Agents IA à base de LLM avec Langchain

Christophe Guyeux

IUT NFC – UMLP

10 mai 2026

Sommaire

Introduction

Objectifs du cours

- Comprendre ce qu'est un LLM et un agent à base de LLM.
- Maîtriser les briques de Langchain : modèles, prompts, chaînes, outils, agents.
- Construire progressivement un agent capable de raisonner et d'utiliser des outils.
- Tout coder avec des modèles *gratuits* : Mistral (cloud, avec clé) ou Ollama (local).

Qu'est-ce qu'un LLM ?

Un Large Language Model est un réseau de neurones (Transformer) entraîné à prédire le mot suivant à partir d'un contexte.

- Entrée : une séquence de tokens (texte découpé).
- Sortie : une distribution de probabilités sur le prochain token.
- Itéré, cela produit une génération cohérente.

Conséquence pratique

Un LLM ne « sait » rien : il *prédit du texte plausible*. D'où l'intérêt de lui donner du contexte (prompts, outils, mémoire) — c'est tout l'objet du cours.

Qu'est-ce qu'un agent ?

Un agent est un programme qui :

1. reçoit un objectif en langage naturel,
2. décide d'actions à entreprendre (appeler un outil, consulter une base, écrire un fichier...),
3. observe les résultats et itère jusqu'à atteindre l'objectif.

Le LLM joue le rôle de cerveau décisionnel ; les outils sont ses « mains ».

`Objectif` → `LLM raisonne` → `outil` → `observation` → `LLM raisonne` → ...

Pourquoi Langchain ?

Langchain est un framework Python qui standardise l'intégration des LLMs :

- **Agnostique** : Mistral, OpenAI, Anthropic, DeepSeek, Ollama, etc. derrière une interface unique.
- **Composable** (LCEL) : `prompt | model | parser` comme dans un shell Unix.
- **Outillage agent** : `create_agent`, gestion de la mémoire, des outils, du streaming.
- **Écosystème** : `langgraph` (boucles complexes), `langchain-community`, RAG, vecteurs.

Référence officielle : https://docs.langchain.com/oss/python/langchain/agents

Plan du cours

1. Premier prompt + LLM : invocation simple, prompt template, chaîne LCEL.
2. Sorties structurées avec Pydantic.
3. Outils (*tools*) et appels de fonctions.
4. Premier agent avec `create_agent`.
5. Mémoire et streaming.
6. LangGraph pour les boucles agentiques complexes.
7. RAG minimal : retrouver l'information dans des documents.

Prérequis et installation

- Python 3.11+
- uv (https://docs.astral.sh/uv/)
- Optionnel : Ollama (https://ollama.com/) pour exécuter un LLM en local

```
# Installation initiale (chapitre 00)

uv sync
```

Premier prompt avec un LLM

Bibliothèques pour ce chapitre

Au début de chaque cours, on rappelle les paquets à installer :

```
# Bibliothèques à installer (chapitre 01)

uv add langchain langchain-mistralai python-dotenv

# Variante locale sans clé API (Ollama) :
uv add langchain-ollama
ollama pull qwen2.5:3b
```

Configurer la clé d'API

Créez un fichier `.env` à la racine du projet :

```
LLM_BACKEND=mistral
MISTRAL_API_KEY=xxxxxxxxxxxxxxxxxxxxxxxx
```

- Mistral : créez une clé sur https://console.mistral.ai/ (tier *Experiment* gratuit, sans CB).
- Tout en local : passez à Ollama (`LLM_BACKEND=ollama`) après `ollama pull qwen2.5:3b`.

La bibliothèque `python-dotenv` chargera ces variables au lancement du script.

Première invocation d'un LLM

Le pattern minimal : on instancie un modèle, on l'invoque avec une chaîne de caractères, on lit `response.content`.

```python
"""Première invocation : un prompt, un LLM, une réponse."""
import os
from dotenv import load_dotenv

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)

response = llm.invoke("Quelle est la capitale de l'Albanie ? Réponds en une phrase.")
print(response.content)
```

Que se passe-t-il ?

1. `load_dotenv()` : importe la clé d'API depuis `.env`.
2. `ChatMistralAI(...)` : instancie un *chat model* Langchain.
3. `llm.invoke(...)` : envoie un appel HTTP à Mistral, attend la réponse.
4. `response.content` : le texte généré par le modèle.

À retenir

`ChatXxx` est l'interface uniforme : changer de fournisseur ne demande qu'une ligne. Le reste du code reste identique.

Structurer le prompt : message système + utilisateur

Un `ChatPromptTemplate` sépare le rôle (système, utilisateur) du contenu, et permet l'interpolation de variables.

```python
"""ChatPromptTemplate : séparer rôle système et message utilisateur."""
import os
from dotenv import load_dotenv
from langchain_core.prompts import ChatPromptTemplate

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)

prompt = ChatPromptTemplate.from_messages([
    ("system", "Tu es un expert en géographie. Tu réponds en une seule phrase concise."),
    ("human", "{question}"),
])

messages = prompt.invoke({"question": "Quelle est la capitale du Bhoutan ?"})
response = llm.invoke(messages)
print(response.content)
```

Pourquoi un message système ?

- Il cadre le comportement du modèle (ton, format, contraintes).
- Il est *persistent* dans la conversation, contrairement à un préfixe ajouté à la question.
- Il facilite les tests : changer le système sans toucher aux questions utilisateur.

Bonne pratique : un message système *court* et *spécifique* produit de meilleurs résultats qu'un long préambule.

Composer une chaîne avec LCEL

Le Langchain Expression Language compose des étapes avec l'opérateur | (pipe).

```python
"""LCEL : composer prompt | llm | parser avec l'opérateur pipe."""
import os
from dotenv import load_dotenv
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0.7)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0.7)

prompt = ChatPromptTemplate.from_template(
    "Raconte-moi une blague courte sur le sujet : {sujet}"
)

chain = prompt | llm | StrOutputParser()

for sujet in ["les pompiers", "les data scientists"]:
    print(f"--- {sujet} ---")
    print(chain.invoke({"sujet": sujet}))
```

Anatomie de la chaîne

```
prompt | llm | StrOutputParser()
```

- `prompt` : transforme un `dict` en `messages`.
- `llm` : transforme des `messages` en `AIMessage`.
- `StrOutputParser()` : extrait `.content` en `str`.

Chaque maillon est un Runnable. On peut les substituer, les paralléliser (`RunnableParallel`), les router (`RunnableBranch`).

Et si je veux un autre fournisseur ?

Langchain expose la même interface `ChatModel` pour tous les fournisseurs : seul l'import et l'instanciation changent. Quelques exemples :

OpenAI

uv add langchain-openai

```python
# Clé : platform.openai.com/api-keys
import os
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(
    model="gpt-4.1-mini",
    temperature=0,
    api_key=os.environ["OPENAI_API_KEY"],
)
print(llm.invoke("Bonjour !").content)
```

DeepSeek

uv add langchain-deepseek

```python
# Clé : platform.deepseek.com
import os
from langchain_deepseek import
      ChatDeepSeek

llm = ChatDeepSeek(
    model="deepseek-chat",
    temperature=0,
    api_key=os.environ["DEEPSEEK_API_KEY"
        ],
)
print(llm.invoke("Bonjour !").content)
```

Google Gemini

uv add langchain-google-genai

```python
# Clé : aistudio.google.com/app/apikey
import os
from langchain_google_genai import
      ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    temperature=0,
    google_api_key=os.environ["
        GOOGLE_API_KEY"],
)
print(llm.invoke("Bonjour !").content)
```

Le reste du code (`prompt`, `chain`, `StrOutputParser`, agents...) reste identique : c'est la promesse de l'interface unifiée.

