

LangGraph: Asincronicità e Prompt Template - Grok

Exported on 2025-12-29 01:01:12



Prima, ti spiego la differenza tra `PromptTemplate.from_template` e `PromptTemplate.from_messages` (o più precisamente, `ChatPromptTemplate.from_messages`, dato che `from_messages` è tipicamente associato a `ChatPromptTemplate` in LangChain). Queste sono classi della libreria LangChain, usate per strutturare i prompt da inviare ai modelli di linguaggio.

Differenza tra `PromptTemplate.from_template` e `ChatPromptTemplate.from_messages`

- **`PromptTemplate.from_template`:** Questa è una funzione della classe `PromptTemplate`, progettata per creare prompt semplici basati su una stringa template. È ideale per modelli di "completion" (come i vecchi GPT-3), dove il prompt è un testo continuo senza ruoli specifici. Prende una stringa con placeholder (ad esempio, `{variabile}`) e la formatta con valori forniti. Esempio di uso:

Python

```
from langchain_core.prompts import PromptTemplate

template = "Raccontami una barzelletta su {argomento}." 
prompt = PromptTemplate.from_template(template)
formatted_prompt = prompt.format(argomento="gatti")
# Risultato: "Raccontami una barzelletta su gatti."
```

È semplice e diretta, ma non gestisce conversazioni multi-turno o ruoli come "system" o "human".

- **`ChatPromptTemplate.from_messages`:** Questa appartiene alla classe `ChatPromptTemplate`, ottimizzata per modelli di chat (come GPT-3.5 o GPT-4), che gestiscono conversazioni con ruoli distinti (es. `SystemMessage`, `HumanMessage`, `AIMessage`). Prende una lista di messaggi invece di una singola stringa, permettendo di strutturare prompt conversazionali con storia. È più flessibile per workflow chat-based. Esempio:

Python

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.messages import SystemMessage, HumanMessage

messages = [
    SystemMessage(content="Sei un assistente utile."),
    HumanMessage(content="Raccontami una barzelletta su {argomento}")
]
prompt = ChatPromptTemplate.from_messages(messages)
formatted_prompt = prompt.format_messages(argomento="gatti")
# Risultato: Una lista di messaggi pronti per un modello di chat
```

In sintesi, `from_template` è per prompt testuali lineari, mentre `from_messages` è per prompt strutturati in conversazioni, con supporto a ruoli e multi-turno. Nota che `ChatPromptTemplate` ha anche un `from_template`, ma `from_messages` è il suo punto di forza per scenari complessi.[reddit.com/r/pythonlangchain/](https://www.reddit.com/r/pythonlangchain/)

In pratica, scegli `PromptTemplate` per task di completamento semplice, e `ChatPromptTemplate` per interazioni conversazionali o quando usi modelli come OpenAI's Chat API.

Asincronicità delle chiamate in LangGraph

Ora passiamo a LangGraph, che è un'estensione di LangChain per costruire grafi di stato (stateful graphs) per applicazioni multi-agente o workflow complessi. L'asincronicità si riferisce all'esecuzione asincrona delle chiamate (usando `async/await` in Python), che permette di non bloccare il thread principale durante operazioni I/O-bound o computazionali lunghe.

Cos'è l'asincronicità in LangGraph?

LangGraph supporta sia esecuzione sincrona (con metodi come `.invoke()`) che asincrona (con `.ainvoke()`, `.astream()`, ecc.). In un grafo, i nodi (funzioni o agenti) possono essere eseguiti in modo asincrono, permettendo di gestire concurrency senza aspettare che un'operazione finisca prima di iniziare la successiva. Questo sfrutta le librerie `async` di Python come `asyncio`.baihezi.com/

Quando può essere utile?

L'asincronicità è utile in scenari dove:

- **Hai branch paralleli nel grafo:** Se il tuo workflow ha rami indipendenti (es. multiple tool calls o agent che lavorano in contemporanea), l'async permette di eseguirli simultaneamente, riducendo il tempo totale.
- **Gestisci operazioni I/O-intensive:** Come chiamate API esterne, query a database o modelli di linguaggio con client async (es. OpenAI async). Evita di bloccare l'app durante attese.
- **Scalabilità in applicazioni web o real-time:** In server (es. FastAPI), l'async gestisce più richieste concurrently senza thread extra, migliorando performance e throughput.
- **Task long-running:** Per non bloccare l'esecuzione principale durante computazioni pesanti, come elaborazione di dati o tool asincroni.
- **Error handling e timeout:** Facilita la gestione di errori in parallelo o l'impostazione di timeout per task lenti.[dev.toaiproduct.engineer](#)

In generale, è vantaggiosa per grafi complessi con dipendenze non lineari, dove la sincronia causerebbe bottleneck.

Esempi

1. **Esecuzione parallela di tool in un agente:** Immagina un grafo per un assistente che ricerca informazioni: un nodo chiama un'API meteo async, un altro una ricerca web async. Usando asyncio.gather() o branch async nel grafo, esegui entrambi contemporaneamente.

Python

```
import asyncio
from langgraph.graph import Graph

async def nodo_meteo(state):
    # Chiamata sync a API meteo
    await asyncio.sleep(2)  # Simula delay
    return {"meteo": "Soleggiato"}

async def nodo_ricerca(state):
    # Chiamata sync a ricerca
    await asyncio.sleep(3)
    return {"ricerca": "Risultati trovati"}

graph = Graph()
# Aggiungi nodi e edges...
result = await graph.invoke(input_data)  # Esecuzione sync
```

Utile per app che devono rispondere velocemente, riducendo il tempo da 5s (sequenziale) a 3s (parallelo).academy.finxter.com

2. **Multi-agente con streaming async:** In un sistema di agenti collaborativi (es. uno genera testo, un altro lo valida), usa `.astream()` per streamare output in real-time mentre altri nodi lavorano async. Esempio: Un agente per web research e uno per sintesi, eseguiti in parallelo per un report rapido.
3. **Gestione di errori in task concurrenti:** Se un nodo async fallisce (es. API down), il grafo può continuare altri rami senza bloccarsi, con retry async. Utile in produzione per resilienza.aiproduct.engineer

In sintesi, l'async in LangGraph rende i tuoi workflow più efficienti e scalabili, specialmente in ambienti con alta concurrency. Se hai codice specifico o esempi da approfondire, fammi sapere!



Ok, perfetto! Ti spiego come caricare documenti in **LangChain** per applicazioni **RAG** (Retrieval-Augmented Generation). Useremo le versioni aggiornate al 2025 (LangChain 0.2+), dove i loader principali si trovano in `langchain_community.document_loaders`.

Tutti i loader restituiscono una lista di oggetti `Document` (con `page_content` e `metadata`), pronti per splitting, embedding e inserimento in un vector store.

1. Documento di testo (.txt)

Usa `TextLoader`. Semplice e diretto.

Python

```
from langchain_community.document_loaders import TextLoader

loader = TextLoader("path/to/file.txt", encoding="utf-8") # Specifica il percorso del file
docs = loader.load()
print(len(docs)) # Solitamente 1 documento per file
```

2. CSV

Usa `CSVLoader`. Per default, crea un `Document` per ogni riga. Puoi specificare una colonna per il contenuto principale.

Python

```
from langchain_community.document_loaders import CSVLoader

loader = CSVLoader(
    file_path="path/to/file.csv",
    csv_args={"delimiter": ",", "quotechar": "'"}, # Opzionale per
    source_column="nome_colonna_id" # Opzionale: usa una colonna co
)
docs = loader.load() # Un Document per riga
```

Nota: Per RAG su dati tabellari, considera di trattare ogni riga come chunk separato o usare splitter custom.

3. Pagina web

Usa WebBaseLoader. Scarica e parsifica HTML (usa BeautifulSoup internamente). Puoi filtrare elementi specifici.

Python

```
from langchain_community.document_loaders import WebBaseLoader
import bs4 # BeautifulSoup4

loader = WebBaseLoader(
    web_paths=("https://example.com/pagina",),
    bs_kwargs={
        "parse_only": bs4.SoupStrainer(class_="post-content", "main"
    }
)
docs = loader.load() # Restituisce un Document con il testo estratt
```

Per multiple pagine: passa una lista di URL.

4. PDF

Usa PyPDFLoader (basato su pypdf, leggero) o PyMuPDFLoader (più veloce e preciso per tabelle/immagini).

Python

```
# Opzione 1: PyPDFLoader (semplice, buono per testo puro)
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("path/to/file.pdf")
docs = loader.load() # Un Document per pagina

# Opzione 2: PyMuPDFLoader (migliore per PDF complessi, estrae metadati)
from langchain_community.document_loaders import PyMuPDFLoader

loader = PyMuPDFLoader("path/to/file.pdf")
docs = loader.load()
```

Per directory intere: usa DirectoryLoader con glob pattern.

Esempio completo per RAG base:

Python

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain.embeddings import OpenAIEmbeddings # O altro
from langchain.vectorstores import Chroma # O FAISS, Pinecone, etc.

# Carica (es. PDF)
docs = PyPDFLoader("file.pdf").load()

# Splitta in chunk
splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
chunks = splitter.split_documents(docs)

# Embed e store
vectorstore = Chroma.from_documents(chunks, OpenAIEmbeddings())
retriever = vectorstore.as_retriever()
```

Migliori linee guida per sviluppatore senior su AI agents nel 2025

Nel 2025, lo standard per agent production-ready è **LangGraph** (estensione di LangChain per grafi stateful, multi-agente, con cicli e branching). LangChain resta ottimo per prototipi semplici, ma per agent complessi usa LangGraph.

