

CARNET DE TP

Séance 1 — Fondations & Architecture Agents

Implémentation pas à pas — LangGraph + Ollama

Nom	Prénom	Date	Score /35 pts
-----	--------	------	------------------

Ce carnet vous guide micro-étape par micro-étape.
Chaque case est une action concrète à effectuer.
Chaque bloc vert 'Résultat attendu' vous dit quoi vérifier.
Ne passez à l'étape suivante que si la vérification passe.

Avant de commencer — 5 minutes

Effectuez ces vérifications AVANT d'ouvrir le projet.

Vérifier qu'Ollama tourne

1

```
ollama serve &

# Dans un autre terminal :
curl http://localhost:11434/
# Résultat attendu : Ollama is running
```

⚠ Si Ollama n'est pas démarré, l'agent ne peut pas appeler le LLM.
Gardez le terminal ollama serve ouvert pendant tout le TP.

Vérifier le modèle disponible

2

```
ollama list
# Vous devez voir llama3.1:8b OU mistral:7b dans la liste
```

✓ Résultat attendu

NAME	ID	SIZE	MODIFIED
llama3.1:8b	...	4.7 GB	...

⚠ Si la liste est vide : ollama pull llama3.1:8b (4.7 GB, prévoir 10 min)

Extraire et ouvrir le projet squelette

3

```
# Extraire le zip fourni par le formateur
unzip S1_skeleton.zip -d agent-project
cd agent-project

# Ouvrir dans VS Code
code .
```

Dans VS Code, vous devez voir cette structure dans l'explorateur :

```
agent-project/
  src/
    agents/react_agent.py ← à implémenter
    tools/catalogue.py ← à implémenter
    models/provider.py ← à implémenter
  tests/unit/
    test_tools.py ← fourni (spec)
```

```
test_agent.py      ← fourni (spec)
main.py           ← fourni
```

4 Installer les dépendances

```
uv sync --extra dev

# Vérifier que LangGraph est installé
uv run python -c "import langgraph; print(langgraph.__version__)"
# Résultat attendu : 0.2.x ou supérieur
```

✓ Résultat attendu

```
0.2.x
```

5 Copier et vérifier le .env

```
cp .env.example .env

# Vérifier le contenu
cat .env

# Vous devez voir :
# LLM_BASE_URL=http://localhost:11434/v1
# MODEL_NAME=llama3.1:8b
```

⚠ Si vous utilisez mistral:7b, modifiez MODEL_NAME=mistral:7b dans .env

6 Lancer les tests — tous doivent échouer (c'est normal)

```
uv run pytest tests/unit/ -v 2>&1 | tail -20
```

✓ Résultat attendu

```
FAILED
tests/unit/test_tools.py::TestSearchProduct::test_found_by_partial_name
...
NotImplementedError: TODO — implémenter search_product
35 failed in X.Xs
```

✓ 35 tests FAILED = normal ! Votre travail est de les faire passer un par un.
0 PASSED au départ est attendu.

PHASE 1 — Les outils du catalogue

Fichier : `src/tools/catalogue.py` | Durée estimée : 25 min | 23 tests à faire passer

Ouvrez `src/tools/catalogue.py` dans VS Code.
Le dictionnaire CATALOGUE est déjà rempli — ne le modifiez pas.
Votre travail : écrire le corps des 4 fonctions marquées TODO.

Outil 1 — `search_product()` (5 tests)

Objectif : parcourir CATALOGUE et retourner les produits dont le nom ou la description contient le mot cherché (insensible à la casse).

Comprendre la structure de CATALOGUE

Regardez les premières lignes du dictionnaire :

1

```
"laptop-pro-15": {
    "name": "Laptop Pro 15",
    "price": 1299.0,
    "stock": 5,
    "category": "informatique",
    "description": "Laptop haute performance...",
},
```

La clé est l'identifiant (ex: 'laptop-pro-15').

Vous devez chercher dans : la clé ET le champ 'name' ET le champ 'description'.

