Face
- Follow Ben on X
- Follow Ben on Linkedin

Thomas Simonini

Thomas is a machine learning engineer at Hugging Face and delivered the successful Deep RL and ML for games courses. Thomas is a big fan of Agents and is excited to see what the community will build.

- Follow Thomas on Hugging Face
- Follow Thomas on X
- Follow Thomas on Linkedin

Sergio Paniego

Sergio is a machine learning engineer at Hugging Face. He contributed to several sections of Units 2, 3, 4, and the bonus units.

- Follow Sergio on Hugging Face
- Follow Sergio on X
- Follow Sergio on Linkedin

Onboarding

Source: https://huggingface.co/learn/agents-course/unit0/onboarding

Agents Course

Onboarding: Your First Steps ■

Now that you have all the details, let’s get started! We’re going to do four things:

- Create your Hugging Face Account if it’s not already done
- Sign up to Discord and introduce yourself (don’t be shy ■)
- Follow the Hugging Face Agents Course on the Hub
- Spread the word about the course

Step 1: Create Your Hugging Face Account

(If you haven’t already) create a Hugging Face account here .

Step 2: Join Our Discord Community

■■ Join our discord server here.
When you join, remember to introduce yourself in #introduce-yourself .
We have multiple AI Agent-related channels:

- agents-course-announcements : for the latest course information .
- ■-agents-course-general : for general discussions and chitchat .
- agents-course-questions : to ask questions and help your classmates .
- agents-course-showcase : to show your best agents .

In addition you can check:

- smolagents : for discussion and support with the library .

If this is your first time using Discord, we wrote a Discord 101 to get the best practices. Check the next section .

Step 3: Follow the Hugging Face Agent Course Organization

Stay up to date with the latest course materials, updates, and announcements by following the Hugging Face Agents Course Organization .
■ Go here and click on follow .

Step 4: Spread the word about the course

Help us make this course more visible! There are two way you can help us:

- Show your support by ■ the course's repository .
- Share Your Learning Journey: Let others know you're taking this course ! We've prepared an illustration you can use in your social media posts

You can download the image by clicking ■ here

Step 5: Running Models Locally with Ollama (In case you run into Credit limits)

- Install Ollama Follow the official Instructions here.
- Pull a model Locally
- Start Ollama in the background (In one terminal)
- Use LiteLLMModel Instead of HfApiModel
- Why this works?
- Ollama serves models locally using an OpenAI-compatible API at http://localhost:11434 .
- LiteLLMModel is built to communicate with any model that supports the OpenAI chat/completion API format.
- This means you can simply swap out HfApiModel for LiteLLMModel no other code changes required. It's a seamless, plug-and-play solution.

Congratulations! ■ You've completed the onboarding process ! You're now ready to start learning about AI Agents. Have fun!
Keep Learning, stay awesome ■

Discord 101

Source: https://huggingface.co/learn/agents-course/unit0/discord101

Agents Course

(Optional) Discord 101

This guide is designed to help you get started with Discord, a free chat platform popular in the gaming and ML communities.
Join the Hugging Face Community Discord server, which has over 100,000 members , by clicking here . It’s a great place to connect with others!

The Agents course on Hugging Face’s Discord Community

Starting on Discord can be a bit overwhelming, so here’s a quick guide to help you navigate.
The HF Community Server hosts a vibrant community with interests in various areas, offering opportunities for learning through paper discussions, events, and more.
After signing up , introduce yourself in the #introduce-yourself channel.
We created 4 channels for the Agents Course:

- agents-course-announcements : for the latest course informations .
- ■-agents-course-general : for general discussions and chitchat .
- agents-course-questions : to ask questions and help your classmates .
- agents-course-showcase : to show your best agents .

In addition you can check:

- smolagents : for discussion and support with the library .

Tips for using Discord effectively

How to join a server

If you are less familiar with Discord, you might want to check out this guide on how to join a server. Here's a quick summary of the steps:

- Click on the Invite Link .
- Sign in with your Discord account, or create an account if you don't have one.
- Validate that you are not an AI agent!
- Setup your nickname and avatar.
- Click "Join Server".

How to use Discord effectively

Here are a few tips for using Discord effectively:

- Voice channels are available, though text chat is more commonly used.
- You can format text using markdown style , which is especially useful for writing code. Note that markdown doesn't work as well for links.
- Consider opening threads for long conversations to keep discussions organized.

We hope you find this guide helpful! If you have any questions, feel free to ask us on Discord ■.

Unit 1

Introduction

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

Agents Course

Introduction to Agents

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?
- 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.
- 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.
- 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 .
- 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.
This Unit is your essential starting point , laying the groundwork for understanding Agents before you move on to more advanced topics.
It's a big unit, so take your time and don't hesitate to come back to these sections from time to time.
Ready? Let's dive in! ■

What are LLMs?

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

Agents Course

What are 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.
There are 3 types of transformers:

- bullet 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
- bullet Example : BERT from Google
- bullet Use Cases : Text classification, semantic search, Named Entity Recognition
- bullet Typical Size : Millions of parameters
- bullet 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
- bullet Example : Llama from Meta
- bullet 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
- 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:
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:
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.

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.
In other words, an LLM will decode text until it reaches the EOS. But what happens during a single decoding loop?
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.

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):

- 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.
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:

- Run Locally (if you have sufficient hardware).
- 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

Source: https://huggingface.co/learn/agents-course/unit1/messages-and-special-tokens

Agents Course

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.
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

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.

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:
With this System Message, Alfred becomes polite and helpful:
But if we change it to:
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:
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:
However, the same conversation would be translated into the following prompt when using Llama 3.2:
Templates can handle complex multi-turn conversations while maintaining context:

Base Models vs. Instruct Models

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

- bullet A Base Model is trained on raw text data to predict the next token.
- bullet 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:
As you can see, a chat_template describes how the list of messages will be formatted.
Given these messages:
The previous chat template will produce the following string:
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.
To convert the previous conversation into a prompt, we load the tokenizer and call apply_chat_template :
The rendered_prompt returned by this function is now ready to use as the input for the model you chose!
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:

bullet Hugging Face Chat Templating Guide

bullet Transformers Documentation

What are Tools?

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

Agents Course

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:
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.
- 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:

bullet What the tool does

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

bullet a (int): An integer.

bullet b (int): An integer.

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

bullet (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.
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:

- bullet A descriptive name of what it does: calculator
- bullet A longer description, provided by the function's docstring comment: Multiply two integers.
- bullet The inputs and their type: the function clearly expects two int s.
- bullet 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:
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:
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.
It may seem complicated, but if we go slowly through it we can see what it does. We define a Tool class that includes:

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

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

- bullet A growing list of pre-built integrations that your LLM can directly plug into
- bullet The flexibility to switch between LLM providers and vendors
- bullet 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.
Tools play a crucial role in enhancing the capabilities of AI agents.
To summarize, we learned:

- bullet What Tools Are : Functions that give LLMs extra capabilities, such as performing calculations or accessing external data.
- bullet How to Define a Tool : By providing a clear textual description, inputs, outputs, and a callable function.
- bullet 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.
But first, it's time for another short quiz!

Understanding AI Agents through the Thought-Action-Observation Cycle

Source: https://huggingface.co/learn/agents-course/unit1/agent-steps-and-structure

Agents Course

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 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:

- Thought : The LLM part of the Agent decides what the next step should be.
- Action: The agent takes an action, by calling the tools with the associated arguments.
- 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:
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?"
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.
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:

bullet 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.

bullet 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.

bullet 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.
Let’s now dive deeper into the Thought, Action, Observation as the individual steps of the process.

Thoughts: Internal Reasoning and the Be-An-AgentPlus

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

Agents Course

Thought: Internal Reasoning and the ReAct 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 examples of common thoughts:

The ReAct 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.
Now that we better understand the Thought process, let's go deeper on the second part of the process: Act.

Actions: Enabling the Agent to Engage with its Environment

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

Agents Course

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:
Actions themselves can serve many purposes:
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:

bullet Generation in a Structured Format :

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

bullet Halting Further Generation :

Once the action is complete, the agent stops generating additional tokens . This prevents extra or erroneous output.

- 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:
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.

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

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

Agents Course

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.

How Are the Results Appended?

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

- Parse the action to identify the function(s) to call and the argument(s) to use.
- Execute the action.
- 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

Source: https://huggingface.co/learn/agents-course/unit1/dummy-agent-library

Agents Course

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 , LangChain , 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.
output:
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 .
If we now add the special tokens related to the Llama-3.2-3B-Instruct model that we're using, the behavior changes and it now produces the expected EOS.
output:
Using the "chat" method is a much more convenient and reliable way to apply chat templates:
output:
The chat method is the RECOMMENDED method to use in order to ensure a smooth transition between models, but since this notebook is only educational, we will keep using the "text_generation"

method to understand the details.

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:

- Information about the tools
- Cycle instructions (Thought → Action → Observation)

Since we are running the “text_generation” method, we need to apply the prompt manually:
We can also do it like this, which is what happens inside the chat method :
The prompt now is :
Let’s decode!
output:
Do you see the issue?
output:
Much Better! Let’s now create a dummy get weather function. In a real situation, you would likely call an API.
output:
Let’s concatenate the base prompt, the completion until function execution and the result of the function as an Observation and resume generation.
Here is the new prompt:
Output:
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.
Now, we’re ready to create our first real Agent using the smolagents library.

Let's Create Our First Agent Using smokeagents

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

Agents Course

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:
Exciting, right?

Let’s build our Agent!

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

- First, get your Hugging Face token from https://hf.co/settings/tokens with permission for inference, if you don’t already have one
- Go to your duplicated Space and click on the Settings tab
- Scroll down to the Variables and Secrets section and click New Secret
- Create a secret with the name HF_TOKEN and paste your token as the value
- 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

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

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

To define your tool it is important to:

- Provide input and output types for your function, like in get_current_time_in_timezone(timezone: str) -> str:
- 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.
This Agent still uses the InferenceClient we saw in an earlier section behind the HfApiModel 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 seperate 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":
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!

Quick Quiz 1

Source: https://huggingface.co/learn/agents-course/unit1/quiz1

Agents Course

Q1: What is an Agent?

Which of the following best describes an AI Agent?

Q2: What is the Role of Planning in an Agent?

Why does an Agent need to plan before taking an action?

Q3: How Do Tools Enhance an Agent’s Capabilities?

Why are tools essential for an Agent?

Q4: How Do Actions Differ from Tools?

What is the key difference between Actions and Tools?

Q5: What Role Do Large Language Models (LLMs) Play in Agents?

How do LLMs contribute to an Agent’s functionality?

Q6: Which of the Following Best Demonstrates an AI Agent?

Which real-world example best illustrates an AI Agent at work?
Congrats on finishing this Quiz ■! If you need to review any elements, take the time to revisit the chapter to reinforce your knowledge before diving deeper into the “Agent’s brain”: LLMs.

Quick Quiz 2

Source: https://huggingface.co/learn/agents-course/unit1/quiz2

Agents Course

Quick Self-Check (ungraded)

What?! Another Quiz? We know, we know, … ■ But this short, ungraded quiz is here to help you reinforce key concepts you've just learned .
This quiz covers Large Language Models (LLMs), message systems, and tools; essential components for understanding and building AI agents.

Q1: Which of the following best describes an AI tool?

Q2: How do AI agents use tools as a form of "acting" in an environment?

Q3: What is a Large Language Model (LLM)?

Q4: Which of the following best describes the role of special tokens in LLMs?

Q5: How do AI chat models process user messages internally?

Got it? Great! Now let’s dive into the complete Agent flow and start building your first AI Agent!

Unit 1 Final Quiz

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

Agents Course

Unit 1 Quiz

Well done on working through the first unit! Let’s test your understanding of the key concepts covered so far.
When you pass the quiz, proceed to the next section to claim your certificate.
Good luck!