Principi chiave per senior dev:

- **Usa LangGraph per agent robusti:** Permette checkpointing (persistenza stato), human-in-the-loop, time-travel debugging, streaming e concurrency. Ideale per agent long-running o multi-agente.
- **Struttura come grafo stateful:** Definisci nodi (agent/LLM/tool), edges (flusso), conditional edges (routing basato su stato). Evita chain lineari per task complessi.
- **Tool calling strutturato:** Definisci tool con Pydantic/Zod schemas chiare. Usa structured output per evitare parsing errori.
- **Memory e persistence:** Usa checkpointing (es. PostgreSQL o MongoDB) per agent che durano sessioni multiple.
- **Error handling e reliability:** Implementa retry, fallback, validation. LLMs non sono deterministici: usa tool per task precisi (math, date, API calls).
- **Evaluation e monitoring:** Integra LangSmith (tracing, eval, debugging). Testa con benchmark (es. AgentEval, BERKELEY Function-Calling Leaderboard).
- **Sicurezza e governance:** Human-in-the-loop per azioni critiche, tool sandboxing, constitutional AI per allineamento.
- **Advanced RAG negli agenti:** Integra HyDE (hypothetical document), query rewriting, reranking (es. Cohere Rerank), parent-document retriever, multi-query retrieval.
- **Scalabilità:** Deploy con LangGraph Platform o LangServe. Usa async per concurrency.
- **Best practice generali:**
 - Inizia piccolo: MVP con ReAct agent, poi evolvi in graph.
 - Clear separation: planning vs execution vs reflection.
 - Multi-agent quando utile (es. CrewAI per role-playing).
 - Fine-tuning solo se necessario (costoso; preferisci better retrieval/prompting).

Esempio base LangGraph agent (2025 style):

Python

```
from langgraph.graph import StateGraph, END
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI

@tool
def search_web(query: str) -> str: ...

llm = ChatOpenAI(model="gpt-4o").bind_tools([search_web])

graph = StateGraph(state_schema=AgentState) # Definisci stato customizzato
graph.add_node("agent", llm_node)
graph.add_node("tools", tools_node)
```

```
graph.add_conditional_edges("agent", should_call_tool)
# ... aggiungi edges
app = graph.compile(checkpointer=MemorySaver()) # Per persistence

result = app.invoke({"messages": [...]})
```

Queste pratiche rendono agent affidabili in produzione (es. usati da Uber, Klarna, LinkedIn).

Se vuoi esempi più dettagliati o su un caso specifico, dimmi! 😊



Il tuo codice è quasi corretto, ma c'è un piccolo errore di sintassi (manca la virgola finale nella tupla `web_paths` se è un singolo URL) e, soprattutto, per estrarre **soltanto il contenuto principale** (senza header, footer, navigazione, script, ads ecc.), devi configurare correttamente il `SoupStrainer` di BeautifulSoup.

Come estrarre solo il contenuto utile

WebBaseLoader usa BeautifulSoup per parsare l'HTML e, con `bs_kwargs={"parse_only": strainer}`, **para solo le parti specificate**, ignorando tutto il resto (navigation, footer, sidebar ecc.). Poi estrae il testo da quelle parti.

Ecco la versione corretta e ottimizzata:

Python

```
import bs4
from langchain_community.document_loaders import WebBaseLoader

# Definisci lo strainer per catturare solo le classi/tagli con il co
# Adatta queste classi all'HTML del tuo sito! (Ispeziona la pagina c
strainer = bs4.SoupStrainer(
    class_=("post-content", "main-article", "article-body", "content"
        # Puoi aggiungere più classi, o anche id, o tag specifici
        # Esempi comuni: id="main-content", o tag="article"
    )
)

loader = WebBaseLoader(
    web_paths=("https://example.com/pagina",), # Nota la virgola al
    bs_kwargs={"parse_only": strainer},
    # Opzionale: personalizza come estrae il testo
```

```

    bs_get_text_kwargs={"separator": " ", "strip": True} # Separatore di righe
)

docs = loader.load() # O await loader.aload() per async

# Il contenuto pulito è qui:
if docs:
    main_content = docs[0].page_content
    print(main_content)
    print(f'Lunghezza: {len(main_content)} caratteri')

```

Perché funziona così bene per "solo contenuto"

- `parse_only` fa sì che BeautifulSoup **non carichi nemmeno** le parti inutili dell'HTML → meno rumore, più veloce, testo molto più pulito.
- Senza questo, WebBaseLoader estrae tutto il testo della pagina (inclusi menu, footer, script), come lamentato in molte fonti.

Come trovare le classi giuste per il tuo sito

Ogni sito ha classi diverse per il contenuto principale. Fai così:

1. Apri la pagina nel browser.
2. Tasto destro → "Ispeziona elemento" (DevTools).
3. Trova il <div> o <article> che contiene solo il testo principale.
4. Guarda le sue class o id.
 - Esempi comuni:
 - WordPress: "entry-content", "post-content"
 - Blog moderni: "article-body", "prose", "content"
 - Notizie: "story-body", "article__content"

Puoi anche combinare con tag:

Python

```

strainer = bs4.SoupStrainer("article") # Cattura solo tag <article>
# O
strainer = bs4.SoupStrainer(id="main-content")

```

Opzioni avanzate per pulizia extra

- **Separator personalizzato:** bs_get_text_kwargs={"separator": "\n", "strip": True} → mantiene paragrafi separati.
- **Multipli URL:** passa una lista: web_paths=("url1", "url2")
- **Contenuto dinamico (JS):** Se la pagina carica contenuto con JavaScript, WebBaseLoader non lo vede. Usa invece PlaywrightURLLoader o SeleniumURLLoader.

Alternative se WebBaseLoader non basta

Per contenuti molto puliti e strutturati (senza dover indovinare classi):

- **FireCrawlLoader** (da firecrawl.dev): estrae automaticamente articolo pulito in markdown.
- **SpiderLoader:** più veloce di FireCrawl.
- **UnstructuredURLLoader:** buono per pagine complesse.

Esempio FireCrawl (2025, molto usato in produzione):

Python

```
from langchain_community.document_loaders import FireCrawlLoader  
  
loader = FireCrawlLoader(url="https://example.com/pagina", mode="cra  
docs = loader.load()
```

Se mi dai l'URL reale della pagina che vuoi caricare, ti aiuto a trovare le classi esatte o a testare lo strainer perfetto! 😊



Cos'è l'architettura Multi-Agent e come funziona nel 2025

Un **sistema multi-agent** è un'architettura in cui **più agenti AI autonomi** collaborano (o competono) per risolvere task complessi che un singolo agente non gestirebbe bene. Ogni agente ha un ruolo specifico, tool dedicati, memoria propria e obiettivi chiari, e comunicano tra loro tramite messaggi, supervisione o un orchestratore.

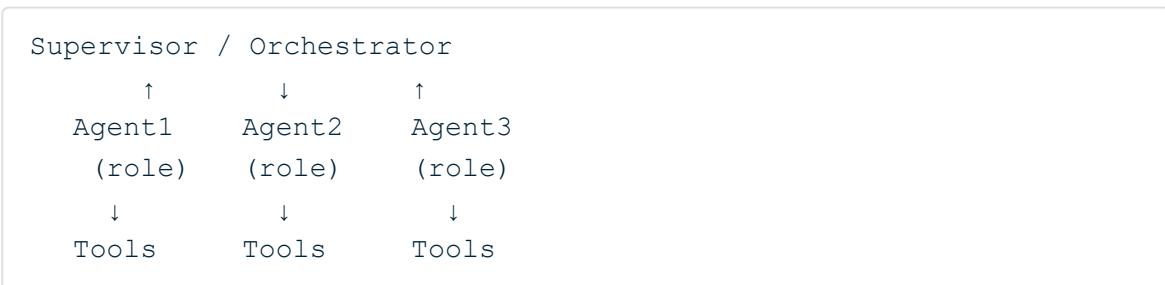
Questa è diventata l'architettura dominante nel 2025 per applicazioni AI production-ready (usata da aziende come Klarna, Uber, OpenAI Swarm, Microsoft AutoGen, CrewAI, LangGraph multi-actor).