Exercices

1. Modifiez `01_invoke_simple.py` pour afficher également `response.usage_metadata` et observer le coût en tokens.
2. Dans `02_chat_prompt.py`, changez le message système pour produire une réponse en anglais, puis en deux phrases.
3. Dans `03_chain.py`, faites varier la `temperature` (0, 0.7, 1.5) et comparez la diversité des blagues.
4. Bascule en local : `LLM_BACKEND=ollama` et relancez les trois scripts. Que constatez-vous sur la qualité et la latence ?

Récapitulatif

- Un `ChatModel` s'invoque avec une string ou des messages.
- Un `ChatPromptTemplate` structure le prompt et permet l'interpolation.
- Le pipe | de LCEL compose prompt, modèle et parser en une seule chaîne.
- Changer de fournisseur (Mistral $\leftrightarrow$ Ollama) ne change qu'une ligne.

Suite

Au prochain cours : forcer le LLM à produire des sorties structurées avec Pydantic.

Sorties structurées avec Pydantic

Bibliothèques pour ce chapitre

```
# Bibliothèques à installer (chapitre 02)

# Aucune nouvelle dépendance : pydantic est fourni par langchain-core.
# Tout est déjà disponible après ‘uv sync‘.
```

pydantic est déjà installé via langchain-core.

Pourquoi structurer la sortie ?

Un `ChatModel` renvoie par défaut du texte libre : pratique pour un humain, fragile pour un programme.

- Comment extraire *automatiquement* l'âge dans « Marie a 34 ans » ?
- Comment garantir qu'un classifieur renvoie bien `"positif" | "négatif" | "neutre"` et rien d'autre ?
- Comment connecter un LLM à une base de données ou une API qui attend du JSON typé ?

Idée

Décrire la forme attendue avec un modèle Pydantic, et laisser Langchain demander au LLM de produire un objet conforme.

Rappel express : Pydantic

`pydantic.BaseModel` est la brique standard de modélisation typée en Python : on déclare des classes avec annotations, et la validation est gratuite.

- Champs typés (`int`, `str`, `list[X]`, `Literal[...]`, modèles imbriqués...).
- `Field(description=...)` : la description sert de prompt pour le LLM.
- Contraintes : `ge`, `le`, `min_length`, `pattern`, etc.
- Validation à l'instanciation : un objet créé est garanti conforme.

Premier modèle structuré

```python
"""Sortie structurée : du texte vers un objet Pydantic typé."""
import os
from dotenv import load_dotenv
from pydantic import BaseModel, Field

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


class Personne(BaseModel):
    """Une personne identifiée dans un texte."""
    nom: str = Field(description="Prénom et nom complet")
    age: int = Field(description="Âge en années")


structured_llm = llm.with_structured_output(Personne)
resultat = structured_llm.invoke("Marie Dupont a 34 ans et habite à Paris.")

print(resultat)
print(f"Type : {type(resultat).__name__}")
print(f"Nom  : {resultat.nom}")
print(f"Age  : {resultat.age}")
```

Que se passe-t-il ?

1. `class Personne(BaseModel)` : on déclare la forme attendue.
2. `llm.with_structured_output(Personne)` : Langchain transforme le modèle en *schema*, l'envoie au LLM (par tool-calling ou JSON mode selon le fournisseur).
3. `.invoke(...)` renvoie directement une instance de `Personne`, déjà validée.

À retenir

Plus de `response.content` à parser à la main. `resultat.nom`, `resultat.age` sont typés et disponibles tout de suite.

Champs imbriqués et listes

Un modèle peut contenir des listes, des sous-modèles, des contraintes (`ge`, `le`, `pattern`...).

```python
"""Extraction d'un objet avec listes et contraintes : critique de film."""
import os
from dotenv import load_dotenv
from pydantic import BaseModel, Field

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


class Critique(BaseModel):
    titre: str = Field(description="Titre du film")
    note: int = Field(description="Note sur 10", ge=0, le=10)
    points_forts: list[str] = Field(description="Aspects positifs cités")


texte = "Inception : effets visuels sublimes, scénario captivant. Je mets 9/10."

structured_llm = llm.with_structured_output(Critique)
critique = structured_llm.invoke(f"Extrais une critique de :\n{texte}")

print(f"{critique.titre} : {critique.note}/10")
for p in critique.points_forts:
    print(f"  - {p}")
```

Anatomie de l'extraction

- `titre: str` et `note: int` : champs scalaires typés.
- `ge=0, le=10` : contraintes de plage. Le LLM voit le schéma et s'y conforme ; Pydantic vérifie à la réception.
- `points_forts: list[str]` : un champ liste, parfait pour les énumérations extraites du texte.

Bonne pratique

Les `description=...` de chaque `Field` se retrouvent telles quelles dans le prompt envoyé au modèle. Soigner ces descriptions = soigner le prompt.

Classification avec `Literal`

`typing.Literal[...]` contraint un champ à un ensemble fini de valeurs. Idéal pour la classification.

```python
"""Classification : Literal contraint le LLM à choisir parmi des valeurs fixes."""
import os
from typing import Literal
from dotenv import load_dotenv
from pydantic import BaseModel, Field

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


class Sentiment(BaseModel):
    polarite: Literal["positif", "negatif", "neutre"]
    confiance: int = Field(description="Confiance 0-100", ge=0, le=100)


structured_llm = llm.with_structured_output(Sentiment)

for m in [
    "Ce film m'a touché, je le recommande !",
    "Service catastrophique, je ne reviendrai pas.",
    "Le rapport est arrivé hier matin.",
]:
    s = structured_llm.invoke(m)
    print(f"{m[:38]:38s} -> {s.polarite:8s} ({s.confiance}%)")
```

Sortie typée = sortie sûre

- Le LLM *ne peut pas* renvoyer une polarité hors de la liste : Pydantic refuserait l'objet.
- En pratique, le schéma envoyé au modèle inclut l'énumération : il choisit dans le menu, plutôt que de halluciner une valeur.
- Combiné à un `int` avec `ge`/`le`, on obtient un classifieur autodescriptif et typé en quelques lignes.

Alternative explicite : `PydanticOutputParser`

`with_structured_output` est magique : il choisit la bonne stratégie selon le fournisseur. Si on veut tout contrôler (instructions de format dans le prompt, modèles sans tool-calling…), on chaîne un `PydanticOutputParser`.

```python
"""Alternative explicite : PydanticOutputParser dans une chaîne LCEL."""
import os
from dotenv import load_dotenv
from pydantic import BaseModel, Field
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import PydanticOutputParser

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


class Recette(BaseModel):
    plat: str = Field(description="Nom du plat")
    ingredients: list[str] = Field(description="Ingrédients principaux")


parser = PydanticOutputParser(pydantic_object=Recette)
prompt = ChatPromptTemplate.from_messages([
    ("system", "Tu réponds uniquement au format demandé."),
    ("human", "{question}\n\n{format_instructions}"),
]).partial(format_instructions=parser.get_format_instructions())

chain = prompt | llm | parser
recette = chain.invoke({"question": "Recette d'une omelette aux fines herbes."})

print(f"Plat : {recette.plat}")
for i in recette.ingredients:
    print(f"  - {i}")
```

Quand utiliser quoi ?

- `with_structured_output(Modèle)` : choix par défaut, concis.
- `PydanticOutputParser` dans une LCEL chain : quand on veut visualiser/maîtriser les `format_instructions`, ou cibler un modèle qui n'expose pas de tool-calling.
- Pour les sorties très simples : `StrOutputParser`, `JsonOutputParser`, ou rien du tout.