Quiz

Here is the interactive quiz. The quiz is hosted on the Hugging Face Hub in a space. It will take you through a set of multiple choice questions to test your understanding of the key concepts covered in this unit. Once you’ve completed the quiz, you’ll be able to see your score and a breakdown of the correct answers.
One important thing: don’t forget to click on Submit after you passed, otherwise your exam score will not be saved!
You can also access the quiz ■ here

Certificate

Now that you have successfully passed the quiz, you can get your certificate ■
When you complete the quiz, it will grant you access to a certificate of completion for this unit. You can download and share this certificate to showcase your progress in the course.
Once you receive your certificate, you can add it to your LinkedIn ■■■ or share it on X, Bluesky, etc.
We would be super proud and would love to congratulate you if you tag @huggingface ! ■

Conclusion

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

Agents Course

Conclusion

Congratulations on finishing this first Unit ■
You've just mastered the fundamentals of Agents and you've created your first AI Agent!
It's normal if you still feel confused by some of these elements . Agents are a complex topic and it's common to take a while to grasp everything.
Take time to really grasp the material before continuing. It's important to master these elements and have a solid foundation before entering the fun part.
And if you pass the Quiz test, don't forget to get your certificate ■ ■ here
In the next (bonus) unit, you're going to learn to fine-tune a Agent to do function calling (aka to be able to call tools based on user prompt) .
Finally, we would love to hear what you think of the course and how we can improve it . If you have some feedback then, please ■ fill this form

Keep Learning, stay awesome ■

Unit 2

Frameworks for AI Agents

Source: https://huggingface.co/learn/agents-course/unit2/introduction

Agents Course

Introduction to Agentic Frameworks

Welcome to this second unit, where we'll explore different agentic frameworks that can be used to build powerful agentic applications.
We will study:

- In Unit 2.1: smolagents
- In Unit 2.2: LlamaIndex
- In Unit 2.3: LangGraph

Let's dive in! ■

When to Use an Agentic Framework

An agentic framework is not always needed when building an application around LLMs . They provide flexibility in the workflow to efficiently solve a specific task, but they're not always necessary.
Sometimes, predefined workflows are sufficient to fulfill user requests, and there is no real need for an agentic framework. If the approach to build an agent is simple, like a chain of prompts, using plain code may be enough. The advantage is that the developer will have full control and understanding of their system without abstractions .
However, when the workflow becomes more complex, such as letting an LLM call functions or using multiple agents, these abstractions start to become helpful.
Considering these ideas, we can already identify the need for some features:

- An LLM engine that powers the system.
- A list of tools the agent can access.
- A parser for extracting tool calls from the LLM output.
- A system prompt synced with the parser.
- A memory system .
- Error logging and retry mechanisms to control LLM mistakes. We'll explore how these topics are resolved in various frameworks, including smolagents , LlamaIndex , and LangGraph .

Agentic Frameworks Units

Introduction to smokeagents

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/introduction

Agents Course

Introduction to smolagents

Welcome to this module, where you'll learn how to build effective agents using the smolagents library, which provides a lightweight framework for creating capable AI agents.
smolagents is a Hugging Face library; therefore, we would appreciate your support by starring the smolagents repository :

Module Overview

This module provides a comprehensive overview of key concepts and practical strategies for building intelligent agents using smolagents .
With so many open-source frameworks available, it's essential to understand the components and capabilities that make smolagents a useful option or to determine when another solution might be a better fit.
We'll explore critical agent types, including code agents designed for software development tasks, tool calling agents for creating modular, function-driven workflows, and retrieval agents that access and synthesize information.
Additionally, we'll cover the orchestration of multiple agents as well as the integration of vision capabilities and web browsing, which unlock new possibilities for dynamic and context-aware applications.
In this unit, Alfred, the agent from Unit 1, makes his return. This time, he's using the smolagents framework for his internal workings. Together, we'll explore the key concepts behind this framework as Alfred tackles various tasks. Alfred is organizing a party at the Wayne Manor while the Wayne family ■ is away, and he has plenty to do. Join us as we showcase his journey and how he handles these tasks with smolagents !

Contents

During this unit on smolagents , we cover:

Resources

- smolagents Documentation - Official docs for the smolagents library
- Building Effective Agents - Research paper on agent architectures
- Agent Guidelines - Best practices for building reliable agents
- LangGraph Agents - Additional examples of agent implementations
- Function Calling Guide - Understanding function calling in LLMs
- RAG Best Practices - Guide to implementing effective RAG

1■■ Why Use smolagents

smolagents is one of the many open-source agent frameworks available for application development. Alternative options include LlamaIndex and LangGraph , which are also covered in other modules in this course. smolagents offers several key features that might make it a great fit for specific use cases, but we should always consider all options when selecting a framework. We'll explore the advantages and drawbacks of using smolagents , helping you make an informed decision based on your project's requirements.

2■■ CodeAgents

CodeAgents are the primary type of agent in smolagents . Instead of generating JSON or text, these agents produce Python code to perform actions. This module explores their purpose, functionality, and how they work, along with hands-on examples to showcase their capabilities.

3■■ ToolCallingAgents

ToolCallingAgents are the second type of agent supported by smolagents . Unlike CodeAgents , which generate Python code, these agents rely on JSON/text blobs that the system must parse and interpret to execute actions. This module covers their functionality, their key differences from CodeAgents , and it provides an example to illustrate their usage.

4■■ Tools

As we saw in Unit 1, tools are functions that an LLM can use within an agentic system, and they act as the essential building blocks for agent behavior. This module covers how to create tools, their structure, and different implementation methods using the Tool class or the @tool decorator. You'll also learn about the default toolbox, how to share tools with the community, and how to load community-contributed tools for use in your agents.

5■■ Retrieval Agents

Retrieval agents allow models access to knowledge bases, making it possible to search, synthesize, and retrieve information from multiple sources. They leverage vector stores for efficient retrieval and implement Retrieval-Augmented Generation (RAG) patterns. These agents are particularly useful for integrating web search with custom knowledge bases while maintaining conversation context through memory systems. This module explores implementation strategies, including fallback mechanisms for robust information retrieval.

6■■ Multi-Agent Systems

Orchestrating multiple agents effectively is crucial for building powerful, multi-agent systems. By combining agents with different capabilities—such as a web search agent with a code execution agent—you can create more sophisticated solutions. This module focuses on designing, implementing, and managing multi-agent systems to maximize efficiency and reliability.

7■■ Vision and Browser agents

Vision agents extend traditional agent capabilities by incorporating Vision-Language Models (VLMs) , enabling them to process and interpret visual information. This module explores how to design and integrate VLM-powered agents, unlocking advanced functionalities like image-based reasoning, visual data analysis, and multimodal interactions. We will also use vision agents to build a browser agent that can browse the web and extract information from it.

Introduction to LangGraph

Source: https://huggingface.co/learn/agents-course/unit2/langgraph/introduction

Agents Course

Introduction to LangGraph

Welcome to this next part of our journey, where you'll learn how to build applications using the LangGraph framework designed to help you structure and orchestrate complex LLM workflows. LangGraph is a framework that allows you to build production-ready applications by giving you control tools over the flow of your agent.

Module Overview

In this unit, you'll discover:

Resources

- LangGraph Agents - Examples of LangGraph agent
- LangChain academy - Full course on LangGraph from LangChain

1■■ What is LangGraph, and when to use it?

2■■ Building Blocks of LangGraph

3■■ Alfred, the mail sorting butler

4■■ Alfred, the document Analyst agent

5■■ Quiz

By the end of this unit, you'll be equipped to build robust, organized and production ready applications ! That being said, this section is an introduction to langGraph and more advances topics can be discovered in the free langChain academy course : Introduction to LangGraph
Let's get started!

Introduction to LlamaIndex

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/introduction

Agents Course

Introduction to LlamaIndex

Welcome to this module, where you'll learn how to build LLM-powered agents using the LlamaIndex toolkit.
LlamaIndex is a complete toolkit for creating LLM-powered agents over your data using indexes and workflows . For this course we'll focus on three main parts that help build agents in LlamaIndex: Components , Agents and Tools and Workflows .
Let's look at these key parts of LlamaIndex and how they help with agents:

bullet Components : Are the basic building blocks you use in LlamaIndex. These include things like prompts, models, and databases. Components often help connect LlamaIndex with other tools and libraries.

bullet Tools : Tools are components that provide specific capabilities like searching, calculating, or accessing external services. They are the building blocks that enable agents to perform tasks.

bullet Agents : Agents are autonomous components that can use tools and make decisions. They coordinate tool usage to accomplish complex goals.

bullet Workflows : Are step-by-step processes that process logic together. Workflows or agentic workflows are a way to structure agentic behaviour without the explicit use of agents.

What Makes LlamaIndex Special?

While LlamaIndex does some things similar to other frameworks like smolagents, it has some key benefits:

bullet Clear Workflow System : Workflows help break down how agents should make decisions step by step using an event-driven and async-first syntax. This helps you clearly compose and organize your logic.

bullet Advanced Document Parsing with LlamaParse : LlamaParse was made specifically for LlamaIndex, so the integration is seamless, although it is a paid feature.

- bullet — Many Ready-to-Use Components : LlamaIndex has been around for a while, so it works with lots of other frameworks. This means it has many tested and reliable components, like LLMs, retrievers, indexes, and more.
- bullet — LlamaHub : is a registry of hundreds of these components, agents, and tools that you can use within LlamaIndex.

All of these concepts are required in different scenarios to create useful agents. In the following sections, we will go over each of these concepts in detail. After mastering the concepts, we will use our learnings to create applied use cases with Alfred the agent !
Getting our hands on LlamaIndex is exciting, right? So, what are we waiting for? Let’s get started with finding and installing the integrations we need using LlamaHub! ■

Tools

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/tools

Agents Course

Tools

As we explored in unit 1 , agents use tools to perform various actions. In smolagents , tools are treated as functions that an LLM can call within an agent system .
To interact with a tool, the LLM needs an interface description with these key components:

- Name : What the tool is called
- Tool description : What the tool does
- Input types and descriptions : What arguments the tool accepts
- Output type : What the tool returns

For instance, while preparing for a party at Wayne Manor, Alfred needs various tools to gather information - from searching for catering services to finding party theme ideas. Here's how a simple search tool interface might look:

- Name: web_search
- Tool description: Searches the web for specific queries
- Input: query (string) - The search term to look up
- Output: String containing the search results

By using these tools, Alfred can make informed decisions and gather all the information needed for planning the perfect party.
Below, you can see an animation illustrating how a tool call is managed:

Tool Creation Methods

In smolagents , tools can be defined in two ways:

- Using the @tool decorator for simple function-based tools
- Creating a subclass of Tool for more complex functionality

Default Toolbox

smolagents comes with a set of pre-built tools that can be directly injected into your agent. The default toolbox includes:

- PythonInterpreterTool
- FinalAnswerTool
- UserInputTool
- DuckDuckGoSearchTool
- GoogleSearchTool
- VisitWebpageTool

Alfred could use various tools to ensure a flawless party at Wayne Manor:

- First, he could use the DuckDuckGoSearchTool to find creative superhero-themed party ideas.
- For catering, he'd rely on the GoogleSearchTool to find the highest-rated services in Gotham.
- To manage seating arrangements, Alfred could run calculations with the PythonInterpreterTool .
- Once everything is gathered, he'd compile the plan using the FinalAnswerTool .

With these tools, Alfred guarantees the party is both exceptional and seamless. ■■

Sharing and Importing Tools

One of the most powerful features of smolagents is its ability to share custom tools on the Hub and seamlessly integrate tools created by the community. This includes connecting with HF Spaces and LangChain tools , significantly enhancing Alfred's ability to orchestrate an unforgettable party at Wayne Manor. ■
With these integrations, Alfred can tap into advanced event-planning tools—whether it's adjusting the lighting for the perfect ambiance, curating the ideal playlist for the party, or coordinating with Gotham's finest caterers.
Here are examples showcasing how these functionalities can elevate the party experience:

Resources

- Tools Tutorial - Explore this tutorial to learn how to work with tools effectively.
- Tools Documentation - Comprehensive reference documentation on tools.
- Tools Guided Tour - A step-by-step guided tour to help you build and utilize tools efficiently.
- Building Effective Agents - A detailed guide on best practices for developing reliable and high-performance custom function agents.

The @tool Decorator

The @tool decorator is the recommended way to define simple tools . Under the hood, smolagents will parse basic information about the function from Python. So if you name your function clearly and write a good docstring, it will be easier for the LLM to use.
Using this approach, we define a function with:

- A clear and descriptive function name that helps the LLM understand its purpose.
- Type hints for both inputs and outputs to ensure proper usage.
- A detailed description , including an Args: section where each argument is explicitly described. These descriptions provide valuable context for the LLM, so it's important to write them carefully.

Defining a Tool as a Python Class

This approach involves creating a subclass of Tool . For complex tools, we can implement a class instead of a Python function. The class wraps the function with metadata that helps the LLM understand how to use it effectively. In this class, we define:

- name : The tool's name.
- description : A description used to populate the agent's system prompt.
- inputs : A dictionary with keys type and description , providing information to help the Python interpreter process inputs.
- output_type : Specifies the expected output type.
- forward : The method containing the inference logic to execute.

Below, we can see an example of a tool built using Tool and how to integrate it within a CodeAgent .

Sharing a Tool to the Hub

Sharing your custom tool with the community is easy! Simply upload it to your Hugging Face account using the push_to_hub() method.
For instance, Alfred can share his party_theme_tool to help others find the best catering services in Gotham. Here's how to do it:

Importing a Tool from the Hub

You can easily import tools created by other users using the load_tool() function. For example, Alfred might want to generate a promotional image for the party using AI. Instead of building a tool from scratch, he can leverage a predefined one from the community:

Importing a Hugging Face Space as a Tool

You can also import a HF Space as a tool using Tool.from_space() . This opens up possibilities for integrating with thousands of spaces from the community for tasks from image generation to data analysis.
The tool will connect with the spaces Gradio backend using the gradio_client , so make sure to install it via pip if you don't have it already.
For the party, Alfred can use an existing HF Space for the generation of the AI-generated image to be used in the announcement (instead of the pre-built tool we mentioned before). Let's build it!