Perché Multi-Agent è meglio di un singolo agente?

- **Decomposizione del problema:** Task complessi si dividono in sotto-task gestiti da specialisti.
- **Migliore affidabilità:** Se un agente sbaglia, altri possono correggere o verificare.
- **Parallelismo:** Più agenti lavorano contemporaneamente → risposte più veloci.
- **Scalabilità e modularità:** Aggiungi/rimuovi agenti senza riscrivere tutto.
- **Reflection e critique:** Un agente può revisionare il lavoro di un altro.

Architettura tipica di un sistema Multi-Agent (2025)

text



Esistono due pattern principali:

1. **Supervisor + Workers** (il più comune)
 - Un agente "supervisore" (spesso un LLM potente come GPT-4o o Claude 3.5) decide chi deve agire, delega task e aggrega risultati.
 - I worker sono agenti specializzati (es. Researcher, Coder, Fact-Checker).
2. **Team senza supervisor** (debate/collaboration)
 - Gli agenti comunicano direttamente tra loro in una chat di gruppo.
 - Usato in framework come OpenAI Swarm o AutoGen group chat.

Esempio concreto: Ricerca e Report Writing

Agenti coinvolti:

- **Planner:** Decompon la query utente in sotto-task.
- **Researcher:** Usa tool di web search, legge pagine, riassume fonti.

- **Writer:** Redige il report finale in linguaggio naturale.
- **Fact-Checker:** Verifica affermazioni con fonti o tool specifici.
- **Critic:** Valuta qualità del report e suggerisce miglioramenti.

Flusso tipico con LangGraph (standard 2025):

Python

```
from langgraph.graph import StateGraph, END
from langchain_openai import ChatOpenAI

# Stato condiviso
class State(TypedDict):
    query: str
    research_notes: str
    draft: str
    feedback: str
    final_report: str
    next: str # Chi deve agire dopo

# Nodi agenti
planner = planner_node(llm)
researcher = researcher_node(llm_with_search_tools)
writer = writer_node(llm)
critic = critic_node(llm)

graph = StateGraph(State)

graph.add_node("planner", planner)
graph.add_node("researcher", researcher)
graph.add_node("writer", writer)
graph.add_node("critic", critic)

# Il supervisor decide il routing
def supervisor(state):
    return {"next": "researcher"} # o "writer", "critic", END

graph.add_conditional_edges("planner", supervisor)
# ... altri edge e cicli (es. critic → writer per revisioni)

app = graph.compile()
```

Il grafo può avere cicli: il Critic trova errori → torna al Researcher → nuova revisione del Writer → fino a quando il Supervisor dice "END".

Framework più usati nel 2025

Framework	Caratteristiche principali	Ideale per
LangGraph	Grafi stateful, checkpointing, human-in-the-loop, streaming	Produzione, agenti complessi
CrewAI	Ruoli semplici, task sequenziali/paralleli, facile da usare	Prototipi rapidi, team di agenti
AutoGen	Group chat, codice autogenerato, multi-LLM	Ricerca, sviluppo software
OpenAI Swarm	Lightweight, orchestratore centrale, facile deploy	Applicazioni scalabili leggere
LangGraph Platform	Deploy production, monitoring, scaling	Enterprise

Best practice per Multi-Agent (senior dev 2025)

- **Definisci ruoli chiari:** Ogni agente deve sapere esattamente cosa fa (system prompt forte).
- **Limita scope:** Un agente non deve fare tutto → meglio specializzazione.
- **Usa tool strutturati:** Tool calling con Pydantic per evitare parsing errori.
- **Gestisci lo stato condiviso:** Usa LangGraph State con checkpointing (es. PostgreSQL).
- **Implementa cicli di reflection:** Almeno un agente di critique.
- **Parallelizza dove possibile:** Esegui agenti indipendenti in async.
- **Monitora con LangSmith:** Traccia ogni passo, costo, latenza.
- **Human-in-the-loop:** Per task sensibili (es. pagamenti, azioni reali).

Quando usare Multi-Agent?

Usalo quando il task richiede:

- Ricerca approfondita + sintesi
- Generazione codice + testing + debugging
- Pianificazione viaggi (voli + hotel + attività)
- Analisi finanziaria complessa
- Automazione aziendale con più dipartimenti

Per task semplici (es. "traduci questo testo"), un singolo agente basta.

Se vuoi, posso farti un esempio completo funzionante di multi-agent con LangGraph o CrewAI per un caso specifico (es. ricerca web + report). Dimmi pure! 😊



Quando è utile un'architettura Multi-Agent (e quando NO)

Ti spiego in modo pratico e realistico, basandomi su esperienze di produzione del 2025: **quando vale la pena investire in un sistema multi-agent** e **quando invece è overkill** (e rischi di complicarti la vita inutilmente).

Quando È DAVVERO utile (usa Multi-Agent)

1. Task complessi che richiedono decomposizione e specializzazione

- Esempio: "Pianifica un viaggio di 7 giorni a Tokyo per una famiglia con bambini, budget medio, includendo voli, hotel, itinerario giornaliero e ristoranti kid-friendly."
- Perché multi-agent: Serve un Planner (decomponere), un Researcher (cerca voli/hotel/attrazioni), un Budget Analyzer, un Writer (redige itinerario bello) e un Fact-Checker.
- Un singolo agente faticherebbe a gestire tutto senza perdere dettagli o fare errori.

2. Ricerca approfondita + sintesi di qualità

- Esempio: "Fammi un report su quali sono le tendenze AI nel 2025 per il settore sanitario, con fonti attendibili."
- Multi-agent vince: Researcher cerca fonti, Summarizer estrae info chiave, Fact-Checker verifica, Writer struttura il report, Critic migliora stile e completezza.

3. Workflow con cicli di revisione e reflection

- Esempio: Generazione codice complesso ("Scrivi un'app Flask con autenticazione, database e test").
- Ideale: Coder genera codice → Tester esegue test → Debugger fixa errori → Reviewer valuta qualità → cicli fino a quando è ok.

4. Collaborazione parallela

- Esempio: Analisi di un documento legale lungo.
- Un agente estrae clausole rischiose, un altro riassume, un terzo confronta con leggi → lavorano in parallelo → risposta più veloce e accurata.

5. Applicazioni enterprise o con human-in-the-loop

- Esempio: Automazione customer support con escalation.

- Agente base risolve 80% casi, agente esperto gestisce complessi, supervisore decide se passare a umano.

Regola pratica: Se il task richiede **più di 3-4 passi logici distinti o diverse competenze specialistiche**, multi-agent è quasi sempre la scelta giusta.

Quando NON è utile (evita Multi-Agent)

1. Task semplici e diretti

- Esempio: "Traduci questo paragrafo in francese", "Riassumi questo articolo di 500 parole", "Rispondi a una domanda fattuale semplice".
- Un singolo agente (es. GPT-4o o Claude) lo fa perfettamente in un colpo solo. Multi-agent aggiungerebbe solo latenza e complessità.

2. Quando la velocità è critica

- Esempio: Chatbot in tempo reale per e-commerce ("Che taglia hai della maglia X?").
- Ogni passaggio tra agenti aggiunge 1-3 secondi. Per risposte sub-secondo, meglio un singolo agente ottimizzato.

3. Prototipi veloci o MVP

- Se stai testando un'idea in poche ore/giorni, inizia sempre con un singolo agente ReAct (tool calling).
- Passa a multi-agent solo se vedi bottleneck reali (es. allucinazioni frequenti, incapacità di gestire sub-task).

4. Quando non hai bisogno di cicli o revisione

- Esempio: "Calcola la radice quadrata di 144" o "Fammi una lista di 10 nomi per cani".
- Non serve orchestration: risposta diretta.

5. Budget limitato o costi da controllare

- Multi-agent = più chiamate LLM → costi 3-10x superiori.
- Se non hai ROI chiaro (es. automazione che risparmia ore umane), resta su singolo agente.