Exercices

1. Modifiez `01_pydantic_basics.py` pour ajouter un champ `ville: str | None`. Testez sur une phrase qui ne mentionne aucune ville.
2. Dans `02_extraction.py`, ajoutez un sous-modèle `Acteur` avec `nom` et `role`, et un champ `acteurs: list[Acteur]`.
3. Dans `03_classification.py`, élargissez `Literal` à cinq valeurs (`très_positif`, `positif`, `neutre`, `négatif`, `très_négatif`). Que constatez-vous sur la qualité ?
4. Comparez les `format_instructions` générés par `PydanticOutputParser.get_format_instructions()` pour deux modèles différents.

Récapitulatif

- Un modèle `Pydantic` décrit la forme attendue de la réponse.
- `llm.with_structured_output(Modèle)` renvoie une instance typée et validée.
- Les `description=...` des champs sont injectées dans le prompt : c'est là que se joue la qualité de l'extraction.
- `Literal` et les contraintes (`ge`, `le`, `pattern`) verrouillent le format.

Suite

Au prochain cours : donner au LLM la possibilité d'appeler des outils (fonctions Python, APIs, recherche web).

Outils et appels de fonctions

Bibliothèques pour ce chapitre

```
# Bibliothèques à installer (chapitre 03)

# Aucune nouvelle dépendance : `@tool` et `bind_tools` viennent de langchain-core.
# Tout est déjà disponible après `uv sync`.
```

`@tool` et `bind_tools` viennent de `langchain-core`.

Pourquoi des outils ?

Un LLM seul a des angles morts bien connus :

- Pas d'arithmétique fiable au-delà de quelques chiffres.
- Pas de date du jour, pas d'heure, pas d'accès au système.
- Pas de recherche web, pas de base de données, pas d'API.
- Connaissances figées au moment de l'entraînement.

Solution

Lui exposer des fonctions Python qu'il peut décider d'appeler. Le LLM produit l'appel ; nous l'exécutons ; nous lui renvoyons le résultat.

Le décorateur `@tool`

Une fonction Python annotée + une docstring suffisent à devenir un outil.

- Les annotations de type décrivent les arguments au LLM.
- La docstring explique *quand* l'utiliser.
- Le nom de la fonction devient le nom de l'outil.

Bonne pratique

Une bonne docstring d'outil = une description claire *de l'usage*, pas une explication de l'implémentation.

Premier outil : voir la demande d'appel

```python
"""Premier outil : exposer une fonction Python au LLM."""
import os
from dotenv import load_dotenv
from langchain_core.tools import tool

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


@tool
def multiplier(a: int, b: int) -> int:
    """Multiplie deux entiers."""
    return a * b


llm_avec_outil = llm.bind_tools([multiplier])
reponse = llm_avec_outil.invoke("Combien font 137 fois 89 ?")

print("Contenu :", reponse.content or "(vide : le modèle a appelé un outil)")
print("tool_calls :", reponse.tool_calls)
```

Que se passe-t-il ?

1. `llm.bind_tools([multiplier])` : le LLM sait désormais qu'il a accès à `multiplier(a:int, b:int)`.
2. Sur « Combien font 137 fois 89 ? », le modèle ne calcule pas : il demande d'appeler `multiplier(a=137, b=89)`.
3. `reponse.content` est vide ; `reponse.tool_calls` contient la demande d'appel.

À retenir

`bind_tools` ne déclenche aucune exécution : le LLM décide juste *quoi* appeler. C'est à nous d'exécuter et de lui rendre la main.

Cycle complet : appel, exécution, réponse

```python
"""Cycle complet : LLM demande l'outil, on l'exécute, on renvoie le résultat."""
import os
from dotenv import load_dotenv
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage, ToolMessage

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


@tool
def multiplier(a: int, b: int) -> int:
    """Multiplie deux entiers."""
    return a * b


outils = {"multiplier": multiplier}
llm_avec_outils = llm.bind_tools(list(outils.values()))

messages = [HumanMessage("Combien font 137 fois 89 ?")]
ai_msg = llm_avec_outils.invoke(messages)
messages.append(ai_msg)

for call in ai_msg.tool_calls:
    resultat = outils[call["name"]].invoke(call["args"])
    messages.append(ToolMessage(content=str(resultat), tool_call_id=call["id"]))

reponse_finale = llm_avec_outils.invoke(messages)
print(reponse_finale.content)
```

Anatomie du cycle

1. `HumanMessage` : la question initiale.
2. Premier `invoke` : le LLM renvoie un `AIMessage` contenant des `tool_calls`.
3. Pour chaque appel, on exécute l'outil et on ajoute un `ToolMessage` avec le résultat (lié par `tool_call_id`).
4. Second `invoke` : le LLM voit le résultat et formule la réponse en langage naturel.

Ce cycle est le cœur d'un agent. Au prochain chapitre, `create_agent` l'automatisera, mais le fonctionnement reste exactement celui-ci.

Plusieurs outils : le LLM choisit

```python
"""Plusieurs outils : le LLM choisit le bon selon la question."""
import os
from datetime import date
from dotenv import load_dotenv
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage, ToolMessage

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


@tool
def multiplier(a: int, b: int) -> int:
    """Multiplie deux entiers."""
    return a * b


@tool
def date_du_jour() -> str:
    """Renvoie la date du jour au format ISO (AAAA-MM-JJ)."""
    return date.today().isoformat()


outils = {"multiplier": multiplier, "date_du_jour": date_du_jour}
llm_avec_outils = llm.bind_tools(list(outils.values()))

for question in ["Quelle est la date du jour ?", "Calcule 42 fois 17."]:
    messages = [HumanMessage(question)]
    ai_msg = llm_avec_outils.invoke(messages)
    messages.append(ai_msg)
    for call in ai_msg.tool_calls:
        res = outils[call["name"]].invoke(call["args"])
        messages.append(ToolMessage(content=str(res), tool_call_id=call["id"]))
    reponse = llm_avec_outils.invoke(messages)
    print(f"Q: {question}")
    print(f"R: {reponse.content}\n")
```

Comment le modèle choisit-il ?

- Il reçoit la liste complète des outils avec leurs signatures et docstrings.
- Il choisit en fonction du contenu de la question : « date du jour » déclenche `date_du_jour`, « calcule » déclenche `multiplier`.
- Il peut aussi décider de ne rien appeler et répondre directement.

Bonne pratique

Donner peu d'outils, bien décrits, vaut mieux qu'un catalogue mal documenté. Le LLM se trompe d'autant plus qu'il y a d'options ambiguës.

Appels parallèles

Pour une question portant sur plusieurs sujets indépendants, le LLM peut renvoyer plusieurs `tool_calls` en une seule réponse.

```python
"""Appels parallèles : le LLM peut demander plusieurs outils en une seule réponse."""
import os
from dotenv import load_dotenv
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage, ToolMessage

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


@tool
def temperature(ville: str) -> int:
    """Renvoie la température en Celsius pour une ville (donnée fictive)."""
    return {"Paris": 18, "Tokyo": 24, "Lima": 22}.get(ville, 20)


llm_avec_outil = llm.bind_tools([temperature])
messages = [HumanMessage("Quelles sont les températures à Paris, Tokyo et Lima ?")]
ai_msg = llm_avec_outil.invoke(messages)
messages.append(ai_msg)

print(f"Le modèle a fait {len(ai_msg.tool_calls)} appel(s) :")
for call in ai_msg.tool_calls:
    res = temperature.invoke(call["args"])
    print(f"  - {call['name']}({call['args']}) = {res}")
    messages.append(ToolMessage(content=str(res), tool_call_id=call["id"]))

reponse = llm_avec_outil.invoke(messages)
print(f"\nRéponse : {reponse.content}")
```

Pourquoi c'est précieux

- Trois appels `temperature` en parallèle = une seule itération LLM au lieu de trois.
- Latence et coût divisés par autant.
- Tous les fournisseurs majeurs supportent ces appels parallèles (Mistral, OpenAI, Anthropic, Ollama avec modèles récents).

