GUIDE PRATIQUE 2024

Comment créer un agent IA

Le guide pratique

De la théorie au code : construisez votre premier agent intelligent en Python, avec des outils, de la mémoire et un déploiement en production.

Build Agent • Guide Technique • Niveau : Intermédiaire

Table des matières

CHAPITRE 01

Introduction — Qu'est-ce qu'un agent IA ?

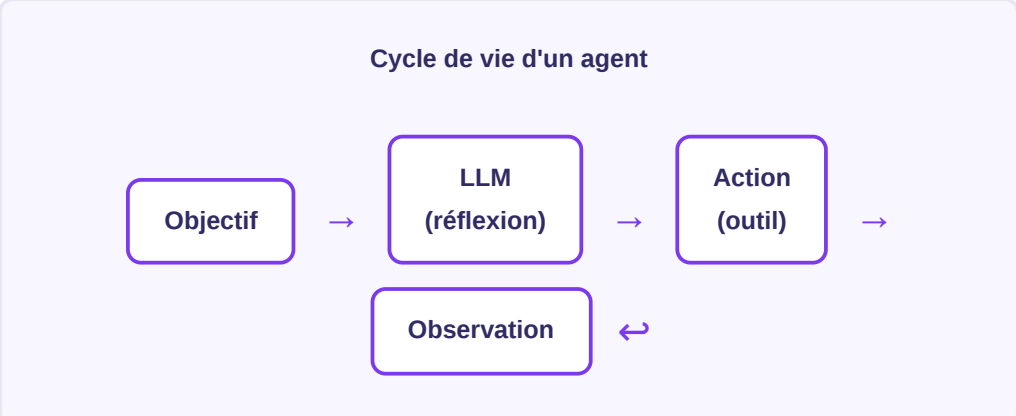
Un agent IA est un programme capable d'**agir de manière autonome** pour accomplir des objectifs définis. Contrairement à un simple chatbot qui répond à des questions, un agent peut planifier une suite d'actions, utiliser des outils externes (APIs, bases de données, code), et adapter sa stratégie en fonction des résultats qu'il obtient.

La définition la plus utilisée est celle d'un *système qui perçoit son environnement, prend des décisions et exécute des actions pour atteindre un but*. Dans le contexte des LLMs modernes, cela se traduit concrètement par une boucle : le modèle lit une situation, choisit une action, observe le résultat, puis recommence jusqu'à la résolution du problème.

Ce qui distingue un agent d'un LLM classique

Un LLM classique (comme GPT-4 ou Claude) est essentiellement stateless : il reçoit un prompt et génère une réponse. Un agent, lui, possède :

- **Une boucle d'exécution** : il agit plusieurs fois de suite avant de rendre un résultat final
- **Des outils** : il peut appeler des fonctions, interroger des APIs, exécuter du code
- **De la mémoire** : il retient le contexte de ce qui s'est passé dans la session, voire au-delà
- **Un objectif** : il travaille vers un but, pas juste une réponse unique



Cas d'usage concrets

Les agents IA s'appliquent dans des domaines très variés :

Domaine	Exemple d'agent	Outils typiques
Productivité	Agent qui gère votre boîte mail et classe les emails	Gmail API, agenda
Développement	Agent qui corrige des bugs en lisant le code et les tests	Bash, Git, éditeur de fichiers
Recherche	Agent qui résume des articles scientifiques et extrait des données	Recherche web, PDF parser
E-commerce	Agent qui surveille les prix et commande quand un seuil est atteint	Scraping, API boutique
Support client	Agent qui répond aux demandes et escalade les cas complexes	Base de connaissance, CRM

POURQUOI MAINTENANT ?

Les agents IA sont devenus viables grâce à deux percées récentes : (1) les LLMs peuvent désormais planifier et raisonner de façon fiable sur plusieurs étapes, et (2) le *function calling* permet aux modèles d'appeler des outils de manière structurée. Ces deux capacités combinées ouvrent la porte à des applications autonomes réellement utiles.