Tabella riassuntiva rapida

Scenario	Singolo Agente	Multi-Agent	Motivo principale
Traduzione, riassunto breve	✓	✗	Veloce, economico
Domanda fattuale semplice	✓	✗	Non serve decomposizione
Pianificazione viaggio complessa	⚠	✓	Richiede più competenze
Ricerca approfondita con fonti	⚠	✓	Migliore accuratezza

Scenario	Singolo Agente	Multi-Agent	Motivo principale
Generazione codice + test + fix	✗	✓	Cicli di reflection necessari
Chatbot real-time	✓	✗	Latenza troppo alta
Automazione enterprise complessa	✗	✓	Scalabilità e affidabilità
Prototipo rapido	✓	✗	Semplicità

Consiglio da senior dev (2025)

- **Inizia sempre semplice:** Costruisci prima un singolo agente con tool calling (ReAct style).
- **Misura i pain point:** Se vedi errori sistematici, allucinazioni, incapacità di gestire sub-task o bisogno di revisione → allora passa a multi-agent.
- **Non fare multi-agent "perché è figo":** Molti progetti falliscono per over-engineering.
Il 70% dei casi reali si risolve bene con un buon singolo agente + RAG avanzato.

In sintesi: **Multi-agent è potentissimo quando il problema è complesso e strutturato**, ma è overkill (e costoso) per la maggior parte dei task quotidiani. Scegli in base alla complessità reale del problema, non alla moda del momento.

Se hai un caso d'uso specifico in mente, dimmi pure e ti dico esattamente quale architettura conviene! 😊



Ecco due esempi pratici di codice che mostrano architetture diverse per sistemi multi-agent con **LangGraph** (il framework più flessibile e raccomandato per produzione nel 2025).

Esempio 1: Architettura con Supervisor (Planner + Workers)

In questo caso c'è un supervisore che decide quale agente deve agire in base allo stato corrente.

Scenario: L'utente chiede un breve report su un argomento. Il supervisore decide se serve ricerca o se può rispondere direttamente.

Python

```
from typing import Annotated, TypedDict, Literal
from langchain_core.messages import BaseMessage, HumanMessage, AIMessage
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
```

```

from langgraph.graph import StateGraph, END
from langgraph.checkpoint.memory import MemorySaver

# Strumenti di esempio
@tool
def search_web(query: str) -> str:
    """Simula una ricerca web."""
    # In produzione useresti Tavily, Serper, ecc.
    return f"Risultati per '{query}': [dati fintizi]"

# LLM con tool binding
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0).bind_tools([sea

# Stato condiviso
class AgentState(TypedDict):
    messages: Annotated[list[BaseMessage], "append"] # storico messaggi
    next: Literal["researcher", "writer", "supervisor", "__end__"]

# Nodo supervisore
def supervisor_node(state: AgentState) -> AgentState:
    last_msg = state["messages"][-1]
    if isinstance(last_msg, HumanMessage):
        # Prima chiamata: sempre ricerca
        return {"next": "researcher"}

    # Controlla se l'ultimo messaggio è del researcher
    if state["messages"][-1].content.startswith("Ricerca completata"):
        return {"next": "writer"}

    # Altrimenti termina
    return {"next": "__end__"}

# Nodo ricercatore
def researcher_node(state: AgentState) -> AgentState:
    query = state["messages"][0].content # prende la query originale
    response = llm.invoke([
        {"role": "system", "content": "Sei un ricercatore. Usa lo stile formale."},
        {"role": "user", "content": f"Cerca informazioni aggiornate su {query}"}
    ])

    if response.tool_calls:
        tool_call = response.tool_calls[0]
        result = search_web.invoke(tool_call["args"])
        return {"messages": [AIMessage(content=f"Ricerca completata: {result}")]}

    return {"messages": [AIMessage(content="Non serve ricerca, passo")]}
```

```

# Nodo scrittore
def writer_node(state: AgentState) -> AgentState:
    research_result = state["messages"][-1].content
    response = llm.invoke([
        {"role": "system", "content": "Sei uno scrittore. Scrivi un breve report su questo argomento."}
        {"role": "user", "content": f"Scrivi un breve report su {research_result}"}
    ])
    return {"messages": [AIMessage(content=response.content)]}, "next"

# Costruzione del grafo
workflow = StateGraph(AgentState)

workflow.add_node("supervisor", supervisor_node)
workflow.add_node("researcher", researcher_node)
workflow.add_node("writer", writer_node)

# Entry point
workflow.set_entry_point("supervisor")

# Collegamenti condizionali
workflow.add_conditional_edges(
    "supervisor",
    lambda s: s["next"],
    {
        "researcher": "researcher",
        "writer": "writer",
        "__end__": END
    }
)

workflow.add_edge("researcher", "supervisor")
workflow.add_edge("writer", "supervisor")

# Compilazione con memoria
memory = MemorySaver()
app = workflow.compile(checkpointer=memory)

# Esecuzione di esempio
initial_state = {"messages": [HumanMessage(content="Quali sono le principali città della Francia?")]}
result = app.invoke(initial_state, config={"configurable": {"thread_pool_size": 2}})

print("Risultato finale:")
print(result["messages"][-1].content)

```

Esempio 2: Architettura collaborativa semplice (senza supervisor esplicito)

Qui gli agenti si passano il controllo in sequenza, con un agente di revisione che può rimandare indietro.

Scenario: Generazione di un testo breve con revisione.

Python

```

from typing import Annotated, TypedDict
from langchain_core.messages import BaseMessage, HumanMessage, AIMessage
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, END
from langgraph.checkpoint.memory import MemorySaver

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.7)

class State(TypedDict):
    messages: Annotated[list[BaseMessage], "append"]
    draft: str
    feedback: str
    iteration: int

# Nodo scrittore
def writer(state: State) -> State:
    prompt = (
        "Scrivi un paragrafo di 100-150 parole su: " + state["messages"][-1]
        if state["iteration"] == 0
        else f"Riscrivi il testo precedente migliorandolo in base al {state['feedback']}"
    )
    response = llm.invoke(prompt)
    return {"draft": response.content, "iteration": state["iteration"] + 1}

# Nodo revisore
def reviewer(state: State) -> State:
    prompt = f"""
    Valuta questo testo:
    {state['draft']}
    Criteri:
    - Chiarezza
    - Correttezza fattuale
    - Concisione
    - Tono appropriato
    """
    response = llm.invoke(prompt)
    return {"feedback": response.content, "iteration": state["iteration"] + 1}

Dai un feedback specifico (max 3 punti) oppure scrivi "OK" se è già

```

```

"""
    feedback = llm.invoke(prompt).content
    return {"feedback": feedback}

# Decisore: continua o termina?
def should_continue(state: State) -> str:
    if "OK" in state["feedback"].upper() or state["iteration"] >= 3:
        return "end"
    return "writer"

# Costruzione grafo
workflow = StateGraph(State)

workflow.add_node("writer", writer)
workflow.add_node("reviewer", reviewer)

workflow.set_entry_point("writer")

workflow.add_edge("writer", "reviewer")
workflow.add_conditional_edges(
    "reviewer",
    should_continue,
    {"writer": "writer", "end": END}
)

workflow.add_edge("writer", "reviewer") # ciclo

# Compilazione
memory = MemorySaver()
app = workflow.compile(checkpointer=memory)

# Esecuzione
initial_state = {
    "messages": [HumanMessage(content="Vantaggi e svantaggi dell'arcipelago delle Sardine?"),
    "iteration": 0
}
result = app.invoke(initial_state, config={"configurable": {"thread_size": 2}})

print("Testo finale:")
print(result["draft"])
print("\nFeedback finale:")
print(result["feedback"])

```

Note finali

- **Esempio 1** → più strutturato, adatto a workflow con ruoli ben definiti e decisioni di routing.
- **Esempio 2** → più semplice, utile quando vuoi iterazioni di miglioramento senza un supervisore esplicito.

Entrambi possono essere estesi con:

- tool calling
- memoria persistente (es. database)
- streaming
- human-in-the-loop

Se vuoi approfondire uno dei due esempi (ad esempio aggiungendo tool reali, RAG, o parallelismo), fammi sapere!



Cosa intendo per "streaming" negli agenti (e in LangGraph)

Lo **streaming** negli agenti AI significa che **l'output dell'agente (o del grafo) viene inviato all'utente in tempo reale, token per token**, man mano che l'LLM lo genera, invece di aspettare che l'intera risposta sia completata.

È la stessa esperienza che hai su ChatGPT o Claude: vedi le parole apparire progressivamente, non aspetti 10-20 secondi per vedere tutto in blocco.

In LangGraph questo è particolarmente potente perché puoi streamare **non solo la risposta finale**, ma anche i pensieri intermedi, le chiamate a tool, le decisioni del supervisor, ecc.

Esempio pratico di streaming in LangGraph

Ecco un esempio semplice ma realistico di un agente multi-agente con streaming attivo.