Importing a LangChain Tool

We'll discuss the LangChain framework in upcoming sections. For now, we just note that we can reuse LangChain tools in your smolagents workflow!
You can easily load LangChain tools using the Tool.from_langchain() method. Alfred, ever the perfectionist, is preparing for a spectacular superhero night at Wayne Manor while the Waynes are away. To make sure every detail exceeds expectations, he taps into LangChain tools to find top-tier entertainment ideas.
By using Tool.from_langchain() , Alfred effortlessly adds advanced search functionalities to his smolagent, enabling him to discover exclusive party ideas and services with just a few commands.
Here's how he does it:

Importing a tool collection from any MCP server

smolagents also allows importing tools from the hundreds of MCP servers available on glama.ai or smithery.ai .
The MCP servers tools can be loaded in a ToolCollection object as follow:
With this setup, Alfred can quickly discover luxurious entertainment options, ensuring Gotham's elite guests have an unforgettable experience. This tool helps him curate the perfect superhero-themed event for Wayne Manor! ■

Generating a tool that retrieves the highest-rated catering

Let's imagine that Alfred has already decided on the menu for the party, but now he needs help preparing food for such a large number of guests. To do so, he would like to hire a catering service and needs to identify the highest-rated options available. Alfred can leverage a tool to search for the best catering services in his area.
Below is an example of how Alfred can use the @tool decorator to make this happen:

Generating a tool to generate ideas about the superhero-themed party

Alfred's party at the mansion is a superhero-themed event , but he needs some creative ideas to make it truly special. As a fantastic host, he wants to surprise the guests with a unique theme.

To do this, he can use an agent that generates superhero-themed party ideas based on a given category. This way, Alfred can find the perfect party theme to wow his guests.
With this tool, Alfred will be the ultimate super host, impressing his guests with a superhero-themed party they won't forget! ■■■■■■■■

Using Tools in LlamaIndex

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/tools

Agents Course

Using Tools in LlamaIndex

Defining a clear set of Tools is crucial to performance. As we discussed in unit 1 , clear tool interfaces are easier for LLMs to use. Much like a software API interface for human engineers, they can get more out of the tool if it's easy to understand how it works.
There are four main types of tools in LlamaIndex :

- FunctionTool : Convert any Python function into a tool that an agent can use. It automatically figures out how the function works.
- QueryEngineTool : A tool that lets agents use query engines. Since agents are built on query engines, they can also use other agents as tools.
- Toolspecs : Sets of tools created by the community, which often include tools for specific services like Gmail.
- Utility Tools : Special tools that help handle large amounts of data from other tools.

We will go over each of them in more detail below.

Creating a FunctionTool

A FunctionTool provides a simple way to wrap any Python function and make it available to an agent. You can pass either a synchronous or asynchronous function to the tool, along with optional name and description parameters. The name and description are particularly important as they help the agent understand when and how to use the tool effectively. Let's look at how to create a FunctionTool below and then call it.

Creating a QueryEngineTool

The QueryEngine we defined in the previous unit can be easily transformed into a tool using the QueryEngineTool class. Let's see how to create a QueryEngineTool from a QueryEngine in the example below.

Creating Toolspecs

Think of ToolSpecs as collections of tools that work together harmoniously - like a well-organized professional toolkit. Just as a mechanic's toolkit contains complementary tools that work together for vehicle repairs, a ToolSpec combines related tools for specific purposes. For example, an accounting agent's ToolSpec might elegantly integrate spreadsheet capabilities, email functionality, and calculation tools to handle financial tasks with precision and efficiency.
And now we can load the toolspec and convert it to a list of tools.
To get a more detailed view of the tools, we can take a look at the metadata of each tool.

Utility Tools

Oftentimes, directly querying an API can return an excessive amount of data , some of which may be irrelevant, overflow the context window of the LLM, or unnecessarily increase the number of tokens that you are using. Let's walk through our two main utility tools below.

bullet OnDemandToolLoader : This tool turns any existing LlamaIndex data loader (BaseReader class) into a tool that an agent can use. The tool can be called with all the parameters needed to trigger load_data from the data loader, along with a natural language query string. During execution, we first load data from the data loader, index it (for instance with a vector store), and then query it 'on-demand'. All three of these steps happen in a single tool call.

bullet LoadAndSearchToolSpec : The LoadAndSearchToolSpec takes in any existing Tool as input. As a tool spec, it implements to_tool_list , and when that function is called, two tools are returned: a loading tool and then a search tool. The load Tool execution would call the underlying Tool, and the index the output (by default with a vector index). The search Tool execution would take in a query string as input and call the underlying index.

Now that we understand the basics of agents and tools in LlamaIndex, let's see how we can use LlamaIndex to create configurable and manageable workflows!

Model Context Protocol (MCP) in LlamaIndex

LlamaIndex also allows using MCP tools through a ToolSpec on the LlamaHub . You can simply run an MCP server and start using it through the following implementation.

Quick Quiz 1

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/quiz1

Agents Course

Small Quiz (ungraded)

Let’s test your understanding of smolagents with a quick quiz! Remember, testing yourself helps reinforce learning and identify areas that may need review.
This is an optional quiz and it’s not graded.

Q1: What is one of the primary advantages of choosing smolagents over other frameworks?

Which statement best captures a core strength of the smolagents approach?

Q2: In which scenario would you likely benefit most from using smolagents?

Which situation aligns well with what smolagents does best?

Q3: smolagents offers flexibility in model integration. Which statement best reflects its approach?

Choose the most accurate description of how smolagents interoperates with LLMs.

Q4: How does smolagents handle the debate between code-based actions and JSON-based actions?

Which statement correctly characterizes smolagents' philosophy about action formats?

Q5: How does smolagents integrate with the Hugging Face Hub for added benefits?

Which statement accurately describes one of the core advantages of Hub integration?
Congratulations on completing this quiz! ■ If you missed any questions, consider reviewing the Why use smolagents section for a deeper understanding. If you did well, you're ready to explore more advanced topics in smolagents!

Quick Quiz 1

Source: https://huggingface.co/learn/agents-course/unit2/langgraph/quiz1

Agents Course

Test Your Understanding of LangGraph

Let’s test your understanding of LangGraph with a quick quiz! This will help reinforce the key concepts we’ve covered so far.
This is an optional quiz and it’s not graded.

Q1: What is the primary purpose of LangGraph?

Which statement best describes what LangGraph is designed for?

Q2: In the context of the “Control vs Freedom” trade-off, where does LangGraph stand?

Which statement best characterizes LangGraph’s approach to agent design?

Q3: What role does State play in LangGraph?

Choose the most accurate description of State in LangGraph.

Q4: What is a Conditional Edge in LangGraph?

Select the most accurate description.

Q5: How does LangGraph help address the hallucination problem in LLMs?

Choose the best answer.
Congratulations on completing the quiz! ■ If you missed any questions, consider reviewing the previous sections to strengthen your understanding. Next, we’ll explore more advanced features of LangGraph and see how to build more complex agent workflows.

Quick Quiz 1

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/quiz1

Agents Course

Small Quiz (ungraded)

So far we've discussed the key components and tools used in LlamaIndex. It's time to make a short quiz, since testing yourself is the best way to learn and to avoid the illusion of competence . This will help you find where you need to reinforce your knowledge .
This is an optional quiz and it's not graded.

Q1: What is a QueryEngine?

Which of the following best describes a QueryEngine component?

Q2: What is the Purpose of FunctionTools?

Why are FunctionTools important for an Agent?

Q3: What are Toolspecs in LlamaIndex?

What is the main purpose of Toolspecs?

Q4: What is Required to create a tool?

What information must be included when creating a tool?
Congrats on finishing this Quiz ■, if you missed some elements, take time to read again the chapter to reinforce your knowledge. If you pass it, you're ready to dive deeper into building with these components!

Quick Quiz 2

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/quiz2

Agents Course

Small Quiz (ungraded)

It’s time to test your understanding of the Code Agents , Tool Calling Agents , and Tools sections. This quiz is optional and not graded.

Q1: What is the key difference between creating a tool with the @tool decorator versus creating a subclass of Tool in smolagents?

Which statement best describes the distinction between these two approaches for defining tools?

Q2: How does a CodeAgent handle multi-step tasks using the ReAct (Reason + Act) approach?

Which statement correctly describes how the CodeAgent executes a series of steps to solve a task?

Q3: Which of the following is a primary advantage of sharing a tool on the Hugging Face Hub?

Select the best reason why a developer might upload and share their custom tool.

Q4: ToolCallingAgent differs from CodeAgent in how it executes actions. Which statement is

correct?

Choose the option that accurately describes how ToolCallingAgent works.

Q5: What is included in the smolagents default toolbox, and why might you use it?

Which statement best captures the purpose and contents of the default toolbox in smolagents? Congratulations on completing this quiz! ■ If any questions gave you trouble, revisit the Code Agents , Tool Calling Agents , or Tools sections to strengthen your understanding. If you aced it, you’re well on your way to building robust smolagents applications!

Quick Quiz 2

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/quiz2

Agents Course

Quick Self-Check (ungraded)

What?! Another Quiz? We know, we know, … ■ But this short, ungraded quiz is here to help you reinforce key concepts you've just learned .
This quiz covers agent workflows and interactions - essential components for building effective AI agents.

Q1: What is the purpose of AgentWorkflow in LlamaIndex?

Q2: What object is used for keeping track of the state of the workflow?

Q3: Which method should be used if you want an agent to remember previous interactions?

Q4: What is a key feature of Agentic RAG?

Got it? Great! Now let's do a brief recap of the unit!

Conclusion

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/conclusion

Agents Course

Conclusion

Congratulations on finishing the smolagents module of this second Unit ■
You've just mastered the fundamentals of smolagents and you've built your own Agent! Now that you have skills in smolagents , you can now start to create Agents that will solve tasks you're interested about.
In the next module, you're going to learn how to build Agents with LlamaIndex .
Finally, we would love to hear what you think of the course and how we can improve it . If you have some feedback then, please ■ fill this form

Keep Learning, stay awesome ■

Conclusion

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/conclusion

Agents Course

Conclusion

Congratulations on finishing the llama-index module of this second Unit ■
You've just mastered the fundamentals of llama-index and you've seen how to build your own agentic workflows! Now that you have skills in llama-index , you can start to create search engines that will solve tasks you're interested in.
In the next module of the unit, you're going to learn how to build Agents with LangGraph .
Finally, we would love to hear what you think of the course and how we can improve it . If you have some feedback then, please ■ fill this form

Keep Learning, and stay awesome ■

Conclusion

Source: https://huggingface.co/learn/agents-course/unit2/langgraph/conclusion

Agents Course

Conclusion

Congratulations on finishing the LangGraph module of this second Unit! ■
You've now mastered the fundamentals of building structured workflows with LangGraph which you will be able to send to production.
This module is just the beginning of your journey with LangGraph. For more advanced topics, we recommend:

- Exploring the official LangGraph documentation
- Taking the comprehensive Introduction to LangGraph course from LangChain Academy
- Build something yourself !

In the next Unit, you'll now explore real use cases. It's time to leave theory to get into real action !
We would greatly appreciate your thoughts on the course and suggestions for improvement . If you have feedback, please ■ fill this form

Keep Learning, Stay Awesome! ■

Good Sir/Madam! ■■
-Alfred-

Why use SmokeAgents?

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/why_use_smolagents

Agents Course

Why use smolagents

In this module, we will explore the pros and cons of using smolagents , helping you make an informed decision about whether it’s the right framework for your needs.

What is smolagents ?

smolagents is a simple yet powerful framework for building AI agents. It provides LLMs with the agency to interact with the real world, such as searching or generating images.
As we learned in unit 1, AI agents are programs that use LLMs to generate ‘thoughts’ based on ‘observations’ to perform ‘actions’ . Let’s explore how this is implemented in smolagents.

Resources

bullet smolagents Blog - Introduction to smolagents and code interactions

Key Advantages of smolagents

bullet Simplicity: Minimal code complexity and abstractions, to make the framework easy to understand, adopt and extend

bullet Flexible LLM Support: Works with any LLM through integration with Hugging Face tools and external APIs

bullet Code-First Approach: First-class support for Code Agents that write their actions directly in code, removing the need for parsing and simplifying tool calling

bullet HF Hub Integration: Seamless integration with the Hugging Face Hub, allowing the use of Gradio Spaces as tools

When to use smolagents?

With these advantages in mind, when should we use smolagents over other frameworks?
smolagents is ideal when:

- bullet You need a lightweight and minimal solution.
- bullet You want to experiment quickly without complex configurations.
- bullet Your application logic is straightforward.

Code vs. JSON Actions

Unlike other frameworks where agents write actions in JSON, smolagents focuses on tool calls in code , simplifying the execution process. This is because there's no need to parse the JSON in order to build code that calls the tools: the output can be executed directly.
The following diagram illustrates this difference:
To review the difference between Code vs JSON Actions, you can revisit the Actions Section in Unit 1 .

Agent Types in smolagents

Agents in smolagents operate as multi-step agents .
Each MultiStepAgent performs:

- bullet One thought
- bullet One tool call and execution

In addition to using CodeAgent as the primary type of agent, smolagents also supports ToolCallingAgent , which writes tool calls in JSON.
We will explore each agent type in more detail in the following sections.