Le spectre d'autonomie

Il n'existe pas qu'un seul type d'agent. On parle souvent d'un spectre allant du **chatbot assisté** (l'humain décide de tout) à l'**agent pleinement autonome** (le modèle décide et agit sans supervision). Dans la pratique, la plupart des agents utiles se situent au milieu : ils automatisent les tâches répétitives tout en laissant à l'humain les décisions importantes.

Ce guide vous montrera comment construire des agents fiables, explicables, et progressivement plus autonomes — en commençant par les fondations.

CHAPITRE 02

Les briques fondamentales d'un agent

Un agent IA moderne est composé de cinq briques principales. Comprendre chacune d'elles est indispensable avant d'écrire la moindre ligne de code.

1. Le modèle de langage (LLM)

Le LLM est le *cerveau* de l'agent. C'est lui qui lit le contexte, raisonne, et décide quelle action prendre ensuite. Les modèles les plus utilisés en production sont GPT-4o (OpenAI), Claude 3.5 Sonnet (Anthropic), et des modèles open-source comme Llama 3 ou Mistral.

Le choix du modèle impacte directement la qualité du raisonnement, la capacité à utiliser les outils correctement, et bien sûr le coût. Nous y reviendrons en détail au chapitre 3.

2. Les prompts système

Le **prompt système** est l'instruction permanente que l'agent reçoit au début de chaque conversation. Il définit :

- Le rôle et la personnalité de l'agent
- Les outils disponibles et comment les utiliser
- Les règles de comportement (ne pas halluciner, demander confirmation si incertain…)
- Le format des réponses attendu

> **BONNE PRATIQUE**
>
> Un bon prompt système est court, précis, et ne laisse pas place à l'ambiguïté. Évitez les instructions contradictoires ou les règles trop générales comme "sois utile et précis".

3. Les outils (Tools)

Les outils sont des fonctions Python (ou tout autre langage) que l'agent peut appeler. Un outil peut :

- Faire une recherche sur le web
- Lire ou écrire un fichier
- Interroger une base de données
- Envoyer un email
- Exécuter du code
- Appeler une API tierce

Chaque outil est décrit au LLM par son nom, sa description en langage naturel, et la liste de ses paramètres (avec types et descriptions). C'est cette description qui permet au modèle de savoir quand et comment utiliser l'outil.

4. La mémoire

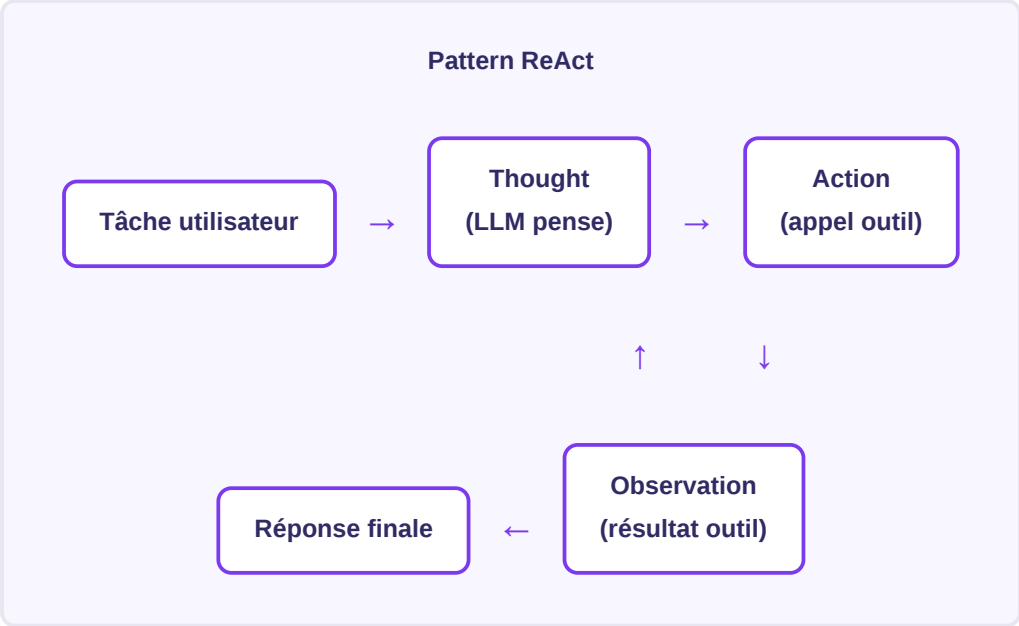
On distingue plusieurs types de mémoire :