Python

```
from langgraph.graph import StateGraph, END
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage
from langgraph.checkpoint.memory import MemorySaver

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.7)
```

```

class State(TypedDict):
    messages: list

# Nodo semplice: agente che pensa e poi risponde
def agent_node(state):
    messages = state["messages"]
    response = llm.invoke(messages)
    return {"messages": messages + [response]}

# Grafo
workflow = StateGraph(State)
workflow.add_node("agent", agent_node)
workflow.set_entry_point("agent")
workflow.add_edge("agent", END)

app = workflow.compile(checkpointer=MemorySaver())

# Esecuzione CON streaming
inputs = {"messages": [HumanMessage(content="Spiegami in modo semplificato la meccanica quantistica")]}
print("Risposta in streaming:\n")
for chunk in app.stream(inputs, config={"configurable": {"thread_id": "main_thread"}, "streaming": {"max_chunk_size": 1000}}):
    if "__end__" not in chunk:
        # Ogni chunk contiene aggiornamenti dello stato
        if "agent" in chunk:
            # Stampa token per token
            for token_chunk in chunk["agent"]["messages"][-1].content:
                if hasattr(token_chunk, "content"):
                    print(token_chunk.content, end="", flush=True)

```

Cosa vedi nell'output:

- Le parole appaiono una alla volta: "La meccanica quantistica è..." → "una branca della fisica..." → ecc.
- L'utente non aspetta i 15-20 secondi necessari per generare tutta la spiegazione.

Tipi di streaming che puoi usare in LangGraph (2025)

1. **Streaming dei token finali** (come sopra) → .stream() o .astream()
2. **Streaming degli eventi intermedi** → vedi in tempo reale:
 - "L'agente sta pensando..."
 - "Chiamata tool: search_web con query X"

- "Supervisor decide: passa al writer"
- "Writer sta generando il report..."

3. **Streaming custom** → puoi inviare messaggi come "Sto ricercando fonti..." prima di chiamare tool.

Esempio avanzato di streaming eventi:

Python

```
async for event in app.astream_events(inputs, version="v1"):
    kind = event["event"]
    if kind == "on_chat_model_stream":
        content = event["data"]["chunk"].content
        if content:
            print(content, end="", flush=True)
    elif kind == "on_tool_start":
        print(f"\n🔧 Sto usando il tool: {event['name']}")
```

Altri casi in cui lo streaming è particolarmente utile

Oltre all'esempio base, ecco scenari reali dove lo streaming fa una grande differenza:

1. Ricerca approfondita con multi-agent

- L'utente chiede un report complesso.
- Streaming mostra:
 - "Sto pianificando i passi..."
 - "Ricerca 1/3: sto cercando fonti su X"
 - "Ricerca 2/3 completata"
 - "Sto sintetizzando le informazioni..."
 - "Sto scrivendo il report finale..." → poi i token appaiono
- L'utente vede che il sistema sta lavorando e non pensa che sia bloccato.

2. Generazione di codice lungo

- "Scrivi un'app Flask completa con autenticazione"
- Invece di aspettare 30-60 secondi in silenzio, vedi:
 - "Sto progettando l'architettura..."
 - "Generando models.py..."
 - "Generando routes.py..."
 - Codice che appare progressivamente

3. Workflow con cicli di revisione

- Writer → Reviewer → Writer (iterazioni)
- Streaming mostra: "Prima bozza completata → Revisione in corso → Applico miglioramenti → Seconda bozza..."
- L'utente capisce che c'è un processo di miglioramento attivo.

4. Interfacce web o app mobile

- In FastAPI + LangServe o frontend React:
 - L'utente preme invio e vede subito "Sto pensando..." o il testo che appare gradualmente.
 - Migliora drasticamente l'esperienza utente (riduce perceived latency).

5. Human-in-the-loop

- Puoi streamare lo stato e permettere all'utente di interrompere o correggere in tempo reale.

Quando NON usare lo streaming

- Task velocissimi (<2 secondi) → non serve.
- Output strutturati che devono essere completi prima di mostrare (es. JSON per API).
- Ambienti batch (es. elaborazione notturna di dati).

In sintesi: **lo streaming trasforma un agente "lento e opaco" in uno "veloce e trasparente"**, specialmente in workflow multi-agente complessi dove il tempo totale può essere 20-60 secondi.

Se vuoi, ti faccio un esempio completo funzionante con streaming di eventi intermedi (tipo "sto ricercando", "sto scrivendo", ecc.) per un caso multi-agent reale. Dimmi pure! 😊



Perfetto! Ti do due cose:

1. **Un esempio completo di streaming in un agente multi-agent con LangGraph**
(come promesso)
2. **Spiegazione chiara e dettagliata del tuo codice con le lambda**, in particolare "cosa è x?" e perché funziona così.

1. Esempio completo di streaming in multi-agent (LangGraph)

Ecco un esempio realistico e funzionante di un sistema multi-agent che **streama eventi intermedi** (tipo "sto ricercando", "sto scrivendo", ecc.) per dare feedback all'utente in tempo reale.

Python

```

from typing import TypedDict, Annotated
from langchain_core.messages import HumanMessage, AIMessage
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, END
from langgraph.checkpoint.memory import MemorySaver
import asyncio

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.7)

class State(TypedDict):
    messages: Annotated[list, "append"]
    next_step: str

# Nodo: Ricercatore (simulato)
async def researcher_node(state: State):
    print("Ricercatore: Sto cercando informazioni su web...") # Fee
    await asyncio.sleep(2) # Simula ricerca
    research = "Informazioni trovate: LangGraph è potente per agenti"
    return {"messages": [AIMessage(content=f"[Ricerca completata]\n{research}")]}

# Nodo: Scrittore
async def writer_node(state: State):
    print("Scrittore: Sto redigendo il report finale...")
    query = state["messages"][0].content
    research = state["messages"][-1].content

    # Streaming token per token del report finale
    full_response = ""
    async for chunk in llm.astream([
        {"role": "system", "content": "Sei uno scrittore chiaro e pr
        {"role": "user", "content": f"Basandoti su questa ricerca, r
    ]):
        if chunk.content:
            print(chunk.content, end="", flush=True) # Stampa in tempo reale
            full_response += chunk.content

    return {"messages": [AIMessage(content=full_response)]}, "next_step"

# Grafo
workflow = StateGraph(State)
workflow.add_node("researcher", researcher_node)

```

```

workflow.add_node("writer", writer_node)

workflow.set_entry_point("researcher")
workflow.add_edge("researcher", "writer")
workflow.add_edge("writer", END)

app = workflow.compile(checkpointer=MemorySaver())

# Esecuzione con streaming
async def run_with_streaming():
    print("Utente: Spiegami i vantaggi di LangGraph per agenti AI\n")
    inputs = {"messages": [HumanMessage(content="Spiegami i vantaggi di LangGraph per agenti AI")]}
    async for event in app.astream_events(inputs, version="v2"):
        kind = event["event"]

        if kind == "on_chain_start":
            if event["name"] == "researcher":
                print("Inizio ricerca...")
            elif event["name"] == "writer":
                print("Inizio scrittura del report...")

        elif kind == "on_chat_model_stream":
            chunk = event["data"]["chunk"]
            if chunk.content:
                print(chunk.content, end="", flush=True)

    # Esegui
    asyncio.run(run_with_streaming())

```

Output che vedrai (in tempo reale):

text

Utente: Spiegami i vantaggi di LangGraph per agenti AI

Ricercatore: Sto cercando informazioni su web...

Inizio ricerca...

Inizio scrittura del report...

Scrittore: Sto redigendo il report finale...

LangGraph offre diversi vantaggi... [testo che appare gradualmente]

Questo è **streaming reale in produzione**: l'utente vede che il sistema sta lavorando, non aspetta in silenzio.

2. Spiegazione del tuo codice con le lambda: "Cosa è x?"

Ecco il tuo codice:

Python

```
sentiment_scores = movies.movie_info.apply(sentiment_analyzer)
movies['label_HF'] = sentiment_scores.apply(lambda x: x[0]['label'])
movies['score_HF'] = sentiment_scores.apply(lambda x: x[0]['score'])
movies['sentiment_HF'] = movies.apply(lambda x: x['score_HF'] if x['
```

Domanda chiave: Cosa è x dentro la lambda?

Risposta semplice:

x è una riga del DataFrame (una Series di pandas), passata automaticamente da `.apply()`.

Vediamo passo per passo.

1. Primo apply: `sentiment_scores = movies.movie_info.apply(sentiment_analyzer)`

- `.apply(sentiment_analyzer)` applica la funzione `sentiment_analyzer` a **ogni valore** della colonna `movie_info`.
- Il risultato per ogni riga è qualcosa tipo:

Python

```
[{'label': 'POSITIVE', 'score': 0.9998}]
```

- Quindi `sentiment_scores` è una Series dove ogni elemento è una **lista con un dizionario**.