Model Integration in smolagents

smolagents supports flexible LLM integration, allowing you to use any callable model that meets certain criteria . The framework provides several predefined classes to simplify model connections:

- bullet TransformersModel : Implements a local transformers pipeline for seamless integration.
- bullet HfApiModel : Supports serverless inference calls through Hugging Face's infrastructure , or via a growing number of third-party inference providers .
- bullet LiteLLMModel : Leverages LiteLLM for lightweight model interactions.
- bullet OpenAIServerModel : Connects to any service that offers an OpenAI API interface.
- bullet AzureOpenAIServerModel : Supports integration with any Azure OpenAI deployment.

This flexibility ensures that developers can choose the model and service most suitable for their specific use cases, and allows for easy experimentation.
Now that we understood why and when to use smolagents, let's dive deeper into this powerful library!

Final Quiz

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/final_quiz

Agents Course

Exam Time!

Well done on working through the material on smolagents ! You've already achieved a lot. Now, it's time to put your knowledge to the test with a quiz. ■

Instructions

- The quiz consists of code questions.
- You will be given instructions to complete the code snippets.
- Read the instructions carefully and complete the code snippets accordingly.
- For each question, you will be given the result and some feedback.

■ This quiz is ungraded and uncertified . It's about you understanding the smolagents library and knowing whether you should spend more time on the written material. In the coming units you'll put this knowledge to the test in use cases and projects.
Let's get started!

Quiz ■

You can also access the quiz ■ here

Vision and Browser agents

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/vision_agents

Agents Course

Vision Agents with smolagents

Empowering agents with visual capabilities is crucial for solving tasks that go beyond text processing. Many real-world challenges, such as web browsing or document understanding, require analyzing rich visual content. Fortunately, smolagents provides built-in support for vision-language models (VLMs), enabling agents to process and interpret images effectively.
In this example, imagine Alfred, the butler at Wayne Manor, is tasked with verifying the identities of the guests attending the party. As you can imagine, Alfred may not be familiar with everyone arriving. To help him, we can use an agent that verifies their identity by searching for visual information about their appearance using a VLM. This will allow Alfred to make informed decisions about who can enter. Let's build this example!

Providing Images at the Start of the Agent's Execution

In this approach, images are passed to the agent at the start and stored as task_images alongside the task prompt. The agent then processes these images throughout its execution.
Consider the case where Alfred wants to verify the identities of the superheroes attending the party. He already has a dataset of images from previous parties with the names of the guests. Given a new visitor's image, the agent can compare it with the existing dataset and make a decision about letting them in.
In this case, a guest is trying to enter, and Alfred suspects that this visitor might be The Joker impersonating Wonder Woman. Alfred needs to verify their identity to prevent anyone unwanted from entering.
Let's build the example. First, the images are loaded. In this case, we use images from Wikipedia to keep the example minimal, but imagine the possible use-case!
Now that we have the images, the agent will tell us whether one guest is actually a superhero (Wonder Woman) or a villain (The Joker).
In the case of my run, the output is the following, although it could vary in your case, as we've already discussed:
In this case, the output reveals that the person is impersonating someone else, so we can prevent The Joker from entering the party!

Providing Images with Dynamic Retrieval

The previous approach is valuable and has many potential use cases. However, in situations where the guest is not in the database, we need to explore other ways of identifying them. One possible solution is to dynamically retrieve images and information from external sources, such as browsing the web for details.
In this approach, images are dynamically added to the agent's memory during execution. As we know, agents in smolagents are based on the MultiStepAgent class, which is an abstraction of the ReAct framework. This class operates in a structured cycle where various variables and knowledge are logged at different stages:

- SystemPromptStep: Stores the system prompt.
- TaskStep: Logs the user query and any provided input.
- ActionStep: Captures logs from the agent's actions and results.

This structured approach allows agents to incorporate visual information dynamically and respond adaptively to evolving tasks. Below is the diagram we've already seen, illustrating the dynamic workflow process and how different steps integrate within the agent lifecycle. When browsing, the agent can take screenshots and save them as observation_images in the ActionStep .
Now that we understand the need, let's build our complete example. In this case, Alfred wants full control over the guest verification process, so browsing for details becomes a viable solution. To complete this example, we need a new set of tools for the agent. Additionally, we'll use Selenium and Helium, which are browser automation tools. This will allow us to build an agent that explores the web, searching for details about a potential guest and retrieving verification information. Let's install the tools needed:
We'll need a set of agent tools specifically designed for browsing, such as search_item_ctrl_f , go_back , and close_popups . These tools allow the agent to act like a person navigating the web.
We also need functionality for saving screenshots, as this will be an essential part of what our VLM agent uses to complete the task. This functionality captures the screenshot and saves it in step_log.observations_images = [image.copy()] , allowing the agent to store and process the images dynamically as it navigates.
This function is passed to the agent as step_callback , as it's triggered at the end of each step during the agent's execution. This allows the agent to dynamically capture and store screenshots throughout its process.
Now, we can generate our vision agent for browsing the web, providing it with the tools we created, along with the DuckDuckGoSearchTool to explore the web. This tool will help the agent retrieve necessary information for verifying guests' identities based on visual cues.
With that, Alfred is ready to check the guests' identities and make informed decisions about whether to let them into the party:
You can see that we include helium_instructions as part of the task. This special prompt is aimed to control the navigation of the agent, ensuring that it follows the correct steps while browsing the web.
Let's see how this works in the video below:
This is the final output:
With all of that, we've successfully created our identity verifier for the party! Alfred now has the necessary tools to ensure only the right guests make it through the door. Everything is set to have a good time at Wayne Manor!

Further Reading

- We just gave sight to smolagents - Blog describing the vision agent functionality.
- Web Browser Automation with Agents ■■ - Example for Web browsing using a vision agent.
- Web Browser Vision Agent Example - Example for Web browsing using a vision agent.

Multi-Agent Systems

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/multi_agent_systems

Agents Course

Multi-Agent Systems

Multi-agent systems enable specialized agents to collaborate on complex tasks , improving modularity, scalability, and robustness. Instead of relying on a single agent, tasks are distributed among agents with distinct capabilities.
In smolagents , different agents can be combined to generate Python code, call external tools, perform web searches, and more. By orchestrating these agents, we can create powerful workflows.
A typical setup might include:

- A Manager Agent for task delegation
- A Code Interpreter Agent for code execution
- A Web Search Agent for information retrieval

The diagram below illustrates a simple multi-agent architecture where a Manager Agent coordinates a Code Interpreter Tool and a Web Search Agent , which in turn utilizes tools like the DuckDuckGoSearchTool and VisitWebpageTool to gather relevant information.

Multi-Agent Systems in Action

A multi-agent system consists of multiple specialized agents working together under the coordination of an Orchestrator Agent . This approach enables complex workflows by distributing tasks among agents with distinct roles.
For example, a Multi-Agent RAG system can integrate:

- A Web Agent for browsing the internet.
- A Retriever Agent for fetching information from knowledge bases.
- An Image Generation Agent for producing visuals.

All of these agents operate under an orchestrator that manages task delegation and interaction.

Solving a complex task with a multi-agent hierarchy

The reception is approaching! With your help, Alfred is now nearly finished with the preparations.
But now there’s a problem: the Batmobile has disappeared. Alfred needs to find a replacement, and find it quickly.
Fortunately, a few biopics have been done on Bruce Wayne’s life, so maybe Alfred could get a car left behind on one of the movie sets, and re-engineer it up to modern standards, which certainly would include a full self-driving option.
But this could be anywhere in the filming locations around the world - which could be numerous.
So Alfred wants your help. Could you build an agent able to solve this task?
Let’s build this!
This example needs some additional packages, so let’s install them first:

Resources

- Multi-Agent Systems – Overview of multi-agent systems.
- What is Agentic RAG? – Introduction to Agentic RAG.
- Multi-Agent RAG System ■■■ Recipe – Step-by-step guide to building a multi-agent RAG system.

We first make a tool to get the cargo plane transfer time.

Setting up the agent

For the model provider, we use Together AI, one of the new inference providers on the Hub !
The GoogleSearchTool uses the Serper API to search the web, so this requires either having setup env variable SERPER_API_KEY and passing provider="serpapi" or having SERPER_API_KEY and passing provider=serper .
If you don’t have any Serp API provider setup, you can use DuckDuckGoSearchTool but beware that it has a rate limit.
We can start by creating a simple agent as a baseline to give us a simple report.
In our case, it generates this output:
We could already improve this a bit by throwing in some dedicated planning steps, and adding more prompting.
Planning steps allow the agent to think ahead and plan its next steps, which can be useful for more complex tasks.
In our case, it generates this output:

Thanks to these quick changes, we obtained a much more concise report by simply providing our agent a detailed prompt, and giving it planning capabilities!
The model's context window is quickly filling up. So if we ask our agent to combine the results of detailed search with another, it will be slower and quickly ramp up tokens and costs .
➡■ We need to improve the structure of our system.

✌■ Splitting the task between two agents

Multi-agent structures allow to separate memories between different sub-tasks, with two great benefits:

- Each agent is more focused on its core task, thus more performant
- Separating memories reduces the count of input tokens at each step, thus reducing latency and cost.

Let's create a team with a dedicated web search agent, managed by another agent.
The manager agent should have plotting capabilities to write its final report: so let us give it access to additional imports, including plotly , and geopandas + shapely for spatial plotting.
The manager agent will need to do some mental heavy lifting.
So we give it the stronger model DeepSeek-R1 , and add a planning_interval to the mix.
Let us inspect what this team looks like:
This will generate something like this, helping us understand the structure and relationship between agents and tools used:
I don't know how that went in your run, but in mine, the manager agent skilfully divided tasks given to the web agent in 1. Search for Batman filming locations , then 2. Find supercar factories , before aggregating the lists and plotting the map.
Let's see what the map looks like by inspecting it directly from the agent state:
This will output the map:

Retrieval Agents

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/retrieval_agents

Agents Course

Building Agentic RAG Systems

Retrieval Augmented Generation (RAG) systems combine the capabilities of data retrieval and generation models to provide context-aware responses. For example, a user's query is passed to a search engine, and the retrieved results are given to the model along with the query. The model then generates a response based on the query and retrieved information.
Agentic RAG (Retrieval-Augmented Generation) extends traditional RAG systems by combining autonomous agents with dynamic knowledge retrieval .
While traditional RAG systems use an LLM to answer queries based on retrieved data, agentic RAG enables intelligent control of both retrieval and generation processes , improving efficiency and accuracy.
Traditional RAG systems face key limitations, such as relying on a single retrieval step and focusing on direct semantic similarity with the user's query, which may overlook relevant information.
Agentic RAG addresses these issues by allowing the agent to autonomously formulate search queries, critique retrieved results, and conduct multiple retrieval steps for a more tailored and comprehensive output.

Basic Retrieval with DuckDuckGo

Let's build a simple agent that can search the web using DuckDuckGo. This agent will retrieve information and synthesize responses to answer queries. With Agentic RAG, Alfred's agent can:

- Search for latest superhero party trends
- Refine results to include luxury elements
- Synthesize information into a complete plan

Here's how Alfred's agent can achieve this:
The agent follows this process:

- Analyzes the Request: Alfred's agent identifies the key elements of the query—luxury superhero-themed party planning, with focus on decor, entertainment, and catering.

- Performs Retrieval: The agent leverages DuckDuckGo to search for the most relevant and up-to-date information, ensuring it aligns with Alfred's refined preferences for a luxurious event.
- Synthesizes Information: After gathering the results, the agent processes them into a cohesive, actionable plan for Alfred, covering all aspects of the party.
- Stores for Future Reference: The agent stores the retrieved information for easy access when planning future events, optimizing efficiency in subsequent tasks.

Custom Knowledge Base Tool

For specialized tasks, a custom knowledge base can be invaluable. Let's create a tool that queries a vector database of technical documentation or specialized knowledge. Using semantic search, the agent can find the most relevant information for Alfred's needs.
A vector database stores numerical representations (embeddings) of text or other data, created by machine learning models. It enables semantic search by identifying similar meanings in high-dimensional space.
This approach combines predefined knowledge with semantic search to provide context-aware solutions for event planning. With specialized knowledge access, Alfred can perfect every detail of the party.
In this example, we'll create a tool that retrieves party planning ideas from a custom knowledge base. We'll use a BM25 retriever to search the knowledge base and return the top results, and RecursiveCharacterTextSplitter to split the documents into smaller chunks for more efficient search.
This enhanced agent can:

- First check the documentation for relevant information
- Combine insights from the knowledge base
- Maintain conversation context in memory

Enhanced Retrieval Capabilities

When building agentic RAG systems, the agent can employ sophisticated strategies like:

- Query Reformulation: Instead of using the raw user query, the agent can craft optimized search terms that better match the target documents
- Multi-Step Retrieval: The agent can perform multiple searches, using initial results to inform subsequent queries
- Source Integration: Information can be combined from multiple sources like web search and local documentation
- Result Validation: Retrieved content can be analyzed for relevance and accuracy before being included in responses

Effective agentic RAG systems require careful consideration of several key aspects. The agent should select between available tools based on the query type and context . Memory systems help maintain conversation history and avoid repetitive retrievals. Having fallback strategies ensures the system can still provide value even when primary retrieval methods fail. Additionally, implementing validation steps helps ensure the accuracy and relevance of retrieved information.