Exercices

1. Ajoutez un outil `additionner(a:int, b:int) -> int` et testez « Combien font 12 + 7 puis multiplié par 3 ? ». Combien d'allers-retours observe-t-on ?
2. Modifiez `04_appels_paralleles.py` pour exécuter les outils en parallèle avec `concurrent.futures`.
3. Faites un outil `rechercher_wikipedia(sujet:str) -> str` (utilisez la lib `wikipedia`). Demandez « Qui est Alan Turing ? ».
4. Que se passe-t-il si vous donnez deux outils aux noms quasi identiques (`temperature_celsius` et `temperature_kelvin`) ?

Récapitulatif

- Le décorateur `@tool` transforme une fonction Python en outil exposable au LLM.
- `llm.bind_tools([...])` liste les outils disponibles ; aucune exécution.
- Le cycle appel → exécution → réponse se gère manuellement avec `HumanMessage`, `AIMessage`, `ToolMessage`.
- Le LLM peut demander plusieurs appels en une fois (parallèles).

Suite

Au prochain cours : `create_agent` automatise la boucle et nous fait passer de « LLM avec outils » à agent au sens propre.

Premier agent avec `create_agent`

Bibliothèques pour ce chapitre

```
# Bibliothèques à installer (chapitre 04)

# Aucune nouvelle dépendance : ‘create_agent‘ vient de langchain.agents (langchain >= 1.0).
# Tout est déjà disponible après ‘uv sync‘.
```

`create_agent` vient de `langchain.agents` (`langchain >= 1.0`).

Du chapitre 3 à l'agent

Au chapitre 3, on a écrit à la main la boucle :

1. Premier `invoke` : le LLM produit des `tool_calls`.
2. Pour chaque appel, on exécute l'outil et on construit un `ToolMessage`.
3. Second `invoke` : le LLM voit les résultats et formule la réponse.

Limite de la boucle manuelle

Si le LLM a besoin de deux étapes successives (ex. : la température puis une soustraction), il faut écrire la boucle *tant que* `tool_calls` n'est pas vide. C'est ce que `create_agent` fait pour nous.

Un agent, c'est une boucle automatisée

`LLM` → `tool_calls ?` → `exécuter` → `rappeler le LLM` → ...
...→ `pas de tool_call` → `réponse finale.`

- L'agent boucle tant que le LLM demande des outils.
- Il s'arrête quand le LLM répond directement (sans `tool_calls`).
- Il garde tout l'historique : chaque tour voit *tous* les messages précédents.

Premier agent

```python
"""Premier agent : create_agent automatise toute la boucle outil/exec/réponse."""
import os
from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_core.tools import tool

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


@tool
def multiplier(a: int, b: int) -> int:
    """Multiplie deux entiers."""
    return a * b


agent = create_agent(model=llm, tools=[multiplier])

resultat = agent.invoke(
    {"messages": [{"role": "user", "content": "Combien font 137 fois 89 ?"}]}
)
print(resultat["messages"][-1].content)
```

Que se passe-t-il sous le capot ?

1. `create_agent(model=llm, tools=[multiplier])` construit un graphe (LangGraph) à deux nœuds : *model* et *tools*.
2. `agent.invoke({"messages": [...]})` démarre la boucle : alternance model → tools → model → ...
3. Le résultat est un `dict` avec une clé `messages` contenant tout l'historique.
4. `resultat["messages"][-1]` est le dernier `AIMessage` : la réponse finale.

À retenir

Toute la boucle manuelle du chapitre 3 tient maintenant en trois lignes.

Donner un rôle : system_prompt

Le system_prompt est injecté en tête de chaque conversation. Idéal pour cadrer le ton, le format, ou rappeler quels outils privilégier.

```python
"""system_prompt : donner un rôle ou des consignes globales à l'agent."""
import os
from datetime import date
from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_core.tools import tool

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


@tool
def date_du_jour() -> str:
    """Renvoie la date du jour au format ISO (AAAA-MM-JJ)."""
    return date.today().isoformat()


agent = create_agent(
    model=llm,
    tools=[date_du_jour],
    system_prompt=(
        "Tu es un assistant concis qui répond en deux phrases maximum. "
        "Pour toute question impliquant la date, utilise l'outil date_du_jour."
    ),
)

resultat = agent.invoke(
    {"messages": [{"role": "user", "content": "On est quel jour ?"}]}
)
print(resultat["messages"][-1].content)
```

Quand utiliser `system_prompt` ?

- Persona / ton (« réponds comme un avocat », « sois concis »).
- Format imposé (« toujours en JSON », « en deux phrases maximum »).
- Politique d'usage des outils (« n'invente jamais une date sans appeler date_du_jour »).
- Garde-fous (« si la question sort de ton domaine, refuse poliment »).

Bonne pratique

Plus le `system_prompt` est *spécifique*, plus l'agent est *prévisible*. Mais attention : un système trop long alourdit chaque appel.

Voir la trajectoire avec `agent.stream`

```python
"""Voir la trajectoire : agent.stream affiche chaque message de la boucle."""
import os
from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_core.tools import tool
from langchain_core.messages import HumanMessage, AIMessage, ToolMessage

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


@tool
def multiplier(a: int, b: int) -> int:
    """Multiplie deux entiers."""
    return a * b


agent = create_agent(model=llm, tools=[multiplier])

for etat in agent.stream(
    {"messages": [{"role": "user", "content": "Combien font 23 fois 17 ?"}]},
    stream_mode="values",
):
    msg = etat["messages"][-1]
    if isinstance(msg, HumanMessage):
        print(f"USER  : {msg.content}")
    elif isinstance(msg, AIMessage):
        if msg.tool_calls:
            calls = [f"{tc['name']}({tc['args']})" for tc in msg.tool_calls]
            print(f"AGENT : appelle {calls}")
        else:
            print(f"AGENT : {msg.content}")
    elif isinstance(msg, ToolMessage):
        print(f"OUTIL : {msg.content}")
```

Lecture d'une trajectoire

Une trajectoire typique avec un seul outil :

1. `HumanMessage` : la question de l'utilisateur.
2. `AIMessage` avec `tool_calls` non vide : l'agent demande l'outil.
3. `ToolMessage` : le résultat brut de l'exécution.
4. `AIMessage` sans `tool_calls` : la réponse finale.

Plus la question est composée, plus la trajectoire s'allonge (étape 2 et 3 répétées).

`stream_mode="values"` renvoie l'état complet à chaque pas. Pratique pour debug ; on peut aussi capturer les `tool_calls` pour journalisation.