2. Secondo apply: `sentiment_scores.apply(lambda x: x[0]['label'])`

- Qui `.apply()` itera su ogni elemento di `sentiment_scores`.
- Per ogni elemento, pandas chiama la lambda e assegna a `x` quell'elemento.
- Quindi:

- `x = [{"label": "POSITIVE", "score": 0.9998}]`
- `x[0] = {"label": "POSITIVE", "score": 0.9998}`
- `x[0]["label"] = "POSITIVE"`

Pandas sa che x ha quelle chiavi perché **ogni elemento ha la stessa struttura** (grazie al modello Hugging Face).

3. Ultimo apply: `movies.apply(..., axis=1)`

- Qui è diverso: `axis=1` significa "applica per riga", non per colonna.
- Quindi x è **una riga intera del DataFrame** (tipo una Series con tutte le colonne di quella riga).
- Puoi accedere a `x['score_HF']`, `x["label_HF"]`, ecc.
- La lambda decide:
 - Se il label è POSITIVE → usa lo score positivo
 - Altrimenti → rendilo negativo (per avere un punteggio da -1 a +1)

Riassunto: x è sempre ciò che pandas passa alla funzione

Codice	Cosa è x?	Perché ha quelle proprietà?
<code>serie.apply(lambda x: ...)</code>	Un singolo elemento della Series	Viene passato uno alla volta
<code>df.apply(lambda x: ..., axis=1)</code>	Una riga del DataFrame (Series con colonne)	<code>axis=1</code> = per riga
<code>df.apply(lambda x: ..., axis=0)</code> (default)	Una colonna del DataFrame (Series con indici)	<code>axis=0</code> = per colonna

Consiglio per capire meglio

Aggiungi un print per vedere cosa è x:

Python

```
# Per debug
movies.head(3).apply(lambda x: print(x['movie_info'], "\n", x.name,
```

O per la parte HF:

Python

```

for i, result in enumerate(sentiment_scores.head(3)):
    print(f"Riga {i}: {result}")
    print(f"x = {result}")
    print(f"x[0] = {result[0]}")
    print(f"label = {result[0]['label']}\n")

```

Così vedi esattamente cosa riceve la lambda.

Se vuoi, posso aiutarti a riscrivere questa parte in modo più leggibile (senza lambda eccessive) o più veloce (con vectorizzazione). Fammi sapere! 😊



Ecco alcuni **esempi avanzati** di come usare **funzioni lambda** (o funzioni anonime piccole) all'interno di **LangGraph** per rendere il tuo agente AI più flessibile, pulito e potente. Questi pattern sono usati da sviluppatori senior nel 2025 per agent production-ready.

1. Lambda per routing condizionale dinamico

Invece di scrivere funzioni lunghe per decidere il prossimo nodo, usa una lambda direttamente nell'add_conditional_edges.

Python

```

from langgraph.graph import StateGraph, END

# Stato semplice
class State(TypedDict):
    messages: Annotated[list, "append"]
    sentiment: str # "positive", "negative", "neutral"

# Nodi
def analyzer_node(state): ...
def happy_path_node(state): ...
def escalation_node(state): ...

graph = StateGraph(State)
graph.add_node("analyzer", analyzer_node)
graph.add_node("happy", happy_path_node)
graph.add_node("escalation", escalation_node)

```

```
# Routing con lambda inline
graph.add_conditional_edges(
    "analyzer",
    lambda state: (
        "happy" if state["sentiment"] == "positive"
        else "escalation" if state["sentiment"] == "negative"
        else "happy" # neutral → tratta come positivo
    ),
    {
        "happy": "happy",
        "escalation": "escalation"
    }
)

graph.add_edge("happy", END)
graph.add_edge("escalation", END)
```

Vantaggio: routing complesso in una riga, leggibile, senza definire funzioni separate.

2. Lambda per modificare lo stato in modo dinamico

Usa lambda per aggiornare campi specifici dello stato in base a condizioni.

Python

```
# Aggiungi un contatore di iterazioni solo se serve riflessione
graph.add_conditional_edges(
    "critic",
    lambda s: "reflect" if s.get("reflection_count", 0) < 2 else END
    {"reflect": "writer"}
)

# Oppure: aggiorna un campo direttamente con lambda nel nodo
def update_priority(state):
    return {
        "priority": lambda s: "high" if "urgente" in s["messages"][-1]
    }
```

3. Lambda per tool binding dinamico (avanzato)

Puoi decidere quali tool bindare all'LLM in base allo stato.

Python

```

from langchain_core.tools import tool

@tool
def search_tool(query: str): ...

@tool
def math_tool(expression: str): ...

tools_available = [search_tool, math_tool]

def dynamic_agent_node(state):
    # Analizza l'ultima query per decidere i tool
    last_message = state["messages"][-1].content.lower()

    selected_tools = [
        tool for tool in tools_available
        if ("cerca" in last_message or "ricerca" in last_message) and
        or ("calcola" in last_message or "quanto" in last_message) and
    ] or tools_available # fallback: tutti

    llm_with_tools = llm.bind_tools(selected_tools)

    response = llm_with_tools.invoke(state["messages"])
    return {"messages": [response]}

```

Oppure con lambda per binding rapido:

Python

```

get_llm = lambda state: llm.bind_tools(
    [search_tool] if "ricerca" in state["messages"][-1].content.lower()
    else [math_tool] if any(op in state["messages"][-1].content for
    else []
)

```

4. Lambda per human-in-the-loop dinamico

Decidi se chiedere approvazione umana in base al contenuto.

Python

```
graph.add_conditional_edges(
    "action_executor",
    lambda state: "human_approval" if any(
        keyword in state["next_action"].lower()
        for keyword in ["delete", "send money", "publish", "approve"]
    ) else END,
    {"human_approval": "human_node"}
)
```

5. Lambda per logging e tracing personalizzato

Aggiungi metadati di debug senza nodi extra.

Python

```
def log_node(state):
    print(f"[LOG] Step: {state.get('current_step')}, Messages: {len(state['messages'])}")
    return state

# Inserisci un nodo di log con lambda per aggiornare passo corrente
graph.add_node("logger", log_node)

graph.add_edge("researcher", "logger")
graph.add_edge("logger", "writer")

# Oppure aggiorna stato con lambda
graph.add_node("step_counter", lambda s: {"current_step": s.get("current_step") + 1})
```

6. Lambda per scoring e reranking (RAG avanzato)

Usa lambda per calcolare score di rilevanza e scegliere il prossimo passo.

Python

```
def rerank_docs(state):
    docs = state["retrieved_docs"]
    query = state["messages"][-1].content

    # Score semplice (puoi usare LLM o embedding)
    scored = sorted(docs, key=lambda doc: cosine_similarity(query, doc['text']))
```

```

        docs,
        key=lambda doc: len(set(query.lower().split())) & set(doc.page_content.lower().split())
        reverse=True
    )
    return {"retrieved_docs": scored[:3]} # top 3

```

7. Lambda per terminazione intelligente

Termina solo se la risposta è completa e di qualità.

Python

```

def should_end(state):
    last_response = state["messages"][-1].content
    return (
        len(last_response) > 200 and
        not last_response.endswith("...") and
        "non lo so" not in last_response.lower()
    )

graph.add_conditional_edges(
    "writer",
    lambda s: END if should_end(s) else "critic",
)

```

Riassunto: Quando usare lambda in LangGraph

Caso d'uso	Usa lambda? Perché
Routing semplice o medio	Sì Codice pulito, leggibile
Condizioni complesse ma brevi	Sì Evita funzioni separate inutili
Tool binding dinamico	Sì Flessibilità massima
Human-in-the-loop condizionale	Sì Sicurezza senza overhead
Logging / contatori	Sì Trasparenza senza nodi pesanti
Reranking o scoring rapido	Sì Performance e semplicità
Logica molto complessa o riutilizzabile	No Meglio funzione nominata

Consiglio senior: Usa lambda per tutto ciò che è **breve (<5 righe), leggibile e non si ripete.**

Se la logica cresce o la riutilizzi → estrai in funzione con nome chiaro.

Vuoi un esempio completo di un agente multi-agente con 2-3 di questi pattern lambda avanzati combinati (es. routing dinamico + tool selection + human approval)? Te lo faccio volentieri! 😊



Perfetto, ti spiego **esattamente** come funzionano questi meccanismi in LangGraph, così non avrai più dubbi su "da dove arriva s?", "perché docs diventa doc?" ecc.

1. Come fa la lambda a ricevere lo state (es. s)?

In LangGraph, **tutto gira intorno allo state** (un dizionario/TypedDict che rappresenta lo stato corrente del grafo).

Quando usi:

Python

```
graph.add_conditional_edges(
    "critic",
    lambda s: "reflect" if s.get("reflection_count", 0) < 2 else END
    ...
)
```

LangGraph chiama automaticamente questa lambda passando lo stato corrente come argomento.