Resources

bullet Agentic RAG: turbocharge your RAG with query reformulation and self-query! ■ - Recipe for developing an Agentic RAG system using smolagents.

Writing actions as code snippets or JSON blobs

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/tool_calling_agents

Agents Course

Writing actions as code snippets or JSON blobs

Tool Calling Agents are the second type of agent available in smolagents . Unlike Code Agents that use Python snippets, these agents use the built-in tool-calling capabilities of LLM providers to generate tool calls as JSON structures . This is the standard approach used by OpenAI, Anthropic, and many other providers.
Let’s look at an example. When Alfred wants to search for catering services and party ideas, a CodeAgent would generate and run Python code like this:
A ToolCallingAgent would instead create a JSON structure:
This JSON blob is then used to execute the tool calls.
While smolagents primarily focuses on CodeAgents since they perform better overall , ToolCallingAgents can be effective for simple systems that don’t require variable handling or complex tool calls.

How Do Tool Calling Agents Work?

Tool Calling Agents follow the same multi-step workflow as Code Agents (see the previous section for details).
The key difference is in how they structure their actions : instead of executable code, they generate JSON objects that specify tool names and arguments . The system then parses these instructions to execute the appropriate tools.

Example: Running a Tool Calling Agent

Let’s revisit the previous example where Alfred started party preparations, but this time we’ll use a ToolCallingAgent to highlight the difference. We’ll build an agent that can search the web using

DuckDuckGo, just like in our Code Agent example. The only difference is the agent type - the framework handles everything else:
When you examine the agent's trace, instead of seeing Executing parsed code: , you'll see something like:
The agent generates a structured tool call that the system processes to produce the output, rather than directly executing code like a CodeAgent .
Now that we understand both agent types, we can choose the right one for our needs. Let's continue exploring smolagents to make Alfred's party a success! ■

Resources

bullet ToolCallingAgent documentation - Official documentation for ToolCallingAgent

Building Agents That Use Code

Source: https://huggingface.co/learn/agents-course/unit2/smolagents/code_agents

Agents Course

Building Agents That Use Code

Code agents are the default agent type in smolagents . They generate Python tool calls to perform actions, achieving action representations that are efficient, expressive, and accurate.
Their streamlined approach reduces the number of required actions, simplifies complex operations, and enables reuse of existing code functions. smolagents provides a lightweight framework for building code agents, implemented in approximately 1,000 lines of code.
Graphic from the paper Executable Code Actions Elicit Better LLM Agents

Why Code Agents?

In a multi-step agent process, the LLM writes and executes actions, typically involving external tool calls. Traditional approaches use a JSON format to specify tool names and arguments as strings, which the system must parse to determine which tool to execute .
However, research shows that tool-calling LLMs work more effectively with code directly . This is a core principle of smolagents , as shown in the diagram above from Executable Code Actions Elicit Better LLM Agents .
Writing actions in code rather than JSON offers several key advantages:

- Composability : Easily combine and reuse actions
- Object Management : Work directly with complex structures like images
- Generality : Express any computationally possible task
- Natural for LLMs : High-quality code is already present in LLM training data

How Does a Code Agent Work?

The diagram above illustrates how CodeAgent.run() operates, following the ReAct framework we mentioned in Unit 1. The main abstraction for agents in smolagents is a MultiStepAgent , which serves as the core building block. CodeAgent is a special kind of MultiStepAgent , as we will see in an example

below.
A CodeAgent performs actions through a cycle of steps, with existing variables and knowledge being incorporated into the agent’s context, which is kept in an execution log:

- The system prompt is stored in a SystemPromptStep , and the user query is logged in a TaskStep .
- Then, the following while loop is executed: 2.1 Method agent.write_memory_to_messages() writes the agent’s logs into a list of LLM-readable chat messages . 2.2 These messages are sent to a Model , which generates a completion. 2.3 The completion is parsed to extract the action, which, in our case, should be a code snippet since we’re working with a CodeAgent . 2.4 The action is executed. 2.5 The results are logged into memory in an ActionStep .

At the end of each step, if the agent includes any function calls (in agent.step_callback), they are executed.

Let’s See Some Examples

Alfred is planning a party at the Wayne family mansion and needs your help to ensure everything goes smoothly. To assist him, we’ll apply what we’ve learned about how a multi-step CodeAgent operates.
If you haven’t installed smolagents yet, you can do so by running the following command:
Let’s also login to the Hugging Face Hub to have access to the Serverless Inference API.

Resources

- smolagents Blog - Introduction to smolagents and code interactions
- smolagents: Building Good Agents - Best practices for reliable agents
- Building Effective Agents - Anthropic - Agent design principles
- Sharing runs with OpenTelemetry - Details about how to setup OpenTelemetry for tracking your agents.

Selecting a Playlist for the Party Using smolagents

Music is an essential part of a successful party! Alfred needs some help selecting the playlist. Luckily, smolagents has got us covered! We can build an agent capable of searching the web using DuckDuckGo. To give the agent access to this tool, we include it in the tool list when creating the agent.
For the model, we’ll rely on HfApiModel , which provides access to Hugging Face’s Serverless Inference API . The default model is "Qwen/Qwen2.5-Coder-32B-Instruct" , which is performant and available for fast inference, but you can select any compatible model from the Hub.
Running an agent is quite straightforward:
When you run this example, the output will display a trace of the workflow steps being executed . It will also print the corresponding Python code with the message:
After a few steps, you’ll see the generated playlist that Alfred can use for the party! ■

Using a Custom Tool to Prepare the Menu

Now that we have selected a playlist, we need to organize the menu for the guests. Again, Alfred can take advantage of smolagents to do so. Here, we use the @tool decorator to define a custom function that acts as a tool. We'll cover tool creation in more detail later, so for now, we can simply run the code. As you can see in the example below, we will create a tool using the @tool decorator and include it in the tools list.
The agent will run for a few steps until finding the answer. Precising allowed values in the docstring helps direct agent to occasion argument values which exist and limit hallucinations.
The menu is ready! ■

Using Python Imports Inside the Agent

We have the playlist and menu ready, but we need to check one more crucial detail: preparation time! Alfred needs to calculate when everything would be ready if he started preparing now, in case they need assistance from other superheroes.
smolagents specializes in agents that write and execute Python code snippets, offering sandboxed execution for security. Code execution has strict security measures - imports outside a predefined safe list are blocked by default. However, you can authorize additional imports by passing them as strings in additional_authorized_imports . For more details on secure code execution, see the official guide .
When creating the agent, we'll use additional_authorized_imports to allow for importing the datetime module.
These examples are just the beginning of what you can do with code agents, and we're already starting to see their utility for preparing the party. You can learn more about how to build code agents in the smolagents documentation .
In summary, smolagents specializes in agents that write and execute Python code snippets, offering sandboxed execution for security. It supports both local and API-based language models, making it adaptable to various development environments.

Sharing Our Custom Party Preparator Agent to the Hub

Wouldn't it be amazing to share our very own Alfred agent with the community ? By doing so, anyone can easily download and use the agent directly from the Hub, bringing the ultimate party planner of Gotham to their fingertips! Let's make it happen! ■
The smolagents library makes this possible by allowing you to share a complete agent with the community and download others for immediate use. It's as simple as the following:
To download the agent again, use the code below:
What's also exciting is that shared agents are directly available as Hugging Face Spaces, allowing you to interact with them in real-time. You can explore other agents here .
For example, the AlfredAgent is available here . You can try it out directly below:
You may be wondering—how did Alfred build such an agent using smolagents ? By integrating several tools, he can generate an agent as follows. Don't worry about the tools for now, as we'll have a dedicated section later in this unit to explore that in detail:
As you can see, we've created a CodeAgent with several tools that enhance the agent's functionality, turning it into the ultimate party planner ready to share with the community! ■

Now, it’s your turn: build your very own agent and share it with the community using the knowledge we’ve just learned! ■■■■■■

Inspecting Our Party Preparator Agent with OpenTelemetry and Langfuse ■

As Alfred fine-tunes the Party Preparator Agent, he’s growing weary of debugging its runs. Agents, by nature, are unpredictable and difficult to inspect. But since he aims to build the ultimate Party Preparator Agent and deploy it in production, he needs robust traceability for future monitoring and analysis.

Once again, smolagents comes to the rescue! It embraces the OpenTelemetry standard for instrumenting agent runs, allowing seamless inspection and logging. With the help of Langfuse and the SmolagentsInstrumentor , Alfred can easily track and analyze his agent’s behavior.

Setting it up is straightforward!

First, we need to install the necessary dependencies:

Next, Alfred has already created an account on Langfuse and has his API keys ready. If you haven’t done so yet, you can sign up for Langfuse Cloud here or explore alternatives .

Once you have your API keys, they need to be properly configured as follows:

Finally, Alfred is ready to initialize the SmolagentsInstrumentor and start tracking his agent’s performance.

Alfred is now connected ■! The runs from smolagents are being logged in Langfuse, giving him full visibility into the agent’s behavior. With this setup, he’s ready to revisit previous runs and refine his Party Preparator Agent even further.

Alfred can now access these logs here to review and analyze them.

Meanwhile, the suggested playlist sets the perfect vibe for the party preparations. Cool, right? ■

Now that we have created our first Code Agent, let’s learn how we can create Tool Calling Agents , the second type of agent available in smolagents .

What is LangGraph?

Source: https://huggingface.co/learn/agents-course/unit2/langgraph/when_to_use_langgraph

Agents Course

What is LangGraph ?

LangGraph is a framework developed by LangChain to manage the control flow of applications that integrate an LLM .

Is LangGraph different from LangChain ?

LangChain provides a standard interface to interact with models and other components, useful for retrieval, LLM calls and tools calls. The classes from LangChain might be used in LangGraph, but do not HAVE to be used.
The packages are different and can be used in isolation, but, in the end, all resources you will find online use both packages hand in hand.

When should I use LangGraph ?

How does LangGraph work?

At its core, LangGraph uses a directed graph structure to define the flow of your application:

- Nodes represent individual processing steps (like calling an LLM, using a tool, or making a decision).
- Edges define the possible transitions between steps.
- State is user defined and maintained and passed between nodes during execution. When deciding which node to target next, this is the current state that we look at.

We will explore those fundamental blocks more in the next chapter!

How is it different from regular python? Why do I need LangGraph?

You might wonder: “I could just write regular Python code with if-else statements to handle all these flows, right?”
While technically true, LangGraph offers some advantages over vanilla Python for building complex systems. You could build the same application without LangGraph, but it builds easier tools and abstractions for you.
It includes states, visualization, logging (traces), built-in human-in-the-loop, and more.

Control vs freedom

When designing AI applications, you face a fundamental trade-off between control and freedom :

- Freedom gives your LLM more room to be creative and tackle unexpected problems.
- Control allows you to ensure predictable behavior and maintain guardrails.

Code Agents, like the ones you can encounter in smolagents , are very free. They can call multiple tools in a single action step, create their own tools, etc. However, this behavior can make them less predictable and less controllable than a regular Agent working with JSON!
LangGraph is on the other end of the spectrum, it shines when you need “Control” on the execution of your agent.
LangGraph is particularly valuable when you need Control over your applications . It gives you the tools to build an application that follows a predictable process while still leveraging the power of LLMs.
Put simply, if your application involves a series of steps that need to be orchestrated in a specific way, with decisions being made at each junction point, LangGraph provides the structure you need .
As an example, let’s say we want to build an LLM assistant that can answer some questions over some documents.
Since LLMs understand text the best, before being able to answer the question, you will need to convert other complex modalities (charts, tables) into text. However, that choice depends on the type of document you have!
This is a branching that I chose to represent as follow :
While this branching is deterministic, you can also design branching that are conditioned on the output of an LLM making them undeterministic.
The key scenarios where LangGraph excels include:

- Multi-step reasoning processes that need explicit control on the flow
- Applications requiring persistence of state between steps
- Systems that combine deterministic logic with AI capabilities
- Workflows that need human-in-the-loop interventions
- Complex agent architectures with multiple components working together

In essence, whenever possible, as a human , design a flow of actions based on the output of each action, and decide what to execute next accordingly. In this case, LangGraph is the correct framework for you!
LangGraph is, in my opinion, the most production-ready agent framework on the market.