Type	Durée	Implémentation
Contexte (court terme)	Une session	Historique des messages dans le prompt
Mémoire épisodique	Plusieurs sessions	Base de données (PostgreSQL, SQLite)
Mémoire sémantique	Permanente	Base vectorielle (Pinecone, pgvector)
Mémoire de travail	Une tâche	Variables Python, fichiers temporaires

5. La boucle d'exécution (ReAct)

La boucle d'exécution est le mécanisme qui orchestre tout. Le pattern le plus populaire s'appelle **ReAct** (Reasoning + Acting) :

1. **Thought** : le LLM réfléchit à l'état actuel et à ce qu'il doit faire
2. **Action** : il choisit un outil à appeler et les arguments à passer
3. **Observation** : le résultat de l'outil est injecté dans le contexte
4. **Répétition** jusqu'à ce que l'objectif soit atteint



Ce pattern simple est à la base de frameworks populaires comme **LangChain**, **LangGraph**, et **AutoGen**. Mais vous n'avez pas besoin d'un framework pour implémenter ReAct — quelques dizaines de lignes Python suffisent, comme nous le verrons au chapitre 4.

CHAPITRE 03

Choisir son modèle de langage

Le choix du modèle est l'une des décisions les plus importantes dans la conception d'un agent. Il n'existe pas de modèle universellement supérieur : chaque option présente des compromis entre performance, coût, latence, et confidentialité des données.

Les modèles propriétaires

OpenAI — GPT-4o et GPT-4o mini

Les modèles GPT-4o sont aujourd'hui la référence en termes de *function calling* et de suivie d'instructions complexes. GPT-4o mini offre un excellent rapport qualité/prix pour les tâches simples.

- **Points forts** : excellent function calling, écosystème mature, Assistants API intégrée
- **Points faibles** : coût élevé pour GPT-4o, données envoyées aux serveurs OpenAI
- **Prix indicatif** : ~$5/M tokens (input) pour GPT-4o, ~$0.15/M pour GPT-4o mini

Anthropic — Claude 3.5 Sonnet / Claude 3 Opus

Les modèles Claude excellent dans les tâches de raisonnement long et la gestion de documents complexes. Claude 3.5 Sonnet est souvent le meilleur choix pour les agents nécessitant une fenêtre de contexte étendue (200K tokens).

- **Points forts** : très long contexte, excellent pour analyser des documents, sécurité renforcée
- **Points faibles** : légèrement plus lent que GPT-4o sur certaines tâches
- **Prix indicatif** : ~$3/M tokens (input) pour Claude 3.5 Sonnet

Les modèles open-source

Meta — Llama 3 / 3.1 / 3.2

Llama 3 (70B et 405B) représente l'état de l'art open-source. Il peut être déployé sur vos propres serveurs, garantissant une confidentialité totale des données.

Mistral AI

Les modèles Mistral (7B, 8x7B Mixtral) sont particulièrement adaptés aux déploiements sur GPU modeste. Mistral propose aussi une API cloud compétitive.