Écrire la boucle de recherche

Remplacez le raise `NotImplementedError` par ce code :

2

```
name_lower = name.lower().strip()
results = []
for key, product in CATALOGUE.items():
    if (name_lower in key
        or name_lower in str(product['name']).lower()
        or name_lower in str(product['description']).lower()):
        results.append(product)
```

Astuce : `str(product['name']).lower()` convertit en minuscules pour la comparaison.
`name_lower in key` cherche dans l'identifiant (ex: 'laptop-pro-15')

Gérer le cas 'aucun résultat'

3

```
if not results:
```

```

return f"Aucun produit trouvé pour '{name}'."
# Astuce : citez le terme cherché dans le message - c'est dans les
tests

```

Formater et retourner les résultats

4

```

lines = []
for p in results:
    stock_info = f"{p['stock']} en stock" if p['stock'] else "RUPTURE"
    lines.append(f"- {p['name']} : {p['price']:.2f}€ ({stock_info})")
return f"Produits trouvés ({len(results)}) :\n" + "\n".join(lines)

```

Lancer les tests search_product

```
uv run pytest tests/unit/test_tools.py::TestSearchProduct -v
```

✓ Résultat attendu

5

```

PASSED test_found_by_partial_name
PASSED test_found_case_insensitive
PASSED test_not_found_returns_message
PASSED test_search_by_description_word
PASSED test_multiple_results
5 passed

```

X Si un test échoue, lisez le message d'erreur — il indique exactement quelle valeur était attendue et ce que vous avez retourné.
Exemple : `AssertionError: assert 'Laptop Pro 15' in 'Aucun produit...'`
→ votre condition de recherche est trop restrictive.

Outil 2 — calculate_price() (7 tests)

Objectif : rechercher un produit, puis calculer prix × quantité avec remise optionnelle.

Valider les paramètres d'entrée

Ajoutez ces validations EN PREMIER dans la fonction :

6

```

if quantity <= 0:
    return "La quantité doit être un entier positif."
if not (0.0 <= discount_percent <= 100.0):
    return "Le pourcentage de remise doit être compris entre 0 et 100."

```

Les tests vérifient exactement ces cas : quantité=0 et discount=150.
Les mots 'positif' et '100' doivent apparaître dans le message retourné.

Rechercher le produit dans CATALOGUE

7

```

name_lower = product_name.lower().strip()
matches = [
    (key, prod) for key, prod in CATALOGUE.items()
    if name_lower in key or name_lower in str(prod['name']).lower()
]

if not matches:
    return f"Produit '{product_name}' introuvable dans le catalogue."

key, product = matches[0] # On prend le premier résultat

```

Calculer le prix

8

```

unit_price = float(product['price'])
subtotal = unit_price * quantity
discount_amount = subtotal * (discount_percent / 100)
total = subtotal - discount_amount

```

Vérification rapide à la main :

souris × 2 = 2 × 49.90 = 99.80 → remise 10% = 9.98 → total = 89.82

Construire et retourner le récapitulatif

9

```

stock_warning = ""
if int(product['stock']) == 0:
    stock_warning = "\n⚠️ RUPTURE DE STOCK – commande impossible."
elif int(product['stock']) < quantity:
    stock_warning = f"\n⚠️ Stock insuffisant : {product['stock']} disponibles."

return (
    f"Produit : {product['name']}\n"
    f"Prix unitaire : {unit_price:.2f}€\n"
    f"Quantité : {quantity}\n"
    f"Remise ({discount_percent}%) : -{discount_amount:.2f}€\n"
    f"TOTAL : {total:.2f}€"
    f"{stock_warning}"
)

```

Lancer les tests calculate_price

1

0

```
uv run pytest tests/unit/test_tools.py::TestCalculatePrice -v
```

✓ Résultat attendu

7 passed

⚠ Si test_price_with_discount échoue sur '89.82' :
vérifiez l'arrondi. Python : $2 \times 49.90 \times 0.9 = 89.82$ exactement.
Utilisez f'{total:.2f}' pour formater avec 2 décimales.