Multi-outils enchaînés

L'agent décide tout seul dans quel ordre appeler les outils, et combien de fois.

```python
"""Agent multi-outils : enchaîner plusieurs appels pour une question composée."""
import os
from datetime import date
from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_core.tools import tool

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


@tool
def temperature(ville: str) -> int:
    """Renvoie la température en Celsius pour une ville (donnée fictive)."""
    return {"Paris": 18, "Tokyo": 24, "Lima": 22}.get(ville, 20)


@tool
def soustraire(a: int, b: int) -> int:
    """Renvoie a - b."""
    return a - b


@tool
def date_du_jour() -> str:
    """Renvoie la date du jour au format ISO."""
    return date.today().isoformat()


agent = create_agent(model=llm, tools=[temperature, soustraire, date_du_jour])

question = (
    "Quel est l'écart de température entre Tokyo et Paris aujourd'hui, "
    "et quelle est la date du jour ?"
)
resultat = agent.invoke({"messages": [{"role": "user", "content": question}]})
print(resultat["messages"][-1].content)
```

Limites et bonnes pratiques

- Le LLM peut boucler : prévoir une limite (`recursion_limit` dans `config`).
- Plus il y a d'outils, plus le LLM se trompe : viser moins mais mieux décrit.
- Coût : chaque tour de boucle = un appel LLM. Mistral « small » reste bon marché ; pour des modèles premium, surveiller.
- Sécurité : les outils sont du code Python qui s'exécute. Ne jamais exposer un `eval` ou un accès filesystem à un agent ouvert au public.

Exercices

1. Comparez ligne à ligne `02_executer_outil.py` (chapitre 3) et `01_premier_agent.py` : que disparaît-il ?
2. Modifiez `02_system_prompt.py` pour interdire au modèle de répondre sur la météo. Vérifiez en lui posant la question.
3. Dans `03_trajectoire.py`, ajoutez un compteur de tours et arrêtez la boucle après 5 messages, même si l'agent n'a pas terminé.
4. Reprenez l'exercice 3 du chapitre 3 (Wikipedia) et écrivez l'équivalent avec `create_agent`. Comparez la concision du code.

Récapitulatif

- `create_agent(model, tools)` construit un agent qui boucle automatiquement `LLM` $\leftrightarrow$ `tools`.
- `system_prompt` cadre le comportement global de l'agent.
- `agent.stream(..., stream_mode="values")` expose la trajectoire pour debug ou journalisation.
- L'agent enchaîne plusieurs outils sans qu'on ait à scripter l'ordre.

Suite

Au prochain cours : doter l'agent d'une mémoire entre conversations, et streamer ses tokens en temps réel.

Mémoire et streaming

Bibliothèques pour ce chapitre

```
# Bibliothèques à installer (chapitre 05)

# ‘langgraph‘ est installé via langchain (transitive dependency).
# Si besoin explicite :
uv add langgraph
```

Pourquoi mémoire et streaming ?

Jusqu'ici, chaque `agent.invoke` est indépendant : aucune mémoire d'un appel à l'autre.

- Pas de « Comment je m'appelle ? » après un « Je m'appelle Marie » dans un appel précédent.
- Pas de chatbot multi-tour, pas d'assistant qui apprend de l'utilisateur dans la session.

Et la sortie arrive d'un seul bloc après plusieurs secondes :

- L'utilisateur attend dans le vide.
- Pour des longues réponses (RAG, rapports...), c'est inacceptable en UX.

Solution : un checkpointer pour la mémoire, un stream pour les tokens.

Le checkpointer : sauvegarder l'état entre appels

- Un `checkpointer` est un magasin clé/valeur où l'agent sérialise tout son état après chaque pas.
- La clé est le `thread_id` : un identifiant de conversation.
- Au tour suivant, l'agent recharge l'état du thread, ajoute le nouveau message et continue.

Trois implémentations courantes :

- `InMemorySaver` (langgraph) : en RAM, disparaît à l'arrêt du process. Idéal pour développer.
- `PostgresSaver`, `SqliteSaver` : persistance disque, durable.
- `RedisSaver` : partage entre instances, latence faible.

Mémoire entre tours

```python
"""Mémoire entre tours : checkpointer + thread_id."""
import os
from dotenv import load_dotenv
from langchain.agents import create_agent
from langgraph.checkpoint.memory import InMemorySaver

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


agent = create_agent(model=llm, tools=[], checkpointer=InMemorySaver())
config = {"configurable": {"thread_id": "session-1"}}

r1 = agent.invoke({"messages": [{"role": "user", "content": "Bonjour, je m'appelle Marie."}]}, config)
print("Tour 1 :", r1["messages"][-1].content)

r2 = agent.invoke({"messages": [{"role": "user", "content": "Comment je m'appelle ?"}]}, config)
print("Tour 2 :", r2["messages"][-1].content)
```

Anatomie

1. `create_agent(..., checkpointer=InMemorySaver())` : l'agent saura sauvegarder son état.
2. Le `config={"configurable": {"thread_id": "session-1"}}` identifie la conversation.
3. Au premier `invoke`, l'historique contient `[Marie]`. Au second, l'agent voit *tout* l'historique et peut répondre « Vous vous appelez Marie ».

À retenir

Sans `thread_id`, le checkpointer ne peut rien sauvegarder. Avec un `thread_id` fixe, on construit une conversation continue.

Threads séparés = conversations isolées

```python
"""Deux threads = deux conversations isolées."""
import os
from dotenv import load_dotenv
from langchain.agents import create_agent
from langgraph.checkpoint.memory import InMemorySaver

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


agent = create_agent(model=llm, tools=[], checkpointer=InMemorySaver())

agent.invoke(
    {"messages": [{"role": "user", "content": "Je m'appelle Bob."}]},
    {"configurable": {"thread_id": "alice"}},
)
agent.invoke(
    {"messages": [{"role": "user", "content": "Je m'appelle Alice."}]},
    {"configurable": {"thread_id": "alice"}},
)

# Sur le thread "bob", l'agent ne sait rien d'Alice ni de Bob.
r = agent.invoke(
    {"messages": [{"role": "user", "content": "Tu connais mon prénom ?"}]},
    {"configurable": {"thread_id": "bob"}},
)
print("Thread bob :", r["messages"][-1].content)
```

Pourquoi des threads ?

- Un même agent dessert plusieurs utilisateurs : un `thread_id` par session web ou par client API.
- Aucune fuite d'information d'un thread à l'autre : c'est l'isolation par défaut.
- Un même utilisateur peut avoir plusieurs conversations parallèles (ex : « conversation projet A », « conversation projet B »).

Bonne pratique

Choisir un `thread_id` stable et secret côté serveur (UUID, hash de session...). Ne jamais le passer en clair côté client.

Streaming token par token

Au lieu d'attendre la réponse complète, on consomme les tokens au fur et à mesure.

```python
"""Streaming token par token : stream_mode="messages"."""
import os
from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_core.messages import AIMessageChunk

load_dotenv()
backend = os.getenv("LLM_BACKEND", "mistral")

if backend == "mistral":
    from langchain_mistralai import ChatMistralAI
    llm = ChatMistralAI(model="mistral-small-latest", temperature=0)
else:
    from langchain_ollama import ChatOllama
    llm = ChatOllama(model=os.getenv("OLLAMA_MODEL", "qwen2.5:3b"), temperature=0)


agent = create_agent(model=llm, tools=[])

for chunk in agent.stream(
    {"messages": [{"role": "user", "content": "Donne-moi trois conseils pour apprendre Python."}]},
    stream_mode="messages",
    version="v2",
):
    if chunk["type"] == "messages":
        token, _ = chunk["data"]
        if isinstance(token, AIMessageChunk) and token.text:
            print(token.text, end="", flush=True)
print()
```

Modes de streaming

`agent.stream(..., stream_mode=X, version="v2")` accepte plusieurs modes :

- `"messages"` : tokens individuels (`AIMessageChunk`). Idéal pour afficher en direct.
- `"values"` : état complet à chaque pas (*vu au chapitre 4*). Idéal pour debug.
- `"updates"` : seulement les changements (sortie des nœuds). Idéal pour journalisation.
- Liste de modes : `["messages", "updates"]` pour combiner.

Avec `version="v2"`, chaque chunk a une clé `type` et une clé `data` : on peut router selon le type. C'est le format moderne.