Guide de sélection

Critère principal	Modèle recommandé	Raison
Performance maximale	GPT-4o ou Claude 3 Opus	Meilleur raisonnement général
Rapport qualité/prix	Claude 3.5 Sonnet ou GPT-4o mini	Excellent sur la plupart des tâches à 10× moins cher
Données confidentielles	Llama 3 auto-hébergé	Aucune donnée ne sort de votre infrastructure
Prototype rapide	GPT-4o mini	API simple, coût faible, bonne documentation
Documents longs	Claude 3.5 Sonnet	200K tokens de contexte, excellent sur PDF

ATTENTION AUX HALLUCINATIONS

Tous les LLMs peuvent inventer des informations (halluciner). Pour un agent en production, prévoyez toujours une validation des outputs critiques, et ne faites pas appeler des actions irréversibles (suppression, paiement) sans confirmation.

Stratégie multi-modèles

Une approche avancée consiste à utiliser **plusieurs modèles en parallèle** : un modèle rapide et bon marché pour les décisions simples, un modèle plus puissant pour les raisonnements complexes. Cette architecture, appelée *routing*, peut réduire les coûts de 60 à 80 % sur des agents en production.

CHAPITRE 04

Construire un premier agent pas à pas

Dans ce chapitre, nous allons construire un agent complet depuis zéro. Il sera capable de répondre à des questions en cherchant des informations sur le web et en effectuant des calculs. Nous n'utiliserons pas de framework — juste Python et l'API OpenAI.

Prérequis

```
pip install openai python-dotenv requests
```

Créez un fichier `.env` avec votre clé API :

```
OPENAI_API_KEY=sk-...
```

Structure du projet

```
agent/
├── .env
├── agent.py          # Boucle principale
├── tools.py          # Définition des outils
└── memory.py         # Gestion du contexte
```

Étape 1 — Définir les outils

```python
# tools.py
import requests
import json

def search_web(query: str) -> str:
    """Recherche des informations sur le web."""
    # Utilise l'API DuckDuckGo (gratuite, pas de clé requise)
    url = f"https://api.duckduckgo.com/?q=
```

```
{query}&format=json&no_html=1"
    response = requests.get(url, timeout=10)
    data = response.json()

    if data.get("AbstractText"):
        return data["AbstractText"][:500]
    elif data.get("RelatedTopics"):
        results = [t.get("Text", "") for t in data["RelatedTopics"]
[:3]]
        return "\n".join(results)
    return "Aucun résultat trouvé."

def calculate(expression: str) -> str:
    """Évalue une expression mathématique Python sécurisée."""
    try:
        allowed = set("0123456789+-*/(). ")
        if not set(expression).issubset(allowed):
            return "Expression non autorisée"
        result = eval(expression)
        return str(result)
    except Exception as e:
        return f"Erreur : {e}"

# Schémas des outils pour l'API OpenAI
TOOLS_SCHEMA = [
    {
        "type": "function",
        "function": {
            "name": "search_web",
            "description": "Recherche des informations actuelles sur
le web",
            "parameters": {
                "type": "object",
                "properties": {
                    "query": {
                        "type": "string",
                        "description": "La requête de recherche"
                    }
                },
                "required": ["query"]
            }
        }
    },
    {
        "type": "function",
        "function": {
            "name": "calculate",
            "description": "Effectue un calcul mathématique",
            "parameters": {
                "type": "object",
```

```
                "properties": {
                    "expression": {
                        "type": "string",
                        "description": "L'expression mathématique
Python à évaluer"
                    }
                },
                "required": ["expression"]
            }
        }
    }
]
```