Outil 3 — list_products() (6 tests)

Objectif : lister les produits du catalogue avec filtre optionnel par catégorie. Séparer les produits disponibles des ruptures de stock.

Filtrer par catégorie (si fournie)

1
1

```
products = list(CATALOGUE.values())

if category:
    category_lower = category.lower().strip()
    products = [p for p in products
                if str(p.get('category', '')).lower() == category_lower]
```

Attention : utiliser == (égalité exacte) et non 'in' pour le filtre catégorie.
Test test_filter_informatique vérifie que 'Souris' n'apparaît PAS.

Gérer la catégorie inconnue

1
2

```
if not products:
    return f"Aucun produit dans la catégorie '{category}'."
# Le test vérifie que le nom de la catégorie apparaît dans le message
```

Séparer en stock / rupture et formater

1
3

```
in_stock = [p for p in products if int(p['stock']) > 0]
out_of_stock = [p for p in products if int(p['stock']) == 0]

lines = [f"Catalogue ({len(products)} produits) :"]

for prod in in_stock:
    lines.append(f" ✓ {prod['name']}: {prod['price']:.2f}€ (stock: {prod['stock']})")

for prod in out_of_stock:
    lines.append(f" ✗ {prod['name']}: {prod['price']:.2f}€ (RUPTURE)")

return "\n".join(lines)
```

Le test `test_out_of_stock_shown` cherche 'RUPTURE' en majuscules.
Assurez-vous que ce mot apparaît exactement ainsi dans votre sortie.

Lancer les tests `list_products`

1
4

```
uv run pytest tests/unit/test_tools.py::TestListProducts -v
```

✓ Résultat attendu

```
6 passed
```

Lancer TOUS les tests outils

1
5

```
uv run pytest tests/unit/test_tools.py -v
```

✓ Résultat attendu

```
23 passed (ou 18 passed si check_stock pas encore fait)
```

✓ Si 23/23 (ou 18/18) passent : félicitations, phase 1 terminée !
Passez à la phase 2. `check_stock` est un bonus à faire en phase 4.

Mettre à jour `CATALOGUE_TOOLS`

En bas de `catalogue.py`, décommentez les outils implémentés :

1
6

```
CATALOGUE_TOOLS = [  
    search_product,  
    calculate_price, # décommenter  
    list_products, # décommenter  
    # check_stock, # laisser commenté pour l'instant (bonus)  
]
```

PHASE 2 — L'abstraction LLM

Fichier : `src/models/provider.py` | Durée estimée : 10 min

Ouvrez `src/models/provider.py` dans VS Code.
Cette fonction est le pont entre LangGraph et Ollama.
En séance 3, vous l'enrichirez pour le routing multi-modèle.

Lire les variables d'environnement

Remplacez le `raise NotImplementedError` par :

```
base_url: str = os.getenv('LLM_BASE_URL', 'http://localhost:11434/v1')
model_name: str = os.getenv('MODEL_NAME', 'llama3.1:8b')
```

1
7

`os.getenv('CLE', 'valeur_par_defaut')` lit la variable ou utilise le défaut.
Ainsi le code fonctionne même sans fichier `.env`.

Instancier et retourner ChatOpenAI

```
return ChatOpenAI(
    base_url=base_url,
    model=model_name,
    api_key='local',      # ignoré par Ollama mais requis
    temperature=0,      # 0 = déterministe, essentiel pour les agents
    max_tokens=int(os.getenv('AGENT_MAX_TOKENS', '4096')),
    timeout=120,
)
```

1
8

Vérifier manuellement la connexion Ollama

```
uv run python -c "
from src.models.provider import get_llm
from langchain_core.messages import HumanMessage
llm = get_llm()
r = llm.invoke([HumanMessage('Dis bonjour en 5 mots.')])
print(r.content)
"
```

1
9

✓ Résultat attendu

```
Bonjour ! Comment puis-je aider ?
(ou toute réponse courte en français)
```

X Si vous voyez 'Connection refused' ou 'Error 404' :

- Ollama n'est pas démarré. Lancez : `ollama serve &`
- X Si vous voyez 'model not found' :
 - Le modèle n'est pas téléchargé. Lancez : `ollama pull llama3.1:8b`

PHASE 3 — Le graphe LangGraph

Fichier : `src/agents/react_agent.py` | Durée estimée : 30 min | 11 tests

Ouvrez `src/agents/react_agent.py` dans VS Code.
 AgentState et SYSTEM_PROMPT sont déjà écrits — ne les modifiez pas.
 Implémentez les 3 fonctions dans l'ordre : `should_continue`, `agent_node`, `build_agent`.