Refactoring : centraliser le choix du backend

Tous nos snippets dupliquent le bloc `if backend == "mistral"...`. Le helper `code/_lib/llm.py` centralise cette logique.

```python
"""Refactoring : extraire la sélection de backend dans `_lib/llm.py`.

Le helper `get_llm()` renvoie ChatMistralAI ou ChatOllama selon LLM_BACKEND.
On ajoute `code/` à sys.path pour que `from _lib.llm import get_llm` fonctionne
quel que soit le dossier de lancement.
"""
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))

from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_core.tools import tool
from _lib.llm import get_llm

load_dotenv()
llm = get_llm(temperature=0)


@tool
def multiplier(a: int, b: int) -> int:
    """Multiplie deux entiers."""
    return a * b


agent = create_agent(model=llm, tools=[multiplier])
resultat = agent.invoke(
    {"messages": [{"role": "user", "content": "Combien font 12 fois 11 ?"}]}
)
print(resultat["messages"][-1].content)
```

Pourquoi cette refactorisation maintenant ?

- Au chapitre 1, l'instanciation directe avait une vertu pédagogique (« voir le ChatModel choisi »).
- Avec 4-5 snippets par chapitre, la duplication devient bruyante et masque l'intention de chaque snippet.
- Centraliser permet de changer de modèle ou ajouter un backend sans toucher 20 fichiers.

Astuce Python

`sys.path.insert(0, ...)` est nécessaire car les snippets sont exécutés depuis leur dossier (pas depuis la racine). Pour un *vrai* projet, on transformerait `_lib` en package installable (`pyproject.toml`) ou on utiliserait `src/`-layout.

Exercices

1. Modifiez `01_memoire_simple.py` pour ajouter un troisième tour qui demande le carré de l'âge mentionné au tour 1.
2. Dans `02_threads_separes.py`, ajoutez un troisième thread qui parle d'un sujet sans lien (recette de cuisine). Vérifiez l'isolation.
3. Combinez `stream_mode=["messages", "updates"]` dans `03_streaming_tokens.py` pour afficher à la fois les tokens et les appels d'outils.
4. Refactorisez les snippets du chapitre 4 pour utiliser `get_llm()`. Combien de lignes économisées ?

Récapitulatif

- `checkpointer=InMemorySaver()` + `thread_id` dans le `config` = mémoire entre appels.
- Plusieurs `thread_id` = conversations isolées en parallèle.
- `stream_mode="messages"` avec `version="v2"` = tokens en direct.
- Centraliser la sélection de backend dans `_lib/llm.py` simplifie l'évolution du cours.

Suite

Au prochain cours : LangGraph pour les boucles non triviales (validation humaine, branchements conditionnels, sous-graphes).

LangGraph pour les boucles complexes

Bibliothèques pour ce chapitre

```
# Bibliothèques à installer (chapitre 06)

# ‘langgraph‘ est déjà installé (transitive via langchain).
# Si besoin explicite :
uv add langgraph
```

Pourquoi sortir de `create_agent` ?

`create_agent` est parfait pour la boucle LLM $\leftrightarrow$ outils. Mais beaucoup de cas réels demandent davantage :

- Branchement conditionnel : si le message est du spam, ne pas appeler le LLM cher.
- Validation humaine avant l'exécution d'un outil sensible.
- Pipelines multi-étapes : classifier $\rightarrow$ raffiner $\rightarrow$ valider $\rightarrow$ répondre.
- Boucles avec condition d'arrêt métier (jusqu'à ce qu'un score atteigne X).

Bonne nouvelle

`create_agent` est un `StateGraph` en interne. On apprend ici à le construire à la main.

Anatomie d'un `StateGraph`

Trois ingrédients seulement :

- State : un `TypedDict` (ou `Pydantic BaseModel`) décrivant ce qui circule.
- Nodes : des fonctions `state` $\rightarrow$ `partial state`. Pures, déterministes, ou appelant un LLM.
- Edges : qui pointe vers qui. Statiques ou conditionnelles.

Deux nœuds spéciaux : `START` (point d'entrée) et `END` (sortie).

Ce qui change

Avec `create_agent`, on *configure* un graphe préfait. Avec `StateGraph`, on décrit le graphe nous-mêmes.

Premier graphe : pipeline linéaire

```python
"""Premier StateGraph : un état, deux nœuds, deux arêtes."""
from typing import TypedDict
from langgraph.graph import StateGraph, START, END


class Etat(TypedDict):
    sujet: str
    blague: str | None


def raffiner_sujet(etat: Etat) -> Etat:
    return {"sujet": etat["sujet"] + " et les chats"}


def generer_blague(etat: Etat) -> Etat:
    return {"blague": f"Voici une blague sur {etat['sujet']}..."}


graphe = (
    StateGraph(Etat)
    .add_node(raffiner_sujet)
    .add_node(generer_blague)
    .add_edge(START, "raffiner_sujet")
    .add_edge("raffiner_sujet", "generer_blague")
    .add_edge("generer_blague", END)
    .compile()
)

resultat = graphe.invoke({"sujet": "les pompiers", "blague": None})
print(resultat)
```

Lecture

1. `Etat` déclare les champs persistants (`sujet`, `blague`).
2. Chaque nœud retourne un dict partiel : seules les clés présentes sont mises à jour.
3. `add_edge(START, "raffiner_sujet")` démarre par ce nœud ;
   `add_edge("generer_blague", END)` termine.
4. `compile()` produit un Pregel qui s'invoque comme un agent.

Branchement conditionnel

Une fonction `router` retourne le nom du prochain nœud (ou `END`).

```python
"""Branchement conditionnel : route le flux selon l'état."""
from typing import Literal, TypedDict
from langgraph.graph import StateGraph, START, END


class Etat(TypedDict):
    message: str
    longueur: int | None
    reponse: str | None


def mesurer(etat: Etat) -> Etat:
    return {"longueur": len(etat["message"])}


def reponse_courte(etat: Etat) -> Etat:
    return {"reponse": "Message court reçu."}


def reponse_longue(etat: Etat) -> Etat:
    return {"reponse": "Message long, je résume mentalement."}


def router(etat: Etat) -> Literal["reponse_courte", "reponse_longue"]:
    return "reponse_courte" if etat["longueur"] < 30 else "reponse_longue"


graphe = (
    StateGraph(Etat)
    .add_node(mesurer)
    .add_node(reponse_courte)
    .add_node(reponse_longue)
    .add_edge(START, "mesurer")
    .add_conditional_edges("mesurer", router)
    .add_edge("reponse_courte", END)
    .add_edge("reponse_longue", END)
    .compile()
)

for msg in ["Salut", "Pourrais-tu me détailler en quelques lignes le principe de la photosynthèse ?"]:
    etat = graphe.invoke({"message": msg, "longueur": None, "reponse": None})
    print(f"({etat['longueur']} car) {etat['reponse']}")
```

Lecture

- `add_conditional_edges("mesurer", router)` : après `mesurer`, c'est `router` qui décide.
- Le type de retour de `router` (`Literal["reponse_courte", "reponse_longue"]`) sert de documentation et de validation.
- Toutes les destinations possibles doivent être des nœuds existants ou `END`.

Bonne pratique

Garder le routeur idempotent et rapide : ne pas y appeler de LLM, juste lire l'état.