Document Analysis Graph

Source: https://huggingface.co/learn/agents-course/unit2/langgraph/document_analysis_agent

Agents Course

Document Analysis Graph

Alfred at your service. As Mr. Wayne's trusted butler, I've taken the liberty of documenting how I assist Mr Wayne with his various documentary needs. While he's out attending to his… nighttime activities, I ensure all his paperwork, training schedules, and nutritional plans are properly analyzed and organized. Before leaving, he left a note with his week's training program. I then took the responsibility to come up with a menu for tomorrow's meals.
For future such events, let's create a document analysis system using LangGraph to serve Mr. Wayne's needs. This system can:

- Process images document
- Extract text using vision models (Vision Language Model)
- Perform calculations when needed (to demonstrate normal tools)
- Analyze content and provide concise summaries
- Execute specific instructions related to documents

The Butler's Workflow

The workflow we'll build follows this structured schema:

Setting Up the environment

and imports :

Defining Agent's State

This state is a little more complex than the previous ones we have seen. AnyMessage is a class from Langchain that defines messages, and add_messages is an operator that adds the latest message rather than overwriting it with the latest state.
This is a new concept in LangGraph, where you can add operators in your state to define the way they should interact together.

Preparing Tools

The nodes

The ReAct Pattern: How I Assist Mr. Wayne

Allow me to explain the approach in this agent. The agent follows what's known as the ReAct pattern (Reason-Act-Observe)

- Reason about his documents and requests
- Act by using appropriate tools
- Observe the results
- Repeat as necessary until I've fully addressed his needs

This is a simple implementation of an agent using LangGraph.
We define a tools node with our list of tools. The assistant node is just our model with bound tools. We create a graph with assistant and tools nodes.
We add a tools_condition edge, which routes to End or to tools based on whether the assistant calls a tool.
Now, we add one new step:
We connect the tools node back to the assistant , forming a loop.

- After the assistant node executes, tools_condition checks if the model's output is a tool call.
- If it is a tool call, the flow is directed to the tools node.
- The tools node connects back to assistant .
- This loop continues as long as the model decides to call tools.
- If the model response is not a tool call, the flow is directed to END, terminating the process.

The Butler in Action

Key Takeaways

Should you wish to create your own document analysis butler, here are key considerations:

- Define clear tools for specific document-related tasks
- Create a robust state tracker to maintain context between tool calls
- Consider error handling for tool failures
- Maintain contextual awareness of previous interactions (ensured by the operator add_messages)

With these principles, you too can provide exemplary document analysis service worthy of Wayne Manor.
I trust this explanation has been satisfactory. Now, if you'll excuse me, Master Wayne's cape requires pressing before tonight's activities.

Example 1: Simple Calculations

Here is an example to show a simple use case of an agent using a tool in LangGraph.
The conversation would proceed:

Example 2: Analyzing Master Wayne's Training Documents

When Master Wayne leaves his training and meal notes:
The interaction would proceed:

Building Your First LangGraph

Source: https://huggingface.co/learn/agents-course/unit2/langgraph/first_graph

Agents Course

Building Your First LangGraph

Now that we understand the building blocks, let's put them into practice by building our first functional graph. We'll implement Alfred's email processing system, where he needs to:

- Read incoming emails
- Classify them as spam or legitimate
- Draft a preliminary response for legitimate emails
- Send information to Mr. Wayne when legitimate (printing only)

This example demonstrates how to structure a workflow with LangGraph that involves LLM-based decision-making. While this can't be considered an Agent as no tool is involved, this section focuses more on learning the LangGraph framework than Agents.

Our Workflow

Here's the workflow we'll build:

Setting Up Our Environment

First, let's install the required packages:
Next, let's import the necessary modules:

Step 1: Define Our State

Let's define what information Alfred needs to track during the email processing workflow:

Step 2: Define Our Nodes

Now, let’s create the processing functions that will form our nodes:

Step 3: Define Our Routing Logic

We need a function to determine which path to take after classification:

Step 4: Create the StateGraph and Define Edges

Now we connect everything together:
Notice how we use the special END node provided by LangGraph. This indicates terminal states where the workflow completes.

Step 5: Run the Application

Let’s test our graph with a legitimate email and a spam email:

Step 6: Inspecting Our Mail Sorting Agent with Langfuse ■

As Alfred fine-tunes the Main Sorting Agent, he’s growing weary of debugging its runs. Agents, by nature, are unpredictable and difficult to inspect. But since he aims to build the ultimate Spam Detection Agent and deploy it in production, he needs robust traceability for future monitoring and analysis.
To do this, Alfred can use an observability tool such as Langfuse to trace and monitor the agent.
First, we pip install Langfuse:
Second, we pip install Langchain (LangChain is required because we use LangFuse):
Next, we add the Langfuse API keys and host address as environment variables. You can get your Langfuse credentials by signing up for Langfuse Cloud or self-host Langfuse .
Then, we configure the Langfuse callback_handler and instrument the agent by adding the langfuse_callback to the invocation of the graph: config={"callbacks": [langfuse_handler]} .
Alfred is now connected ■! The runs from LangGraph are being logged in Langfuse, giving him full visibility into the agent’s behavior. With this setup, he’s ready to revisit previous runs and refine his Mail Sorting Agent even further.
Public link to the trace with the legit email

Visualizing Our Graph

LangGraph allows us to visualize our workflow to better understand and debug its structure:
This produces a visual representation showing how our nodes are connected and the conditional paths that can be taken.

What We've Built

We've created a complete email processing workflow that:

- Takes an incoming email
- Uses an LLM to classify it as spam or legitimate
- Handles spam by discarding it
- For legitimate emails, drafts a response and notifies Mr. Hugg

This demonstrates the power of LangGraph to orchestrate complex workflows with LLMs while maintaining a clear, structured flow.

Key Takeaways

- State Management : We defined comprehensive state to track all aspects of email processing
- Node Implementation : We created functional nodes that interact with an LLM
- Conditional Routing : We implemented branching logic based on email classification
- Terminal States : We used the END node to mark completion points in our workflow

What's Next?

In the next section, we'll explore more advanced features of LangGraph, including handling human interaction in the workflow and implementing more complex branching logic based on multiple conditions.

Building Blocks of LangGraph

Source: https://huggingface.co/learn/agents-course/unit2/langgraph/building_blocks

Agents Course

Building Blocks of LangGraph

To build applications with LangGraph, you need to understand its core components. Let’s explore the fundamental building blocks that make up a LangGraph application.
An application in LangGraph starts from an entrypoint , and depending on the execution, the flow may go to one function or another until it reaches the END.

1. State

State is the central concept in LangGraph. It represents all the information that flows through your application.
The state is User defined , hence the fields should carefully be crafted to contain all data needed for decision-making process!

2. Nodes

Nodes are python functions. Each node:

- Takes the state as input
- Performs some operation
- Returns updates to the state

For example, Nodes can contain:

- LLM calls : Generate text or make decisions
- Tool calls : Interact with external systems
- Conditional logic : Determine next steps
- Human intervention : Get input from users

3. Edges

Edges connect nodes and define the possible paths through your graph:
Edges can be:

- Direct : Always go from node A to node B
- Conditional : Choose the next node based on the current state

4. StateGraph

The StateGraph is the container that holds your entire agent workflow:
Which can then be visualized!
But most importantly, invoked:
output :

What’s Next?

In the next section, we’ll put these concepts into practice by building our first graph. This graph lets Alfred take in your e-mails, classify them, and craft a preliminary answer if they are genuine.

Introduction to LlamaHub

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/llama-hub

Agents Course

Introduction to the LlamaHub

LlamaHub is a registry of hundreds of integrations, agents and tools that you can use within LlamaIndex.
We will be using various integrations in this course, so let's first look at the LlamaHub and how it can help us.
Let's see how to find and install the dependencies for the components we need.

Installation

LlamaIndex installation instructions are available as a well-structured overview on LlamaHub . This might be a bit overwhelming at first, but most of the installation commands generally follow an easy-to-remember format :
Let's try to install the dependencies for an LLM and embedding component using the Hugging Face inference API integration .

Usage

Once installed, we can see the usage patterns. You'll notice that the import paths follow the install command! Underneath, we can see an example of the usage of the Hugging Face inference API for an LLM component .
Wonderful, we now know how to find, install and use the integrations for the components we need.
Let's dive deeper into the components and see how we can use them to build our own agents.

Creating Agents Workflows in LlamaIndex

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/workflows

Agents Course

Creating agentic workflows in LlamaIndex

A workflow in LlamaIndex provides a structured way to organize your code into sequential and manageable steps.
Such a workflow is created by defining Steps which are triggered by Events , and themselves emit Events to trigger further steps. Let’s take a look at Alfred showing a LlamaIndex workflow for a RAG task.
Workflows offer several key benefits:

- Clear organization of code into discrete steps
- Event-driven architecture for flexible control flow
- Type-safe communication between steps
- Built-in state management
- Support for both simple and complex agent interactions

As you might have guessed, workflows strike a great balance between the autonomy of agents while maintaining control over the overall workflow.
So, let’s learn how to create a workflow ourselves!

Creating Workflows

Automating workflows with Multi-Agent Workflows

Instead of manual workflow creation, we can use the AgentWorkflow class to create a multi-agent workflow . The AgentWorkflow uses Workflow Agents to allow you to create a system of one or more agents that can collaborate and hand off tasks to each other based on their specialized capabilities. This enables building complex agent systems where different agents handle different aspects of a task. Instead of importing classes from llama_index.core.agent , we will import the agent classes from llama_index.core.agent.workflow . One agent must be designated as the root agent in the AgentWorkflow constructor. When a user message comes in, it is first routed to the root agent.
Each agent can then:

- bullet Handle the request directly using their tools
- bullet Handoff to another agent better suited for the task
- bullet Return a response to the user

Let’s see how to create a multi-agent workflow.
Agent tools can also modify the workflow state we mentioned earlier. Before starting the workflow, we can provide an initial state dict that will be available to all agents. The state is stored in the state key of the workflow context. It will be injected into the state_prompt which augments each new user message. Let’s inject a counter to count function calls by modifying the previous example:
Congratulations! You have now mastered the basics of Agents in LlamaIndex! ■
Let’s continue with one final quiz to solidify your knowledge! ■

Basic Workflow Creation

We can create a single-step workflow by defining a class that inherits from Workflow and decorating your functions with @step . We will also need to add StartEvent and StopEvent , which are special events that are used to indicate the start and end of the workflow.
As you can see, we can now run the workflow by calling w.run() .

Connecting Multiple Steps

To connect multiple steps, we create custom events that carry data between steps. To do so, we need to add an Event that is passed between the steps and transfers the output of the first step to the second step.
The type hinting is important here, as it ensures that the workflow is executed correctly. Let’s complicate things a bit more!

Loops and Branches

The type hinting is the most powerful part of workflows because it allows us to create branches, loops, and joins to facilitate more complex workflows.
Let’s show an example of creating a loop by using the union operator | . In the example below, we see that the LoopEvent is taken as input for the step and can also be returned as output.

Drawing Workflows

We can also draw workflows. Let’s use the draw_all_possible_flows function to draw the workflow. This stores the workflow in an HTML file.
There is one last cool trick that we will cover in the course, which is the ability to add state to the workflow.

State Management

State management is useful when you want to keep track of the state of the workflow, so that every step has access to the same state. We can do this by using the Context type hint on top of a parameter in the step function.
Great! Now you know how to create basic workflows in LlamaIndex!
However, there is another way to create workflows, which relies on the AgentWorkflow class. Let’s take a look at how we can use this to create a multi-agent workflow.

Using Agents in LlamaIndex

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/agents

Agents Course

Using Agents in LlamaIndex

Remember Alfred, our helpful butler agent from earlier? Well, he's about to get an upgrade! Now that we understand the tools available in LlamaIndex, we can give Alfred new capabilities to serve us better. But before we continue, let's remind ourselves what makes an agent like Alfred tick. Back in Unit 1, we learned that:
LlamaIndex supports three main types of reasoning agents:

- Function Calling Agents - These work with AI models that can call specific functions.
- ReAct Agents - These can work with any AI that does chat or text endpoint and deal with complex reasoning tasks.
- Advanced Custom Agents - These use more complex methods to deal with more complex tasks and workflows.

Initialising Agents

To create an agent, we start by providing it with a set of functions/tools that define its capabilities . Let's look at how to create an agent with some basic tools. As of this writing, the agent will automatically use the function calling API (if available), or a standard ReAct agent loop.
LLMs that support a tools/functions API are relatively new, but they provide a powerful way to call tools by avoiding specific prompting and allowing the LLM to create tool calls based on provided schemas.
ReAct agents are also good at complex reasoning tasks and can work with any LLM that has chat or text completion capabilities. They are more verbose, and show the reasoning behind certain actions that they take.
Agents are stateless by default , add remembering past interactions is opt-in using a Context object
This might be useful if you want to use an agent that needs to remember previous interactions, like a chatbot that maintains context across multiple messages or a task manager that needs to track progress over time.
You'll notice that agents in LlamaIndex are async because they use Python's await operator. If you are new to async code in Python, or need a refresher, they have an excellent async guide .
Now we've gotten the basics, let's take a look at how we can use more complex tools in our agents.

Creating RAG Agents with QueryEngineTools

Agentic RAG is a powerful way to use agents to answer questions about your data. We can pass various tools to Alfred to help him answer questions. However, instead of answering the question on top of documents automatically, Alfred can decide to use any other tool or flow to answer the question.
It is easy to wrap QueryEngine as a tool for an agent. When doing so, we need to define a name and description . The LLM will use this information to correctly use the tool. Let’s see how to load in a QueryEngineTool using the QueryEngine we created in the component section .