Fonction 1 — `should_continue()`

Cette fonction décide à chaque tour : l'agent doit-il appeler un outil ou s'arrêter ?

Lire le compteur et vérifier la limite

Remplacez le raise `NotImplementedError` par :

2
0

```
max_iter = int(os.getenv('AGENT_MAX_ITERATIONS', '20'))

if state.get('iterations', 0) >= max_iter:
    return END # stop : trop d'itérations
```

Cette protection est critique : sans elle, un agent peut tourner à l'infini.
 END est importé depuis `langgraph.graph` — c'est une constante.

Inspecter le dernier message

2
1

```
last_message = state['messages'][-1]

if isinstance(last_message, AIMessage):
    if hasattr(last_message, 'tool_calls') and last_message.tool_calls:
        return 'tools' # le LLM veut utiliser un outil

return END # pas de tool_calls = réponse finale
```

La logique complète de `should_continue` :

```
iterations ≥ max → END
tool_calls présents → 'tools'
sinon → END
```

Tester `should_continue`

2
2

```
uv run pytest tests/unit/test_agent.py::TestShouldContinue -v
```

✓ Résultat attendu

```
PASSED test_returns_tools_when_tool_calls_present
PASSED test_returns_end_when_no_tool_calls
PASSED test_returns_end_at_max_iterations
PASSED test_continues_below_max_iterations
4 passed
```

Fonction 2 — agent_node()

Ce noeud envoie les messages au LLM et récupère sa décision (réponse ou tool_call).

Obtenir le LLM et le lier aux outils

2
3

```
llm = get_llm(task='default')
llm_with_tools = llm.bind_tools(CATALOGUE_TOOLS)

# bind_tools() indique au LLM quels outils sont disponibles
# Il peut alors retourner des tool_calls dans sa réponse
```

Construire la liste de messages avec le prompt système

2
4

```
messages: list[BaseMessage] = [
    SystemMessage(content=SYSTEM_PROMPT), # contexte et règles
    *list(state['messages']),           # historique complet
]
```

Le SystemMessage doit être en PREMIER — c'est le contexte global de l'agent.
L'opérateur * déplie la liste state['messages'] dans la nouvelle liste.

Appeler le LLM et retourner le résultat

2
5

```
response: AIMessage = llm_with_tools.invoke(messages)

return {
    'messages': [response], # sera ajouté à la
    liste
    'iterations': state.get('iterations', 0) + 1, # incrémenté
}
```

⚠️ Retournez toujours 'messages' comme une LISTE (même avec 1 élément).
LangGraph fusionne ce dict avec l'état via operator.add sur messages.
Un dict sans liste provoquerait une erreur de type.

Tester agent_node

```
uv run pytest tests/unit/test_agent.py::TestAgentNode -v
```

2
6

✓ Résultat attendu

```
4 passed
```

Ces tests mockent le LLM : Ollama n'est pas appelé.
Si un test échoue sur 'NotImplementedError' : vous avez peut-être oublié de retirer un raise NotImplementedError.

Fonction 3 — build_agent()

Cette fonction assemble tous les composants dans un graphe LangGraph compilé.

Ajouter les noeuds

Remplacez le raise NotImplementedError par les étapes suivantes :

2
7

```
graph: StateGraph = StateGraph(AgentState)

# Noeud 1 : l'agent (appelle le LLM)
graph.add_node('agent', agent_node)

# Noeud 2 : les outils (exécute les tool_calls automatiquement)
graph.add_node('tools', ToolNode(CATALOGUE_TOOLS))
```

ToolNode est fourni par LangGraph — il lit les tool_calls dans le dernier message AI, appelle les fonctions correspondantes, et retourne les résultats.