Boucle avec condition d'arrêt

Le `Annotated[..., operator.add]` indique à LangGraph que les retours doivent être accumulés plutôt qu'écrasés.

```python
"""Boucle avec arrêt conditionnel : itérer jusqu'à atteindre une condition."""
import operator
from typing import Annotated, Literal, TypedDict
from langgraph.graph import StateGraph, START, END


class Etat(TypedDict):
    # Le reducer `operator.add` accumule au lieu d'écraser.
    valeurs: Annotated[list[int], operator.add]


def ajouter(etat: Etat) -> Etat:
    n = len(etat["valeurs"])
    return {"valeurs": [n + 1]}


def continuer(etat: Etat) -> Literal["ajouter", END]:
    return "ajouter" if sum(etat["valeurs"]) < 20 else END


graphe = (
    StateGraph(Etat)
    .add_node(ajouter)
    .add_edge(START, "ajouter")
    .add_conditional_edges("ajouter", continuer)
    .compile()
)

resultat = graphe.invoke({"valeurs": []})
print("Valeurs :", resultat["valeurs"])
print("Somme  :", sum(resultat["valeurs"]))
```

Reducers

Sans reducer, chaque nœud remplace la valeur précédente. Avec un reducer, il fusionne.

- `operator.add` sur une liste : concatène.
- `add_messages` (langgraph) : ajoute des messages en évitant les doublons (utilisé par `create_agent`).
- Un reducer custom est juste une fonction `(old, new) -> merged`.

Une boucle dans LangGraph est une simple arête conditionnelle qui peut renvoyer vers un nœud déjà visité. Pas de `while` explicite.

Hybride : classifier puis expert (LLM)

Un graphe peut mélanger nœuds Python et nœuds appelant un LLM. Ici un classifieur LLM route vers le bon expert LLM.

```python
"""Hybride LLM : un classifieur route vers le bon expert."""
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))

from typing import Literal, TypedDict
from dotenv import load_dotenv
from langgraph.graph import StateGraph, START, END
from _lib.llm import get_llm

load_dotenv()
llm = get_llm(temperature=0)


class Etat(TypedDict):
    question: str
    domaine: str | None
    reponse: str | None


def classifier(etat: Etat) -> Etat:
    prompt = (
        f"La question suivante porte-t-elle sur les mathématiques ou autre chose ? "
        f"Réponds par un seul mot parmi {{math, autre}}.\n\nQuestion : {etat['question']}"
    )
    domaine = llm.invoke(prompt).content.strip().lower()
    return {"domaine": "math" if "math" in domaine else "autre"}


def expert_math(etat: Etat) -> Etat:
    r = llm.invoke(f"Tu es prof de maths. Réponds en une phrase.\n\n{etat['question']}")
    return {"reponse": r.content}


def expert_general(etat: Etat) -> Etat:
    r = llm.invoke(f"Réponds brièvement.\n\n{etat['question']}")
    return {"reponse": r.content}


def router(etat: Etat) -> Literal["expert_math", "expert_general"]:
    return "expert_math" if etat["domaine"] == "math" else "expert_general"


graphe = (
    StateGraph(Etat)
    .add_node(classifier)
    .add_node(expert_math)
    .add_node(expert_general)
    .add_edge(START, "classifier")
    .add_conditional_edges("classifier", router)
    .add_edge("expert_math", END)
    .add_edge("expert_general", END)
    .compile()
)

for q in ["Combien font 12 fois 13 ?", "Quel est le plus grand fleuve d'Afrique ?"]:
    etat = graphe.invoke({"question": q, "domaine": None, "reponse": None})
    print(f"[{etat['domaine']}] Q: {q}")
    print(f"        R: {etat['reponse']}\n")
```

Pourquoi diviser en nœuds ?

- Chaque nœud a un prompt court et spécifique : meilleur résultat qu'un seul méga-prompt.
- Le `system_prompt` de chaque expert peut être adapté à son domaine.
- On peut surveiller, logger, mettre en cache au niveau d'un seul nœud.
- On peut remplacer un nœud (par ex. expert_math $\rightarrow$ outil calculatrice) sans toucher au reste.

create_agent vs StateGraph

- `create_agent` : 90 % des cas, une ligne suffit. Bon défaut.
- `StateGraph` : quand on a besoin de contrôle fin (validation humaine, branchements métier, sous-graphes, multi-agents).
- Sous le capot, `create_agent` *est* un `StateGraph` avec deux nœuds (model, tools) et une boucle conditionnelle.

Astuce

On peut visualiser un graphe avec `graphe.get_graph().draw_mermaid_png()`. Pratique en debug.

Exercices

1. Modifiez `02_branchement.py` pour ajouter un troisième cas « moyen » (entre 30 et 100 caractères).
2. Dans `03_boucle.py`, faites afficher un message à chaque tour de boucle (ajoutez un nœud `tracer`).
3. Étendez `04_classifier_expert.py` avec un troisième expert (« histoire »). Ajoutez les arêtes nécessaires.
4. Reconstruisez à la main `create_agent` avec un `StateGraph` : un nœud `model`, un nœud `tools`, une boucle conditionnelle. Bonus : faites-le boucler tant que `tool_calls` n'est pas vide.

Récapitulatif

- `StateGraph(State)` + `add_node` + `add_edge` + `compile` = un graphe de calcul typé.
- `add_conditional_edges` permet le routage métier ; le routeur est une fonction pure rapide.
- Les reducers (`Annotated[..., op]`) gouvernent la fusion des états dans les boucles.
- Nœuds Python et nœuds LLM cohabitent sans friction.

Suite

Au prochain et dernier cours : un RAG minimal pour donner à l'agent un accès à des documents externes (PDF, base vectorielle).

RAG minimal

Bibliothèques pour ce chapitre

```
# Bibliothèques à installer (chapitre 07)

uv add langchain-text-splitters numpy

# Si LLM_BACKEND=ollama, il faut aussi un modèle d'embeddings :
ollama pull nomic-embed-text
```

Pourquoi RAG ?

Un LLM a une connaissance figée (date d'entraînement) et ne sait *rien* de :

- Vos documents internes (notes, mails, manuels, contrats).
- Les évolutions postérieures à son entraînement.
- Les bases métier propres à votre organisation.

Idée

Retrieval-Augmented Generation : avant de répondre, on *retrouve* les passages pertinents dans une base, et on les *fournit en contexte* au LLM.

Pipeline RAG en cinq étapes

1. Loader : charger les documents (texte, PDF, web, CSV...).
2. Splitter : découper en *chunks* de taille gérable.
3. Embeddings : convertir chaque chunk en vecteur dense.
4. VectorStore : stocker les vecteurs (en mémoire, Postgres+pgvector, Qdrant...).
5. Retriever : pour une question, retourner les k chunks les plus proches en cosinus.

Le LLM n'intervient qu'à la toute fin, en lisant question + chunks retrouvés.

Les embeddings, en deux mots

Un embedding est un vecteur dense (typiquement 768-1536 dimensions) tel que des textes *sémantiquement proches* ont des vecteurs proches.

- Mistral : `mistral-embed` (1024 dim, cloud).
- Ollama : `nomic-embed-text` (768 dim, local).
- OpenAI : `text-embedding-3-small` (1536 dim, cloud).

Bonne pratique

Toujours *indexer* et *interroger* avec le même modèle d'embeddings. Sinon les distances n'ont aucun sens.

Étape 1 : indexer un document

```python
"""Étape 1 : découper un document, calculer les embeddings, indexer en mémoire."""
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))

from dotenv import load_dotenv
from langchain_core.documents import Document
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_text_splitters import RecursiveCharacterTextSplitter
from _lib.embeddings import get_embeddings

load_dotenv()
texte = (Path(__file__).parent / "data" / "iut.txt").read_text()

splitter = RecursiveCharacterTextSplitter(chunk_size=200, chunk_overlap=40)
chunks = splitter.split_text(texte)
documents = [Document(page_content=c) for c in chunks]

vectorstore = InMemoryVectorStore.from_documents(documents=documents, embedding=get_embeddings())
print(f"{len(documents)} chunks indexés.")
print("Premier chunk :", documents[0].page_content[:80], "...")
```

Choisir `chunk_size` et `chunk_overlap`

- `chunk_size` *trop petit* : on perd le contexte (« cette mesure », « cet auteur » sans antécédent).
- `chunk_size` *trop grand* : la recherche devient bruitée, le LLM doit lire plus.
- `chunk_overlap` : recouvrement entre chunks pour éviter de couper en plein milieu d'une idée.