Creating Multi-agent systems

The AgentWorkflow class also directly supports multi-agent systems. By giving each agent a name and description, the system maintains a single active speaker, with each agent having the ability to hand off to another agent.
By narrowing the scope of each agent, we can help increase their general accuracy when responding to user messages.
Agents in LlamaIndex can also directly be used as tools for other agents, for more complex and custom scenarios.
Now that we understand the basics of agents and tools in LlamaIndex, let’s see how we can use LlamaIndex to create configurable and manageable workflows!

What are Components in LlamaIndex?

Source: https://huggingface.co/learn/agents-course/unit2/llama-index/components

Agents Course

What are components in LlamaIndex?

Remember Alfred, our helpful butler agent from Unit 1? To assist us effectively, Alfred needs to understand our requests and prepare, find and use relevant information to help complete tasks. This is where LlamaIndex's components come in.
While LlamaIndex has many components, we'll focus specifically on the QueryEngine component. Why? Because it can be used as a Retrieval-Augmented Generation (RAG) tool for an agent.
So, what is RAG? LLMs are trained on enormous bodies of data to learn general knowledge. However, they may not be trained on relevant and up-to-date data. RAG solves this problem by finding and retrieving relevant information from your data and giving that to the LLM.
Now, think about how Alfred works:

- You ask Alfred to help plan a dinner party
- Alfred needs to check your calendar, dietary preferences, and past successful menus
- The QueryEngine helps Alfred find this information and use it to plan the dinner party

This makes the QueryEngine a key component for building agentic RAG workflows in LlamaIndex. Just as Alfred needs to search through your household information to be helpful, any agent needs a way to find and understand relevant data. The QueryEngine provides exactly this capability.
Now, let's dive a bit deeper into the components and see how you can combine components to create a RAG pipeline.

Creating a RAG pipeline using components

There are five key stages within RAG, which in turn will be a part of most larger applications you build. These are:

- Loading : this refers to getting your data from where it lives — whether it's text files, PDFs, another website, a database, or an API — into your workflow. LlamaHub provides hundreds of integrations to choose from.

- Indexing : this means creating a data structure that allows for querying the data. For LLMs, this nearly always means creating vector embeddings. Which are numerical representations of the meaning of the data. Indexing can also refer to numerous other metadata strategies to make it easy to accurately find contextually relevant data based on properties.
- Storing : once your data is indexed you will want to store your index, as well as other metadata, to avoid having to re-index it.
- Querying : for any given indexing strategy there are many ways you can utilize LLMs and LlamaIndex data structures to query, including sub-queries, multi-step queries and hybrid strategies.
- Evaluation : a critical step in any flow is checking how effective it is relative to other strategies, or when you make changes. Evaluation provides objective measures of how accurate, faithful and fast your responses to queries are.

Next, let's see how we can reproduce these stages using components.

Loading and embedding documents

As mentioned before, LlamaIndex can work on top of your own data, however, before accessing data, we need to load it. There are three main ways to load data into LlamaIndex:

- SimpleDirectoryReader : A built-in loader for various file types from a local directory.
- LlamaParse : LlamaParse, LlamaIndex's official tool for PDF parsing, available as a managed API.
- LlamaHub : A registry of hundreds of data-loading libraries to ingest data from any source.

The simplest way to load data is with SimpleDirectoryReader . This versatile component can load various file types from a folder and convert them into Document objects that LlamaIndex can work with. Let's see how we can use SimpleDirectoryReader to load data from a folder.
After loading our documents, we need to break them into smaller pieces called Node objects. A Node is just a chunk of text from the original document that's easier for the AI to work with, while it still has references to the original Document object.
The IngestionPipeline helps us create these nodes through two key transformations.

- SentenceSplitter breaks down documents into manageable chunks by splitting them at natural sentence boundaries.
- HuggingFaceEmbedding converts each chunk into numerical embeddings - vector representations that capture the semantic meaning in a way AI can process efficiently.

This process helps us organise our documents in a way that's more useful for searching and analysis.

Storing and indexing documents

After creating our Node objects we need to index them to make them searchable, but before we can do that, we need a place to store our data.
Since we are using an ingestion pipeline, we can directly attach a vector store to the pipeline to populate it. In this case, we will use Chroma to store our documents.
This is where vector embeddings come in - by embedding both the query and nodes in the same vector space, we can find relevant matches. The VectorStoreIndex handles this for us, using the same embedding model we used during ingestion to ensure consistency.

Let’s see how to create this index from our vector store and embeddings:
All information is automatically persisted within the ChromaVectorStore object and the passed directory path.
Great! Now that we can save and load our index easily, let’s explore how to query it in different ways.

Querying a VectorStoreIndex with prompts and LLMs

Before we can query our index, we need to convert it to a query interface. The most common conversion options are:

- as_retriever : For basic document retrieval, returning a list of NodeWithScore objects with similarity scores
- as_query_engine : For single question-answer interactions, returning a written response
- as_chat_engine : For conversational interactions that maintain memory across multiple messages, returning a written response using chat history and indexed context

We’ll focus on the query engine since it is more common for agent-like interactions. We also pass in an LLM to the query engine to use for the response.

Response Processing

Under the hood, the query engine doesn’t only use the LLM to answer the question but also uses a ResponseSynthesizer as a strategy to process the response. Once again, this is fully customisable but there are three main strategies that work well out of the box:

- refine : create and refine an answer by sequentially going through each retrieved text chunk. This makes a separate LLM call per Node/retrieved chunk.
- compact (default): similar to refining but concatenating the chunks beforehand, resulting in fewer LLM calls.
- tree_summarize : create a detailed answer by going through each retrieved text chunk and creating a tree structure of the answer.

The language model won’t always perform in predictable ways, so we can’t be sure that the answer we get is always correct. We can deal with this by evaluating the quality of the answer .

Evaluation and observability

LlamaIndex provides built-in evaluation tools to assess response quality. These evaluators leverage LLMs to analyze responses across different dimensions. Let’s look at the three main evaluators available:

- FaithfulnessEvaluator : Evaluates the faithfulness of the answer by checking if the answer is supported by the context.
- AnswerRelevancyEvaluator : Evaluate the relevance of the answer by checking if the answer is relevant to the question.

bullet CorrectnessEvaluator : Evaluate the correctness of the answer by checking if the answer is correct.

Even without direct evaluation, we can gain insights into how our system is performing through observability. This is especially useful when we are building more complex workflows and want to understand how each component is performing.
We have seen how to use components to create a QueryEngine . Now, let’s see how we can use the QueryEngine as a tool for an agent!

Unit 3

Introduction to Use Case for Agents: RAG

Source: https://huggingface.co/learn/agents-course/unit3/agentic-rag/introduction

Agents Course

Introduction to Use Case for Agentic RAG

In this unit, we will help Alfred, our friendly agent who is hosting the gala, by using Agentic RAG to create a tool that can be used to answer questions about the guests at the gala.
You can choose any of the frameworks discussed in the course for this use case. We provide code samples for each in separate tabs.

A Gala to Remember

Now, it’s time to get our hands dirty with an actual use case. Let’s set the stage!
You decided to host the most extravagant and opulent party of the century. This means lavish feasts, enchanting dancers, renowned DJs, exquisite drinks, a breathtaking fireworks display, and much more.
Alfred, your friendly neighbourhood agent, is getting ready to watch over all of your needs for this party, and Alfred is going to manage everything himself . To do so, he needs to have access to all of the information about the party, including the menu, the guests, the schedule, weather forecasts, and much more!
Not only that, but he also needs to make sure that the party is going to be a success, so he needs to be able to answer any questions about the party during the party , whilst handling unexpected situations that may arise.
He can’t do this alone, so we need to make sure that Alfred has access to all of the information and tools he needs.
First, let’s give him a list of hard requirements for the gala.

The Gala Requirements

A properly educated person in the age of the Renaissance needs to have three main traits. He or she needed to be profound in the knowledge of sports, culture, and science . So, we need to make sure we can impress our guests with our knowledge and provide them with a truly unforgettable gala. However, to avoid any conflicts, there are some topics, like politics and religion, that are to be avoided at a gala. It needs to be a fun party without conflicts related to beliefs and ideals.
According to etiquette, a good host should be aware of guests' backgrounds , including their interests and endeavours. A good host also gossips and shares stories about the guests with one another.
Lastly, we need to make sure that we've got some general knowledge about the weather to ensure we can continuously find a real-time update to ensure perfect timing to launch the fireworks and end the gala with a bang! ■
As you can see, Alfred needs a lot of information to host the gala. Luckily, we can help and prepare Alfred by giving him some Retrieval Augmented Generation (RAG) training!
Let's start by creating the tools that Alfred needs to be able to host the gala!

Building and Integrating Tools for Your Agent

Source: https://huggingface.co/learn/agents-course/unit3/agentic-rag/tools

Agents Course

Building and Integrating Tools for Your Agent

In this section, we'll grant Alfred access to the web, enabling him to find the latest news and global updates. Additionally, he'll have access to weather data and Hugging Face hub model download statistics, so that he can make relevant conversation about fresh topics.

Give Your Agent Access to the Web

Remember that we want Alfred to establish his presence as a true renaissance host, with a deep knowledge of the world.
To do so, we need to make sure that Alfred has access to the latest news and information about the world.
Let's start by creating a web search tool for Alfred!

Creating a Custom Tool for Weather Information to Schedule the Fireworks

The perfect gala would have fireworks over a clear sky, we need to make sure the fireworks are not cancelled due to bad weather.
Let's create a custom tool that can be used to call an external weather API and get the weather information for a given location.

Creating a Hub Stats Tool for Influential AI Builders

In attendance at the gala are the who's who of AI builders. Alfred wants to impress them by discussing their most popular models, datasets, and spaces. We'll create a tool to fetch model statistics from the Hugging Face Hub based on a username.
With the Hub Stats Tool, Alfred can now impress influential AI builders by discussing their most popular models.

Integrating Tools with Alfred

Now that we have all the tools, let's integrate them into Alfred's agent:

Conclusion

By integrating these tools, Alfred is now equipped to handle a variety of tasks, from web searches to weather updates and model statistics. This ensures he remains the most informed and engaging host at the gala.

Conclusion

Source: https://huggingface.co/learn/agents-course/unit3/agentic-rag/conclusion

Agents Course

Conclusion

In this unit, we've learned how to create an agentic RAG system to help Alfred, our friendly neighborhood agent, prepare for and manage an extravagant gala.
The combination of RAG with agentic capabilities demonstrates how powerful AI assistants can become when they have:

- Access to structured knowledge (guest information)
- Ability to retrieve real-time information (web search)
- Domain-specific tools (weather information, Hub stats)
- Memory of past interactions

With these capabilities, Alfred is now well-equipped to be the perfect host, able to answer questions about guests, provide up-to-date information, and ensure the gala runs smoothly—even managing the perfect timing for the fireworks display!

Agentic Retrieval Augmented Generation (RAG)

Source: https://huggingface.co/learn/agents-course/unit3/agentic-rag/agentic-rag

Agents Course

Agentic Retrieval Augmented Generation (RAG)

In this unit, we'll be taking a look at how we can use Agentic RAG to help Alfred prepare for the amazing gala.
LLMs are trained on enormous bodies of data to learn general knowledge. However, the world knowledge model of LLMs may not always be relevant and up-to-date information. RAG solves this problem by finding and retrieving relevant information from your data and forwarding that to the LLM.
Now, think about how Alfred works:

- We've asked Alfred to help plan a gala
- Alfred needs to find the latest news and weather information
- Alfred needs to structure and search the guest information

Just as Alfred needs to search through your household information to be helpful, any agent needs a way to find and understand relevant data. Agentic RAG is a powerful way to use agents to answer questions about your data. We can pass various tools to Alfred to help him answer questions. However, instead of answering the question on top of documents automatically, Alfred can decide to use any other tool or flow to answer the question.
Let's start building our agentic RAG workflow!
First, we'll create a RAG tool to retrieve up-to-date details about the invitees. Next, we'll develop tools for web search, weather updates, and Hugging Face Hub model download statistics. Finally, we'll integrate everything to bring our agentic RAG agent to life!

Creating Your Own Agent

Source: https://huggingface.co/learn/agents-course/unit3/agentic-rag/agent

Agents Course

Creating Your Gala Agent

Now that we've built all the necessary components for Alfred, it's time to bring everything together into a complete agent that can help host our extravagant gala.
In this section, we'll combine the guest information retrieval, web search, weather information, and Hub stats tools into a single powerful agent.

Assembling Alfred: The Complete Agent

Instead of reimplementing all the tools we've created in previous sections, we'll import them from their respective modules which we saved in the tools.py and retriever.py files.
Let's import the necessary libraries and tools from the previous sections:
Your agent is now ready to use!

Using Alfred: End-to-End Examples

Now that Alfred is fully equipped with all the necessary tools, let's see how he can help with various tasks during the gala.