Étape 2 — La boucle principale

```python
# agent.py
import os, json
from openai import OpenAI
from dotenv import load_dotenv
from tools import search_web, calculate, TOOLS_SCHEMA

load_dotenv()
client = OpenAI()

SYSTEM_PROMPT = """Tu es un assistant intelligent disposant d'outils.
Pour répondre, tu peux chercher sur le web et faire des calculs.
Pense d'abord à ce dont tu as besoin, puis utilise les outils
appropriés.
Donne des réponses concises et précises en français."""

def run_agent(user_message: str, max_iterations: int = 10):
    messages = [
        {"role": "system", "content": SYSTEM_PROMPT},
        {"role": "user", "content": user_message}
    ]

    for i in range(max_iterations):
        # Appel au LLM
        response = client.chat.completions.create(
            model="gpt-4o-mini",
            messages=messages,
            tools=TOOLS_SCHEMA,
            tool_choice="auto"
        )

        message = response.choices[0].message
        messages.append(message)
```

```
        # Si pas d'appel d'outil, c'est la réponse finale
        if not message.tool_calls:
            return message.content

        # Exécuter chaque appel d'outil
        for tool_call in message.tool_calls:
            fn_name = tool_call.function.name
            fn_args = json.loads(tool_call.function.arguments)

            print(f"→ Appel outil : {fn_name}({fn_args})")

            if fn_name == "search_web":
                result = search_web(**fn_args)
            elif fn_name == "calculate":
                result = calculate(**fn_args)
            else:
                result = f"Outil inconnu : {fn_name}"

            # Injecter le résultat dans le contexte
            messages.append({
                "role": "tool",
                "tool_call_id": tool_call.id,
                "content": result
            })

    return "Limite d'itérations atteinte."

if __name__ == "__main__":
    while True:
        question = input("\nVous : ")
        if question.lower() in ["quit", "exit"]:
            break
        answer = run_agent(question)
        print(f"\nAgent : {answer}")
```

Tester l'agent

Lancez l'agent avec `python agent.py` et posez-lui des questions :

- "Quelle est la population de Paris ?"
- "Si un produit coûte 49.99€ avec une réduction de 15%, quel est le prix final ?"
- "Qui a inventé Python ?"

Vous verrez l'agent afficher ses appels d'outils en temps réel avant de donner sa réponse finale.

CHAPITRE 05

Ajouter des outils — Function Calling

Le *function calling* est la mécanique centrale qui donne à un agent sa capacité à agir. Comprendre comment bien définir et implémenter des outils est crucial pour la fiabilité d'un agent.

Anatomie d'un bon outil

Un outil bien conçu doit respecter quatre principes :

1. **Nom clair et verbeux** : préférez `get_weather_for_city` à `weather`
2. **Description précise** : expliquez quand utiliser l'outil, pas seulement ce qu'il fait
3. **Paramètres typés** : utilisez les types JSON (string, number, boolean, array, object)
4. **Gestion d'erreurs robuste** : retournez toujours une string, même en cas d'erreur

```
# Mauvaise définition
{
  "name": "db",
  "description": "accès base de données"
}

# Bonne définition
{
  "name": "query_customer_database",
  "description": """Recherche des clients dans la base de données.
  Utilise cet outil quand l'utilisateur demande des infos sur un
client
  spécifique ou veut filtrer des clients par critères.""",
  "parameters": {
    "type": "object",
    "properties": {
      "filter": {
        "type": "string",
        "description": "Filtre SQL WHERE sans le mot-clé WHERE. Ex:
'email LIKE '%@gmail.com%'"
      },
      "limit": {
        "type": "integer",
```

```
        "description": "Nombre max de résultats (défaut: 10, max:
100)"
      }
    },
    "required": ["filter"]
  }
}
```

Outils avec état — gérer les effets de bord

Certains outils modifient le monde (envoyer un email, créer une commande, supprimer un fichier). Pour ces outils :

- Ajoutez `"⚠️ Action irréversible"` dans la description
- Prévoyez une étape de confirmation dans le prompt système
- Loggez systématiquement chaque appel avec son résultat
- Implémentez un *dry_run* mode pour les tests

```python
def send_email(to: str, subject: str, body: str, dry_run: bool =
False) -> str:
    """Envoie un email. ⚠️ Action irréversible."""
    if dry_run:
        return f"[DRY RUN] Email qui serait envoyé à {to}: {subject}"

    # Vraie logique d'envoi...
    import smtplib
    # ...
    return f"Email envoyé à {to}"
```