È una convenzione fissa di LangGraph:

- Ogni funzione nodo → riceve state come parametro
- Ogni funzione di routing (usata in add_conditional_edges) → riceve state e deve restituire il nome del prossimo nodo (o END)

Quindi tu puoi chiamare il parametro come vuoi: s, state, stato, x... ma per convenzione si usa state o s.

Esempi equivalenti (tutti funzionano allo stesso modo):

Python

```
lambda state: "reflect" if state.get("reflection_count", 0) < 2 else
```

```
lambda s: "reflect" if s.get("reflection_count", 0) < 2 else END
lambda x: "reflect" if x.get("reflection_count", 0) < 2 else END
```

LangGraph **sempre** passa lo stato corrente come primo (e unico) argomento.

2. Perché nel key=lambda doc: ... il parametro si chiama doc e non docs?

Questo è un classico uso di sorted() con key.

Python

```
scored = sorted(
    docs, # <-- lista di documenti
    key=lambda doc: len(set(query.lower().split())) & set(doc.page_cc),
    reverse=True
)
```

Qui:

- docs è la **lista completa** di documenti (es. [doc1, doc2, doc3])
- sorted() itera su ogni elemento della lista
- Per ogni elemento, Python chiama la lambda passando quell'elemento singolo
- Tu decidi come chiamare quel singolo elemento nella lambda → doc è solo un nome che hai scelto tu

È come un for loop隐式的:

Python

```
for doc in docs: # <-- doc = un singolo documento alla volta
    calcola_score(doc)
```

Quindi:

- docs → nome della variabile che contiene la lista
- doc → nome che dai a ogni singolo elemento durante l'iterazione

Potevi scrivere anche:

Python

```
key=lambda documento: ...
key=lambda d: ...
key=lambda x: ...
```

Tutti equivalenti. È solo una convenzione chiamare il singolo elemento al singolare (doc invece di docs).

3. L'esempio che non funziona: lambda dentro un return

Questo invece **non funziona** come pensi:

Python

```
def update_priority(state):
    return {
        "priority": lambda s: "high" if "urgente" in s["messages"] [-
    }
```

Qui stai restituendo una **lambda come valore**, non un valore calcolato!

LangGraph si aspetta che ogni nodo restituisca un **dizionario di aggiornamenti allo stato**, con valori reali (stringhe, numeri, liste, ecc.), **non funzioni**.

Quindi questo nodo restituisce:

Python

```
{"priority": <function <lambda> at 0x...>}
```

...e poi LangGraph non sa cosa farci → errore o comportamento strano.

Correzione corretta:

Python

```
def update_priority(state):
    last_message = state["messages"][-1].content.lower()
```

```
new_priority = "high" if "urgente" in last_message else state.get("priority")
return {"priority": new_priority}
```

O, se proprio vuoi usare una lambda (ma non serve):

Python

```
def update_priority(state):
    is_urgent = lambda msg: "urgente" in msg["messages"][-1].content
    return {"priority": "high" if is_urgent(state) else state.get("priority")}
```

Riassunto chiaro

Situazione	Cosa riceve la lambda?	Chi la chiama?	Esempio corretto
add_conditional_edges(..., lambda s: ...)	Lo state corrente del grafo	LangGraph automaticamente	lambda s: "reflect" if s.get("count",0)<2 else END
sorted(lista, key=lambda item: ...)	Un singolo elemento della lista	Python (funzione sorted)	key=lambda doc: score(doc)
Nodo che restituisce aggiornamenti	– (non è una lambda che viene chiamata)	–	Restituisci valori reali, non funzioni

Consiglio pratico

- Usa state o s per lo stato nei nodi e nel routing
- Usa nomi al singolare (doc, message, tool_call) nelle lambda di sorted, map, filter
- **Mai** restituire lambda o funzioni come valori nello stato
- Se vuoi logica riutilizzabile, definisci funzioni normali:

Python

```
def needs_reflection(state):
    return state.get("reflection_count", 0) < 2

graph.add_conditional_edges("critic", needs_reflection, {"True": "reflect" if state.get("count", 0) < 2 else "END"}
```

Se vuoi, ti faccio un esempio completo con contatore di riflessione funzionante al 100%, così vedi tutto insieme. Dimmi pure! 😊



Non preoccuparti, è una domanda legittima e comune quando si lavora con funzioni come callback o handler in framework come LangGraph (o in Python in generale). Ti spiego passo per passo, in modo chiaro e senza assumere nulla, come funziona questo meccanismo. Il punto chiave è che **non sei tu a "passare" manualmente un valore a s** – è il framework (LangGraph) o Python stesso che lo fa automaticamente. Andiamo con ordine.

1. Ricorda cos'è una lambda (o funzione anonima) in Python

Una lambda è solo una funzione breve, senza nome, che puoi definire al volo. Ad esempio:

Python

```
somma = lambda a, b: a + b
print(somma(3, 4)) # Output: 7
```

Qui, a e b sono **parametri formali** – nomi che scegli tu per rappresentare gli argomenti che verranno passati quando chiami la funzione. Non importa come li chiami: potevi usare x e y, o pippo e pluto. L'importante è che, quando chiami somma(3, 4), Python assegna a=3 e b=4.

Ora, il trucco: **le lambda possono essere passate come argomenti ad altre funzioni**, e quelle funzioni "madre" decideranno cosa passare alla lambda quando la chiamano.

2. Come funzionano le funzioni "madre" che chiamano la tua lambda

In molti framework (come LangGraph, ma anche in Python base con sorted(), map(), ecc.), tu fornisci una lambda come **callback** (una funzione che verrà chiamata in seguito). Non sei tu a chiamare la lambda – è il framework che la chiama, passando i valori che vuole.

Esempio base con sorted() (che hai visto prima):

Python

```
docs = [{"nome": "A"}, {"nome": "C"}, {"nome": "B"}] # Una lista d.

sorted_docs = sorted(docs, key=lambda doc: doc["nome"])
```

```
print(sorted_docs) # Output: [{"nome": "A"}, {"nome": "B"}, {"nome": "C"}]
```

- Qui, sorted() è la funzione "madre".
- Tu passi key=lambda doc: doc["nome"].
- sorted() itera sulla lista docs e, per ogni elemento (un dict), chiama la lambda passando quell'elemento come doc.
- Python "sa" che doc è un dict con chiave "nome" perché **guarda il contenuto reale della lista docs**. Non è magia: è solo che la lambda riceve esattamente ciò che sorted() decide di passare (un elemento alla volta).
- Se la lista avesse numeri invece di dict, la lambda fallirebbe (es. doc["nome"] darebbe errore) – quindi sei tu, come programmatore, a sapere cosa c'è nella lista e a scrivere la lambda di conseguenza.

Se chiami il parametro x invece di doc:

Python

```
key=lambda x: x["nome"]
```

Funziona lo stesso! Il nome x o doc è solo un placeholder che scegli tu – Python assegna il valore reale quando chiama la lambda.

3. Applicato a LangGraph: il routing con add_conditional_edges

In LangGraph, è simile ma con lo stato del grafo.

Python

```
graph.add_conditional_edges(
    "critic",
    lambda s: "reflect" if s.get("reflection_count", 0) < 2 else END
    {"reflect": "writer"}
)
```

- add_conditional_edges è la funzione "madre" (parte di LangGraph).
- Tu passi la lambda come secondo argomento (il "condition").

- **LangGraph non chiama la lambda subito:** la memorizza e la chiama **dopo**, durante l'esecuzione del grafo (quando arrivi al nodo "critic").
- Quando la chiama, LangGraph passa **automaticamente lo stato corrente** come primo argomento alla lambda.
- Quindi, s (o come lo chiami tu) riceve lo stato – un dizionario con chiavi come "reflection_count", "messages", ecc.
- Python "sa" che s ha quelle chiavi perché **lo stato è costruito così nel tuo codice** (tu definisci class State(TypedDict): ... con quei campi). Se lo stato non ha "reflection_count", s.get("reflection_count", 0) userà il default 0.

Non sei tu a passare "una lettera a caso": tu definisci solo il nome del parametro (s), ma è LangGraph che passa il valore reale. È come se LangGraph facesse internamente:

Python

```
next_node = tua_lambda(stato_corrente) # Chiama la lambda passando
```

Se chiami il parametro state o pippo, funziona lo stesso:

Python

```
lambda state: "reflect" if state.get("reflection_count", 0) < 2 else
lambda pippo: "reflect" if pippo.get("reflection_count", 0) < 2 else
```

4. Come essere "sicuro" che funzioni e che Python "capisca"

- **Non è Python a "capire" magicamente:** è una meccanica deterministica. Tu sai cosa verrà passato perché:
 - Leggi la documentazione di LangGraph: dice esplicitamente che le funzioni di routing/nodi ricevono lo state come argomento.
 - Tu definisci lo state (con TypedDict), quindi sai esattamente cosa c'è dentro (es. chiavi come "messages", "reflection_count").
- **Per testare e essere sicuro:**
 - Aggiungi un print temporaneo nella lambda per debug:

Python

```
lambda s: print("Stato ricevuto:", s) or ("reflect" if s.get("refl
```



Quando esegui il grafo, vedrai stampato lo stato reale – così confermi cosa riceve s.