Advanced Features: Conversation Memory

To make Alfred even more helpful during the gala, we can enable conversation memory so he remembers previous interactions:
Notice that none of these three agent approaches directly couple memory with the agent. Is there a specific reason for this design choice ■?

bullet smolagents: Memory is not preserved across different execution runs, you must explicitly state it using reset=False .

- LlamaIndex: Requires explicitly adding a context object for memory management within a run.
- LangGraph: Offers options to retrieve previous messages or utilize a dedicated MemorySaver component.

Conclusion

Congratulations! You've successfully built Alfred, a sophisticated agent equipped with multiple tools to help host the most extravagant gala of the century. Alfred can now:

- Retrieve detailed information about guests
- Check weather conditions for planning outdoor activities
- Provide insights about influential AI builders and their models
- Search the web for the latest information
- Maintain conversation context with memory

With these capabilities, Alfred is ready to ensure your gala is a resounding success, impressing guests with personalized attention and up-to-date information.

Example 1: Finding Guest Information

Let's see how Alfred can help us with our guest information.

Example 2: Checking the Weather for Fireworks

Let's see how Alfred can help us with the weather.

Example 3: Impressing AI Researchers

Let's see how Alfred can help us impress AI researchers.

Example 4: Combining Multiple Tools

Let's see how Alfred can help us prepare for a conversation with Dr. Nikola Tesla.

Creating a RAG Tool for Guest Stories

Source: https://huggingface.co/learn/agents-course/unit3/agentic-rag/invitees

Agents Course

Creating a RAG Tool for Guest Stories

Alfred, your trusted agent, is preparing for the most extravagant gala of the century. To ensure the event runs smoothly, Alfred needs quick access to up-to-date information about each guest. Let's help Alfred by creating a custom Retrieval-Augmented Generation (RAG) tool, powered by our custom dataset.

Why RAG for a Gala?

Imagine Alfred mingling among the guests, needing to recall specific details about each person at a moment's notice. A traditional LLM might struggle with this task because:

bullet The guest list is specific to your event and not in the model's training data

bullet Guest information may change or be updated frequently

bullet Alfred needs to retrieve precise details like email addresses

This is where Retrieval Augmented Generation (RAG) shines! By combining a retrieval system with an LLM, Alfred can access accurate, up-to-date information about your guests on demand.

Setting up our application

In this unit, we'll develop our agent within a HF Space, as a structured Python project. This approach helps us maintain clean, modular code by organizing different functionalities into separate files. Also, this makes for a more realistic use case where you would deploy the application for public use.

Dataset Overview

Our dataset agents-course/unit3-invitees contains the following fields for each guest:

bullet Name : Guest’s full name

bullet Relation : How the guest is related to the host

bullet Description : A brief biography or interesting facts about the guest

bullet Email Address : Contact information for sending invitations or follow-ups

Below is a preview of the dataset:

Building the Guestbook Tool

We’ll create a custom tool that Alfred can use to quickly retrieve guest information during the gala. Let’s break this down into three manageable steps:

bullet Load and prepare the dataset

bullet Create the Retriever Tool

bullet Integrate the Tool with Alfred

Let’s start with loading and preparing the dataset!

Example Interaction

During the gala, a conversation might flow like this:
You: “Alfred, who is that gentleman talking to the ambassador?”
Alfred: quickly searches the guest database “That’s Dr. Nikola Tesla, sir. He’s an old friend from your university days. He’s recently patented a new wireless energy transmission system and would be delighted to discuss it with you. Just remember he’s passionate about pigeons, so that might make for good small talk.”

Taking It Further

Now that Alfred can retrieve guest information, consider how you might enhance this system:

bullet Improve the retriever to use a more sophisticated algorithm like sentence-transformers

bullet Implement a conversation memory so Alfred remembers previous interactions

bullet Combine with web search to get the latest information on unfamiliar guests

bullet Integrate multiple indexes to get more complete information from verified sources

Now Alfred is fully equipped to handle guest inquiries effortlessly, ensuring your gala is remembered as the most sophisticated and delightful event of the century!

Project Structure

bullet tools.py – Provides auxiliary tools for the agent.

- retriever.py – Implements retrieval functions to support knowledge access.
- app.py – Integrates all components into a fully functional agent, which we'll finalize in the last part of this unit.

For a hands-on reference, check out this HF Space , where the Agentic RAG developed in this unit is live. Feel free to clone it and experiment!
You can directly test the agent below:

Step 1: Load and Prepare the Dataset

First, we need to transform our raw guest data into a format that's optimized for retrieval.
In the code above, we:

- Load the dataset
- Convert each guest entry into a Document object with formatted content
- Store the Document objects in a list

This means we've got all of our data nicely available so we can get started with configuring our retrieval.

Step 2: Create the Retriever Tool

Now, let's create a custom tool that Alfred can use to search through our guest information.

Step 3: Integrate the Tool with Alfred

Finally, let's bring everything together by creating our agent and equipping it with our custom tool:

Unit 4

Introduction to Final Test

Source: https://huggingface.co/learn/agents-course/unit4/introduction

Agents Course

Welcome to the final Unit

Welcome to the final unit of the course! ■
So far, you've built a strong foundation in AI Agents , from understanding their components to creating your own. With this knowledge, you're now ready to build powerful agents and stay up-to-date with the latest advancements in this fast-evolving field.
This unit is all about applying what you've learned. It's your final hands-on project , and completing it is your ticket to earning the course certificate .

What's the challenge?

You'll create your own agent and evaluate its performance using a subset of the GAIA benchmark .
To successfully complete the course, your agent needs to score 30% or higher on the benchmark. Achieve that, and you'll earn your Certificate of Completion , officially recognizing your expertise. ■
Additionally, see how you stack up against your peers! A dedicated Student Leaderboard is available for you to submit your scores and see the community's progress.
Sounds exciting? Let's get started! ■

Conclusion of the Course

Source: https://huggingface.co/learn/agents-course/unit4/conclusion

Agents Course

Conclusion

Congratulations on finishing the Agents Course!
Through perseverance and dedication, you've built a solid foundation in the world of AI Agents.
But finishing this course is not the end of your journey . It's just the beginning: don't hesitate to explore the next section where we share curated resources to help you continue learning, including advanced topics like MCPs and beyond.
Thank you for being part of this course. We hope you liked this course as much as we loved writing it .
And don't forget: Keep Learning, Stay Awesome ■

What is GAIA?

Source: https://huggingface.co/learn/agents-course/unit4/what-is-gaia

Agents Course

What is GAIA?

GAIA is a benchmark designed to evaluate AI assistants on real-world tasks that require a combination of core capabilities—such as reasoning, multimodal understanding, web browsing, and proficient tool use.
It was introduced in the paper ” GAIA: A Benchmark for General AI Assistants ” .
The benchmark features 466 carefully curated questions that are conceptually simple for humans , yet remarkably challenging for current AI systems .
To illustrate the gap:

- Humans : ~92% success rate
- GPT-4 with plugins : ~15%
- Deep Research (OpenAI) : 67.36% on the validation set

GAIA highlights the current limitations of AI models and provides a rigorous benchmark to evaluate progress toward truly general-purpose AI assistants.

■ GAIA’s Core Principles

GAIA is carefully designed around the following pillars:

- ■ Real-world difficulty : Tasks require multi-step reasoning, multimodal understanding, and tool interaction.
- ■ Human interpretability : Despite their difficulty for AI, tasks remain conceptually simple and easy to follow for humans.
- ■■ Non-gameability : Correct answers demand full task execution, making brute-forcing ineffective.
- ■ Simplicity of evaluation : Answers are concise, factual, and unambiguous—ideal for benchmarking.

Difficulty Levels

GAIA tasks are organized into three levels of increasing complexity , each testing specific skills:

- Level 1 : Requires less than 5 steps and minimal tool usage.
- Level 2 : Involves more complex reasoning and coordination between multiple tools and 5-10 steps.
- Level 3 : Demands long-term planning and advanced integration of various tools.

Example of a Hard GAIA Question

As you can see, this question challenges AI systems in several ways:

- Requires a structured response format
- Involves multimodal reasoning (e.g., analyzing images)
- Demands multi-hop retrieval of interdependent facts: Identifying the fruits in the painting Discovering which ocean liner was used in The Last Voyage Looking up the breakfast menu from October 1949 for that ship
- Identifying the fruits in the painting
- Discovering which ocean liner was used in The Last Voyage
- Looking up the breakfast menu from October 1949 for that ship
- Needs correct sequencing and high-level planning to solve in the right order

This kind of task highlights where standalone LLMs often fall short, making GAIA an ideal benchmark for agent-based systems that can reason, retrieve, and execute over multiple steps and modalities.

Live Evaluation

To encourage continuous benchmarking, GAIA provides a public leaderboard hosted on Hugging Face , where you can test your models against 300 testing questions .
■ Check out the leaderboard here
Want to dive deeper into GAIA?

- ■ Read the full paper
- ■ Deep Research release post by OpenAI
- ■ Open-source DeepResearch – Freeing our search agents

What Should You Learn Next!

Source: https://huggingface.co/learn/agents-course/unit4/additional-readings

Agents Course

And now? What topics I should learn?

Agentic AI is a rapidly evolving field, and understanding foundational protocols is essential for building intelligent, autonomous systems.
Two important standards you should get familiar with are:

- The Model Context Protocol (MCP)
- The Agent-to-Agent Protocol (A2A)

■ Model Context Protocol (MCP)

The Model Context Protocol (MCP) by Anthropic is an open standard that enables AI models to securely and seamlessly connect with external tools, data sources, and applications , making agents more capable and autonomous.
Think of MCP as a universal adapter , like a USB-C port, that allows AI models to plug into various digital environments without needing custom integration for each one .
MCP is quickly gaining traction across the industry, with major companies like OpenAI and Google beginning to adopt it.
■ Learn more:

- Anthropic’s official announcement and documentation
- MCP on Wikipedia
- Blog on MCP

■ Agent-to-Agent (A2A) Protocol

Google has developed the Agent-to-Agent (A2A) protocol as a complementary counterpart to Anthropic’s Model Context Protocol (MCP).
While MCP connects agents to external tools, A2A connects agents to each other , paving the way for cooperative, multi-agent systems that can work together to solve complex problems.

■ Dive deeper into A2A:

bullet Google’s A2A announcement

Get Your Certificate Of Excellence

Source: https://huggingface.co/learn/agents-course/unit4/get-your-certificate

Agents Course

Claim Your Certificate ■

If you scored above 30%, congratulations! ■ You're now eligible to claim your official certificate. Follow the steps below to receive it:

- Visit the certificate page .
- Sign in with your Hugging Face account using the button provided.
- Enter your full name . This is the name that will appear on your certificate.
- Click "Get My Certificate" to verify your score and download your certificate.

Once you've got your certificate, feel free to:

- Add it to your LinkedIn profile ■■■
- Share it on X , Bluesky , etc. ■

Don't forget to tag @huggingface . We'd be super proud and we'd love to cheer you on! ■

The Final Hands-On

Source: https://huggingface.co/learn/agents-course/unit4/hands-on

Agents Course

Hands-On

Now that you're ready to dive deeper into the creation of your final agent, let's see how you can submit it for review.

The Dataset

The Dataset used in this leaderboard consist of 20 questions extracted from the level 1 questions of the validation set from GAIA. The chosen question were filtered based on the number of tools and steps needed to answer a question.
Based on the current look of the GAIA benchmark, we think that getting you to try to aim for 30% on level 1 question is a fair test.

The process

Now the big question in your mind is probably : "How do I start submitting ?"
For this Unit, we created an API that will allow you to get the questions, and send your answers for scoring. Here is a summary of the routes (see the live documentation for interactive details):

- GET /questions : Retrieve the full list of filtered evaluation questions.
- GET /random-question : Fetch a single random question from the list.
- GET /files/{task_id} : Download a specific file associated with a given task ID.
- POST /submit : Submit agent answers, calculate the score, and update the leaderboard.

The submit function will compare the answer to the ground truth in an EXACT MATCH manner, hence prompt it well ! The GAIA team shared a prompting example for your agent here
■ Make the Template Your Own!
To demonstrate the process of interacting with the API, we've included a basic template as a starting point. Please feel free—and actively encouraged —to change, add to, or completely restructure it! Modify it in any way that best suits your approach and creativity.

In order to submit this templates compute 3 things needed by the API :

- Username: Your Hugging Face username (here obtained via Gradio login), which is used to identify your submission.
- Code Link (agent_code): the URL linking to your Hugging Face Space code (.../tree/main) for verification purposes, so please keep you space public.
- Answers (answers): The list of responses ({"task_id": ..., "submitted_answer": ...}) generated by your Agent for scoring.

Hence we encourage you to start by duplicating this template on your own huggingface profile.
■ Check out the leaderboard here
A friendly note: This leaderboard is meant for fun! We know it’s possible to submit scores without full verification. If we see too many high scores posted without a public link to back them up, we might need to review, adjust, or remove some entries to keep the leaderboard useful. The leaderboard will show the link to your space code-base, since this leaderboard is for students only, please keep your space public if you get a score you’re proud of.