Outils parallèles vs séquentiels

Les LLMs modernes (GPT-4o, Claude 3) peuvent appeler **plusieurs outils en parallèle** dans une même réponse. C'est particulièrement utile pour des recherches indépendantes :

```python
# L'agent peut faire ça en un seul tour :
tool_calls = [
    {"name": "get_weather", "args": {"city": "Paris"}},
    {"name": "get_weather", "args": {"city": "Lyon"}},
    {"name": "get_weather", "args": {"city": "Marseille"}}
]
```

```python
# Exécuter en parallèle avec asyncio
import asyncio

results = await asyncio.gather(*[
    asyncio.to_thread(execute_tool, tc["name"], tc["args"])
    for tc in tool_calls
])
```

Bibliothèque d'outils utiles

Catégorie	Outil	Bibliothèque Python
Recherche web	DuckDuckGo, Tavily, Serper	`requests`, `tavily-python`
Fichiers	Lire/écrire PDF, Word, CSV	`pypdf2`, `python-docx`, `pandas`
Code	Exécuter du Python en sandbox	`subprocess`, `RestrictedPython`
Email	Lire/envoyer des emails	`imaplib`, `smtplib`, `sendgrid`
Base de données	SQL, recherche vectorielle	`psycopg2`, `sqlalchemy`, `chromadb`
APIs externes	Calendrier, Slack, GitHub…	`google-api-python-client`, `slack_sdk`

CHAPITRE 06

Gérer la mémoire et le contexte

La gestion de la mémoire est souvent sous-estimée lors des premiers prototypes, mais c'est ce qui fait la différence entre un agent qui "oublie" tout entre les sessions et un assistant vraiment utile au quotidien.

Le problème de la fenêtre de contexte

Les LLMs ont une limite stricte sur le nombre de tokens qu'ils peuvent traiter en une fois (de 8K à 200K selon les modèles). Si une conversation est trop longue, vous devez tronquer l'historique — et donc décider *quoi oublier*.

> **RÈGLE DES 80%**
>
> Ne laissez jamais le contexte dépasser 80% de la fenêtre maximale du modèle. Gardez les 20% restants pour la réponse et les appels d'outils. Avec GPT-4o (128K tokens), déclenchez la compression à ~100K tokens.

Stratégies de compression du contexte

1. Fenêtre glissante (simple)

Gardez toujours les N derniers messages. Simple mais vous perdez des informations potentiellement importantes du début de la conversation.

```
def trim_context(messages: list, max_messages: int = 20) -> list:
    system = [m for m in messages if m["role"] == "system"]
    conversation = [m for m in messages if m["role"] != "system"]
    return system + conversation[-max_messages:]
```

2. Résumé progressif (recommandé)

Périodiquement, demandez au LLM de résumer les échanges anciens en un bloc compact.

```python
async def summarize_old_messages(messages: list, keep_last: int = 10)
-> list:
    if len(messages) <= keep_last + 1:  # +1 pour le système
        return messages

    system = messages[0]
    old = messages[1:-keep_last]
    recent = messages[-keep_last:]

    summary = await llm.complete(
        f"Résume ces échanges en 3-5 phrases clés
:\n{format_messages(old)}"
    )

    summary_msg = {
        "role": "system",
        "content": f"[CONTEXTE ANTÉRIEUR] {summary}"
    }
    return [system, summary_msg] + recent
```

Mémoire persistante multi-sessions

Pour retenir des informations entre les sessions, vous avez besoin d'une **base de données externe**. Voici un exemple simple avec SQLite :

```python
# memory.py
import sqlite3
import json
from datetime import datetime

class AgentMemory:
    def __init__(self, db_path: str = "agent_memory.db"):
        self.conn = sqlite3.connect(db_path)
        self._init_db()

    def _init_db(self):
        self.conn.execute("""
            CREATE TABLE IF NOT EXISTS memories (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                session_id TEXT,
                role TEXT,
                content TEXT,
                timestamp TEXT
            )
        """)
```

```
        self.conn.commit()

    def save(self, session_id: str, messages: list):
        for msg in messages:
            self.conn.execute(
                "INSERT INTO memories VALUES (NULL, ?, ?, ?, ?)",
                (session_id, msg["role"],
                 json.dumps(msg["content"]),
                 datetime.utcnow().isoformat())
            )
        self.conn.commit()

    def load(self, session_id: str) -> list:
        cursor = self.conn.execute(
            "SELECT role, content FROM memories WHERE session_id=?
ORDER BY id",
            (session_id,)
        )
        return [{"role": r, "content": json.loads(c)}
                for r, c in cursor.fetchall()]
```