Définir le point d'entrée et les edges

2
8

```
# Point d'entrée : l'exécution commence toujours par 'agent'
graph.set_entry_point('agent')

# Edge conditionnel : depuis 'agent', appeler should_continue()
# selon le retour : aller vers 'tools' OU terminer
graph.add_conditional_edges(
    'agent',
    should_continue,
    {'tools': 'tools', END: END},
)

# Edge simple : depuis 'tools', toujours retourner vers 'agent'
graph.add_edge('tools', 'agent')
```

Compiler et retourner le graphe

```
checkpointer = MemorySaver() if use_memory else None
return graph.compile(checkpointer=checkpointer)
```

2
9

MemorySaver persiste l'état en RAM — les conversations sont mémorisées pendant toute la durée du processus, mais perdues si vous relancez main.py. En séance 2, vous le remplacerez par RedisSaver (persistant).

Décommenter l'instance globale en bas du fichier

Trouvez la ligne commentée tout en bas de react_agent.py et décommentez-la :

```
# Avant :
# agent: CompiledStateGraph = build_agent(use_memory=True)

# Après (enlever le #) :
agent: CompiledStateGraph = build_agent(use_memory=True)
```

3
0

Tester la construction du graphe

```
uv run pytest tests/unit/test_agent.py::TestBuildAgent -v
```

3
1

✓ Résultat attendu

```
PASSED test_graph_compiles
PASSED test_graph_has_agent_and_tools_nodes
PASSED test_mermaid_is_generated
3 passed
```

Lancer TOUS les tests unitaires

```
uv run pytest tests/unit/ -v
```

3
2

✓ Résultat attendu

```
30 passed (ou 34 avec check_stock)
0 failed
```

✓ 30/30 passent = phases 1, 2 et 3 terminées ! Passez à la phase 4.

PHASE 4 — Test manuel et visualisation

Durée estimée : 10 min | Vérifier que l'agent répond correctement

Lancer l'agent et poser les 5 questions de test

```
uv run python main.py --verbose
```

Posez ces 5 questions une par une et vérifiez les réponses :

#	Question	Vérifier dans la réponse
3 3	1 Quel est le prix d'une souris ergonomique ?	49,90€ et le nom 'Souris Ergonomique'
	2 Je veux 3 claviers avec remise 20%, combien ?	Calcul : $3 \times 129 = 387 - 20\% = 309,60\text{€}$
	3 Quels produits audio avez-vous ?	Casque Audio Pro + Enceinte Bluetooth
	4 Puis-je commander 10 écrans 4K ?	Avertissement stock insuffisant (3 dispo)
	5 Quels produits informatiques coûtent moins de 600€ ?	Laptop Pro 15 et Écran 4K 27

Visualiser et sauvegarder le graphe

```
uv run python main.py --visualize

# Résultat : un diagramme Mermaid dans le terminal
# + fichier sauvegardé dans docs/graphs/agent_react.md
```

✓ Résultat attendu

```
graph TD
  __start__ --> agent
  agent -.-> tools
  agent -.-> __end__
  tools --> agent
```

Sauvegardé dans docs/graphs/agent_react.md

Copiez le code Mermaid et collez-le sur <https://mermaid.live> pour voir le graphe sous forme visuelle.