Valeurs courantes : `chunk_size=500-1000`, `chunk_overlap=50-200`, à ajuster selon le type de texte.

Étape 2 : rechercher par similarité

```python
"""Étape 2 : recherche par similarité sémantique."""
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))

from dotenv import load_dotenv
from langchain_core.documents import Document
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_text_splitters import RecursiveCharacterTextSplitter
from _lib.embeddings import get_embeddings

load_dotenv()
texte = (Path(__file__).parent / "data" / "iut.txt").read_text()
splitter = RecursiveCharacterTextSplitter(chunk_size=200, chunk_overlap=40)
documents = [Document(page_content=c) for c in splitter.split_text(texte)]
vectorstore = InMemoryVectorStore.from_documents(documents=documents, embedding=get_embeddings())

for question in ["Combien d'étudiants par promotion ?", "Sur quoi travaille Christophe Guyeux ?"]:
    print(f"\nQ : {question}")
    for doc in vectorstore.similarity_search(question, k=2):
        print(f"  - {doc.page_content[:100]}...")
```

La similarité sémantique

- `similarity_search(question, k=2)` encode la question, calcule la similarité cosinus avec tous les chunks indexés, retourne les k premiers.
- C'est coûteux en RAM mais simple à mettre en œuvre. Pour un cours, c'est parfait. Pour 10 millions de chunks, on passe à pgvector ou Qdrant.

Les chunks retrouvés ne sont pas toujours pertinents : on appelle ça le *ranker* et on peut le perfectionner avec un re-ranker (ex. : Cohere Rerank).

Étape 3 : RAG « naïf » en chaîne LCEL

```python
"""Étape 3 : RAG « naïf » en chaîne LCEL : retrieve → prompt → llm."""
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))

from dotenv import load_dotenv
from langchain_core.documents import Document
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnablePassthrough
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_text_splitters import RecursiveCharacterTextSplitter
from _lib.embeddings import get_embeddings
from _lib.llm import get_llm

load_dotenv()
texte = (Path(__file__).parent / "data" / "iut.txt").read_text()
splitter = RecursiveCharacterTextSplitter(chunk_size=200, chunk_overlap=40)
documents = [Document(page_content=c) for c in splitter.split_text(texte)]
retriever = InMemoryVectorStore.from_documents(documents, get_embeddings()).as_retriever(k=2)

prompt = ChatPromptTemplate.from_template(
    "Réponds à la question en t'appuyant uniquement sur le contexte ci-dessous. "
    "Si l'info manque, dis-le.\n\nContexte :\n{contexte}\n\nQuestion : {question}"
)


def format_docs(docs):
    return "\n---\n".join(d.page_content for d in docs)


chain = (
    {"contexte": retriever | format_docs, "question": RunnablePassthrough()}
    | prompt
    | get_llm(temperature=0)
    | StrOutputParser()
)

print(chain.invoke("Combien d'étudiants par promotion en informatique à l'IUT NFC ?"))
```

Anatomie de la chaîne

`{contexte: retriever, question: passthrough}` → `prompt` → `llm` → `StrOutputParser`

- `RunnablePassthrough()` laisse passer la question telle quelle.
- Le `retriever` est appelé sur la question puis formaté en texte.
- Le LLM voit *contexte* + *question* et formule la réponse.

Limite

Ce RAG appelle toujours le retriever, même quand ce n'est pas utile (« Combien font 2 + 2 ? »). On améliore avec un agent.

Étape 4 : RAG agentique

L'agent décide de lui-même d'appeler ou non l'outil de recherche.

```python
"""Étape 4 : RAG agentique. L'agent décide quand appeler l'outil de recherche."""
import sys
from pathlib import Path
sys.path.insert(0, str(Path(__file__).parent.parent))

from dotenv import load_dotenv
from langchain.agents import create_agent
from langchain_core.documents import Document
from langchain_core.tools import tool
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_text_splitters import RecursiveCharacterTextSplitter
from _lib.embeddings import get_embeddings
from _lib.llm import get_llm

load_dotenv()
texte = (Path(__file__).parent / "data" / "iut.txt").read_text()
splitter = RecursiveCharacterTextSplitter(chunk_size=200, chunk_overlap=40)
documents = [Document(page_content=c) for c in splitter.split_text(texte)]
vs = InMemoryVectorStore.from_documents(documents, get_embeddings())


@tool
def rechercher_iut(question: str) -> str:
    """Recherche dans la documentation officielle de l'IUT NFC."""
    return "\n---\n".join(d.page_content for d in vs.similarity_search(question, k=2))


agent = create_agent(model=get_llm(temperature=0), tools=[rechercher_iut])

for question in [
    "Que fait Christophe Guyeux dans la recherche ?",
    "Combien font 12 fois 11 ?",  # pas de RAG nécessaire
]:
    r = agent.invoke({"messages": [{"role": "user", "content": question}]})
    print(f"Q : {question}")
    print(f"R : {r['messages'][-1].content}\n")
```

RAG naïf vs RAG agentique

- Naïf : 1 retrieval forcé + 1 appel LLM. Latence et coût stables, mais retrieve même quand inutile.
- Agentique : l'agent peut décider *ne pas* retrieve, ou retrieve *plusieurs fois* avec des reformulations. Plus souple, plus coûteux dans les cas complexes.

Tendance 2025-2026

Le RAG agentique gagne du terrain : les agents savent désormais *quand* chercher, *quoi* chercher, et *comment* reformuler après un échec.

Pour aller plus loin

- Loaders riches : `PyPDFLoader`, `WebBaseLoader`, `NotionLoader`, `GitHubIssuesLoader`.
- VectorStores persistants : `Chroma`, `Qdrant`, `pgvector`, `Weaviate`.
- Re-ranking avec un modèle dédié pour réorganiser le top-k.
- HyDE : générer une réponse hypothétique et faire le retrieval dessus.
- Multi-query : reformuler la question en plusieurs variantes pour élargir le retrieval.

Exercices

1. Remplacez `iut.txt` par un PDF (utilisez `PyPDFLoader`, `uv add langchain-community pypdf`). Comparez les résultats.
2. Faites varier `chunk_size` entre 100 et 1000 dans `01_indexer.py`. Pour quelles questions le RAG fonctionne-t-il mieux ?
3. Dans `04_rag_agentique.py`, ajoutez un second outil `multiplier` et posez une question composée « Combien d'étudiants $\times$ 12 mois ? ». Observez la trajectoire.
4. Persistez l'index sur disque avec `Chroma` ou `FAISS` pour ne pas réindexer à chaque démarrage.

Récapitulatif

- RAG = retrieval de chunks + augmentation du prompt + generation par le LLM.
- Pipeline : Loader → Splitter → Embeddings → VectorStore → Retriever.
- RAG naïf en LCEL : 5 lignes. RAG agentique avec `create_agent` : un outil de recherche + le LLM décide.
- Toujours indexer et interroger avec le même modèle d'embeddings.

Fin du cours

Vous avez maintenant les briques pour construire un assistant documentaire de bout en bout. À vous de jouer.

Merci.

Questions ?