Mémoire sémantique (RAG)

Pour des agents qui doivent accéder à de grandes bases de connaissances (documents, FAQ, historique), le **RAG (Retrieval-Augmented Generation)** est la technique de référence :

1. Indexez vos documents sous forme de *embeddings* dans une base vectorielle
2. À chaque question, cherchez les passages les plus pertinents
3. Injectez ces passages dans le contexte avant de demander une réponse au LLM

Bibliothèques populaires : **LangChain** (tout-en-un), **LlamaIndex** (spécialisé documents), **ChromaDB** ou **pgvector** pour la base vectorielle.

CHAPITRE 07

Déploiement et bonnes pratiques

Passer d'un prototype local à un agent en production nécessite de répondre à plusieurs questions : où héberger, comment gérer les secrets, comment monitorer, et comment scaler.

Architecture de déploiement

Pour la plupart des agents, une architecture simple suffit :

- **API backend** : FastAPI ou Flask exposant un endpoint `POST /run-agent`
- **File de tâches** : Celery + Redis pour les tâches longues
- **Base de données** : PostgreSQL pour la mémoire et les logs
- **Hébergement** : Railway, Render, ou une VM chez OVH/Scaleway

```python
# api.py — FastAPI simple
from fastapi import FastAPI
from pydantic import BaseModel
from agent import run_agent

app = FastAPI()

class AgentRequest(BaseModel):
    message: str
    session_id: str = "default"

@app.post("/run")
async def run(req: AgentRequest):
    result = await run_agent(req.message, req.session_id)
    return {"response": result}
```

Gestion des secrets

NE JAMAIS COMMITTER VOS CLÉS API

> Ajoutez toujours `.env` à votre `.gitignore`. Utilisez les variables d'environnement de votre hébergeur (Railway, Heroku, Vercel…) pour les secrets de production.

Observabilité et monitoring

Un agent en production doit être observable. Loggez systématiquement :

- Chaque appel LLM : modèle, tokens consommés, latence, coût
- Chaque appel d'outil : nom, arguments, résultat, durée
- Les erreurs et les timeouts
- Les itérations par session (pour détecter les boucles infinies)

```
import logging
import time

logger = logging.getLogger("agent")

def log_tool_call(name: str, args: dict, result: str, duration:
float):
    logger.info({
        "event": "tool_call",
        "tool": name,
        "args": args,
        "result_len": len(result),
        "duration_ms": round(duration * 1000)
    })
```

Rate limiting et gestion des coûts

Les API LLM ont des limites de débit et sont facturées à l'usage. En production :

- Implémentez un *budget par utilisateur* (ex: max 50 appels/jour)
- Utilisez un *cache* pour les requêtes identiques (Redis avec TTL)
- Configurez des alertes si le coût journalier dépasse un seuil
- Testez votre agent avec `gpt-4o-mini` avant de passer à GPT-4o

Tests automatisés

Tester un agent est plus complexe que tester du code classique car les réponses LLM sont non-déterministes. Approche recommandée :

1. **Tests unitaires** des outils (sans LLM)
2. **Tests d'intégration** avec des stubs (réponses LLM prédéfinies)
3. **Tests de bout en bout** avec un sous-ensemble de cas réels et évaluation par LLM-as-judge

```
def test_calculate_tool():
    assert calculate("2 + 2") == "4"
    assert calculate("10 * 3.14") == "31.400000000000002"
    assert "Erreur" in calculate("import os")  # Sécurité
```

CHAPITRE 08

Erreurs courantes et checklist finale

Après avoir accompagné des dizaines d'équipes dans la construction d'agents IA, voici les erreurs les plus fréquentes — et comment les éviter.