Vérifier le typage et le linting

```
uv run mypy src/  
uv run ruff check src/
```

✓ Résultat attendu

```
mypy : Success: no issues found in X source files  
ruff : All checks passed.
```

- ⚠ Si mypy remonte des erreurs, lisez le message — il indique le fichier, la ligne, et le type attendu vs trouvé.
- ⚠ Si ruff remonte des erreurs, il propose souvent le fix automatique :
`uv run ruff check src/ --fix`

3
5

PHASE 5 — Bonus : check_stock() + premier commit Git

Durée estimée : 15 min | Facultatif mais recommandé

Bonus 1 — Implémenter check_stock()

Implémenter check_stock

Ouvrez `src/tools/catalogue.py` et remplacez le raise de `check_stock` :

```
if required_quantity <= 0:
    return "La quantité doit être un entier positif."

name_lower = product_name.lower().strip()
matches = [p for k, p in CATALOGUE.items()
            if name_lower in k or name_lower in str(p['name']).lower()]

if not matches:
    return f"Produit '{product_name}' introuvable."

product = matches[0]
stock = int(product['stock'])

if stock == 0:
    return f"❌ {product['name']} : RUPTURE DE STOCK."
elif stock >= required_quantity:
    return f"✅ {product['name']} : en stock ({stock} dispo). Commande possible."
else:
    return f"⚠️ Stock insuffisant : {stock} dispo, {required_quantity} demandés."
```

3
6

Décommentez `check_stock` dans `CATALOGUE_TOOLS` :

```
CATALOGUE_TOOLS = [search_product, calculate_price, list_products,
                    check_stock]
```

Valider les tests check_stock

```
uv run pytest tests/unit/test_tools.py::TestCheckStock -v
```

3
7

✓ Résultat attendu

```
5 passed
```

Bonus 2 — Premier commit Git

Vérification finale avant commit

38

```
# Tous les tests doivent passer
uv run pytest tests/unit/ -v
# Attendu : 35 passed, 0 failed

# Pas d'erreurs de typage
uv run mypy src/
# Attendu : Success: no issues found

# Pas d'erreurs de linting
uv run ruff check src/
# Attendu : All checks passed.
```

Initialiser Git et faire le premier commit

39

```
git init
git add .
git commit -m 'feat(s1): agent ReAct catalogue – séance 1 complète'

# Pousser sur le dépôt de la formation (URL fournie par le formateur)
git remote add origin <url-du-depot>
git push -u origin main
```

Communiquer le lien au formateur

40

Envoyez le lien de votre dépôt dans le canal de la formation.

Format : git clone https://... → uv sync → uv run pytest tests/unit/ → 35 passed

Checklist de validation finale

Cochez chaque case avant de remettre votre travail.

<input type="checkbox"/>	Critère	Vérification
<input type="checkbox"/>	search_product() implémentée et testée	pytest TestSearchProduct → 5 passed
<input type="checkbox"/>	calculate_price() implémentée et testée	pytest TestCalculatePrice → 7 passed
<input type="checkbox"/>	list_products() implémentée et testée	pytest TestListProducts → 6 passed
<input type="checkbox"/>	get_llm() implémentée — Ollama répond	python -c 'from src.models.provider import get_llm'
<input type="checkbox"/>	should_continue() implémentée	pytest TestShouldContinue → 4 passed
<input type="checkbox"/>	agent_node() implémentée	pytest TestAgentNode → 4 passed
<input type="checkbox"/>	build_agent() implémentée	pytest TestBuildAgent → 3 passed
<input type="checkbox"/>	30 tests unitaires passent	pytest tests/unit/ → 30 passed
<input type="checkbox"/>	5 questions de test répondues correctement	uv run python main.py --verbose
<input type="checkbox"/>	Graphe visualisé et sauvegardé	docs/graphs/agent_react.md créé
<input type="checkbox"/>	mypy sans erreur	mypy src/ → no issues
<input type="checkbox"/>	ruff sans erreur	ruff check src/ → All checks passed
<input type="checkbox"/>	(Bonus) check_stock() → 35 tests	pytest tests/unit/ → 35 passed
<input type="checkbox"/>	(Bonus) Premier commit Git poussé	git log → 1 commit

Barème

Critère	Points	Obtenu
23 tests outils passent (phases 1)	10 pts	
get_llm() fonctionnel — Ollama répond	5 pts	
11 tests agent passent (phase 3)	10 pts	

5 questions répondues correctement	5 pts	
Bonus : check_stock() — 5 tests passent	3 pts	
Bonus : premier commit Git	2 pts	
TOTAL	35 pts	