Les 8 erreurs les plus courantes

1. Trop d'outils trop tôt

Donner 20 outils à un agent depuis le début le confuse et augmente les coûts. Commencez avec 3-5 outils essentiels. Ajoutez-en progressivement selon les besoins réels.

2. Des prompts système trop longs

Un prompt système de 5000 tokens dilue les instructions importantes. Visez 300-500 tokens maximum, avec des instructions précises et hiérarchisées.

3. Ignorer les erreurs d'outils

Si un outil retourne une erreur, l'agent doit le savoir pour changer de stratégie. Ne swallowez jamais les exceptions silencieusement — retournez un message d'erreur explicite.

4. Pas de limite d'itérations

Sans `max_iterations`, un agent peut boucler indéfiniment. Fixez toujours une limite (10-20 itérations pour la plupart des cas) et loggez quand elle est atteinte.

5. Ignorer les coûts en développement

GPT-4o à $5/M tokens peut coûter très cher durant le développement si vous ne surveillez pas. Utilisez GPT-4o mini pour les tests et ne passez au modèle fort qu'en production.

6. Pas de timeout sur les outils

Une requête HTTP qui ne répond jamais bloquera votre agent. Ajoutez toujours un timeout sur les appels réseau et une gestion propre du TimeoutError.

7. Confondre "l'agent pense" et "l'agent agit"

Certains modèles génèrent du texte de "réflexion" avant d'agir (chain-of-thought). Ce texte ne doit pas être exécuté. Parsez soigneusement les appels d'outils et ignorez le texte libre.

8. Pas de validation des outputs critiques

Pour des actions critiques (envoi d'email, paiement, modification de base de données), validez toujours la réponse de l'agent avant d'exécuter l'action. Utilisez un deuxième LLM pour la validation si nécessaire.

> **PATTERN DE VALIDATION**
>
> Avant d'exécuter une action irréversible, demandez au LLM : "Es-tu sûr à 100% que cette action est correcte ? Réponds par OUI ou NON, suivi d'une explication." N'exécutez que si la réponse commence par OUI.

Checklist de mise en production

- ☐ Tous les outils ont un timeout configuré
- ☐ Une limite d'itérations est fixée dans la boucle
- ☐ Les clés API sont dans des variables d'environnement, pas dans le code
- ☐ Les appels LLM et outils sont loggés avec timestamp et durée
- ☐ Un budget de tokens/coût par utilisateur est implémenté
- ☐ Les tests unitaires des outils passent à 100%
- ☐ Un test bout-en-bout sur 5 scénarios représentatifs est validé
- ☐ Les erreurs retournent des messages explicites (pas des stack traces)
- ☐ Le prompt système a été validé sur au moins 50 requêtes variées
- ☐ Un mécanisme de fallback existe si le LLM principal est indisponible
- ☐ Le monitoring d'erreurs est configuré (Sentry, DataDog, ou simple email)

- [ ] La documentation des outils est à jour et accessible à l'équipe

Pour aller plus loin

Une fois votre premier agent en production, voici les sujets à explorer :

- **Agents multi-étapes** : LangGraph pour des workflows complexes avec parallélisme
- **Agents multi-modèles** : orchestration de plusieurs LLMs spécialisés
- **Human-in-the-loop** : mécanismes de validation et d'interruption humaine
- **Évaluation systématique** : benchmarks, red-teaming, et amélioration continue
- **Agents autonomes** : AutoGPT, BabyAGI, et leurs successeurs

> **LA RÈGLE D'OR**
>
> Commencez simple. Un agent avec 3 outils bien définis et un prompt clair surpassera toujours un agent complexe avec 20 outils mal documentés. Itérez à partir de ce qui fonctionne.

— Fin du guide —