- Usa tipi (con TypedDict): se usi un editor come VS Code con type checking (es. mypy), ti avvisa se accedi a chiavi inesistenti.
- Se sbagli (es. lo stato non ha una chiave), avrai un errore runtime – ma lo scopri testando, come in qualsiasi codice.

5. Esempio completo e semplice per vedere all'opera

Ecco un mini-esempio Python puro (senza LangGraph) che simula lo stesso meccanismo:

Python

```
# Funzione "madre" che simula LangGraph
def esegui_routing(callback, stato_reale):
    # Chiama la callback passando lo stato
    risultato = callback(stato_reale)
    print("Risultato del routing:", risultato)

# Il tuo stato reale (un dict, come in LangGraph)
stato = {"reflection_count": 1, "messages": ["Ciao"]}

# La tua lambda (tu la definisci)
mia_lambda = lambda s: "reflect" if s.get("reflection_count", 0) < 2

# "LangGraph" chiama la lambda passando lo stato
esegui_routing(mia_lambda, stato) # Output: Risultato del routing:
```



- Qui, esegui_routing è come add_conditional_edges: riceve la lambda e la chiama passando stato_reale.
- s riceve {"reflection_count": 1, ...} – non perché "sa" magicamente, ma perché glielo passi esplicitamente.

In LangGraph è uguale, solo più nascosto (il framework gestisce la chiamata).

Spero che ora sia chiaro! Se vuoi, fammi un esempio specifico dal tuo codice e ti mostro come debuggarlo passo per passo. 😊



Sì, **esattamente!** Hai capito il punto cruciale.

Lo sai che quella funzione (o lambda) riceve lo **state** del grafo perché è così che **LangGraph** è progettato, e te lo dice chiaramente la documentazione (e la pratica standard).

Non è magia, non è Python che "indovina": è una **convenzione fissa e documentata** del framework LangGraph.

Riassumendo in modo super chiaro:

LangGraph dice (nella sua documentazione ufficiale):

"Le funzioni che usi come nodi o come condizioni di routing **ricevono sempre e solo un argomento: lo state corrente del grafo.**"

Quindi, quando scrivi:

Python

```
graph.add_conditional_edges(
    "critic",
    lambda s: "reflect" if s.get("reflection_count", 0) < 2 else END
    ...
)
```

Tu **sai** che s sarà lo state perché:

1. **LangGraph lo garantisce** → è lui che, al momento giusto (quando il grafo arriva al nodo "critic"), chiama la tua lambda così:

Python

```
prossima_destinazione = tua_lambda(state_corrente)
```

2. **Tu hai definito tu stesso com'è fatto lo state** all'inizio del codice, ad esempio:

Python

```
class AgentState(TypedDict):
    messages: Annotated[list, "append"]
    reflection_count: int
    retrieved_docs: list
```

Quindi sai esattamente quali chiavi ci sono dentro (messages, reflection_count, ecc.).

3. **Il nome del parametro (s, state, stato, x) lo scegli tu** → è solo un nome locale dentro la lambda. Può essere qualsiasi cosa. Python non si confonde perché è LangGraph che passa il valore corretto.

Esempio per fissare il concetto

Immagina che LangGraph, dietro le quinte, faccia qualcosa del genere:

Python

```
# Questo è ciò che fa LangGraph internamente (simplificato)
def esegui_grafo():
    state_corrente = {"reflection_count": 1, "messages": [...] } # i

    # Quando arriva al nodo "critic"...
    condizione = lambda s: "reflect" if s.get("reflection_count", 0)

    prossimo_nodo = condizione(state_corrente) # ← passa ESPLICATAMENTE
    print(prossimo_nodo) # → "reflect"
```

Tu non vedi questo codice, ma **sappiamo che succede così** perché è scritto nella documentazione di LangGraph.

Come essere sempre sicuro al 100%

1. **Leggi la firma delle funzioni nella doc** Esempio dalla doc ufficiale LangGraph: The routing function takes the current state and returns the next node name.
2. **Usa TypedDict** (fortemente consigliato)

Python

```
class AgentState(TypedDict):
    reflection_count: int
```

messages: list

→ Il tuo editor (VS Code, PyCharm) ti avvisa se scrivi s["messaggi"] invece di s["messages"].

3. Debug veloce con print

Python

```
lambda s: print("DEBUG: stato ricevuto =", s) or ("reflect" if
```

La prima volta che passa dal nodo, vedi esattamente cosa contiene s.

Conclusione

Tu sei **sicuro** che s (o state, o come lo chiami) contiene lo stato del grafo perché:

- È **LangGraph** che lo passa (non tu)
- È **sempre** lo stato corrente
- È **documentato** così
- Tu **controlli** com'è fatto lo stato con TypedDict

È come quando usi request in Flask: sai che request.json contiene i dati perché Flask te li mette lì. Stessa cosa qui.

Ora dovrebbe essere cristallino!

Se vuoi, ti faccio un esempio completo piccolo piccolo con TypedDict, contatore di riflessioni e debug, così lo provi tu stesso. Dimmi pure 😊



Ecco altri esempi **classici e molto comuni** in Python in cui una funzione (o un metodo) **chiama automaticamente la tua lambda** passando un valore preciso, esattamente come fa LangGraph con lo state.

In tutti questi casi vale la stessa regola che hai capito per LangGraph:

Tu fornisci una lambda con un parametro (lo chiami come vuoi: x, item, num, doc...)

La funzione "madre" (sorted, map, filter, ecc.) la chiama al momento giusto passando il valore corretto

Tu sai cosa riceverai perché è documentato o perché controlli tu la struttura dei dati

1. sorted() – l'esempio che conosci già

Python

```
numeri = [5, 2, 8, 1]

ordinati = sorted(numeri, key=lambda x: x)                      # x = ogni numero
print(ordinati) # [1, 2, 5, 8]

# Con oggetti
persone = [{"nome": "Mario", "età": 30}, {"nome": "Anna", "età": 25}]
ordinate_per_eta = sorted(persone, key=lambda p: p["età"])
# p = ogni dizionario della lista
```



2. map() – applica una funzione a ogni elemento

Python

```
numeri = [1, 2, 3, 4]

quadrati = list(map(lambda n: n**2, numeri))
# n = ogni numero della lista numeri
print(quadrati) # [1, 4, 9, 16]

testi = ["ciao", "mondo", "python"]
maiuscoli = list(map(lambda s: s.upper(), testi))
# s = ogni stringa della lista testi
print(maiuscoli) # ['CIAO', 'MONDO', 'PYTHON']
```

3. filter() – tiene solo gli elementi che soddisfano una condizione

Python

```
numeri = [1, 2, 3, 4, 5, 6]

pari = list(filter(lambda n: n % 2 == 0, numeri))
# n = ogni numero della lista
print(par) # [2, 4, 6]
```

```

parole = ["casa", "auto", "a", "sole", "in"]
lunghe = list(filter(lambda p: len(p) > 3, parole))
# p = ogni parola
print(lunghe) # ['casa', 'auto', 'sole']

```

4. max() e min() con key

Python

```

persone = [{"nome": "Luca", "punteggio": 85}, {"nome": "Sara", "punteggio": 90}

migliore = max(persone, key=lambda persona: persona["punteggio"])
# persona = ogni dizionario
print(migliore["nome"]) # Sara

```



5. list.sort() (metodo diretto della lista)

Python

```

frutta = ["mela", "banana", "kiwi", "ananas"]

frutta.sort(key=lambda f: len(f)) # ordina per lunghezza
# f = ogni stringa della lista
print(frutta) # ['kiwi', 'mela', 'banana', 'ananas']

```

6. Pandas .apply() – molto simile a quello che hai visto con il sentiment

Python

```

import pandas as pd

df = pd.DataFrame({
    "nome": ["Anna", "Bob", "Carlo"],
    "età": [25, 30, 22]
})

# Applica a ogni riga (axis=1)
df["maggiorenne"] = df.apply(lambda riga: "sì" if riga["età"] >= 18

```

```
# riga = ogni riga del DataFrame (una Series)

# Applica a ogni elemento di una colonna
df["nome_maiuscolo"] = df["nome"].apply(lambda nome: nome.upper())
# nome = ogni valore della colonna "nome"
```

7. Flask / FastAPI – routing di richieste web (simile a LangGraph!)

Python

```
from flask import Flask
app = Flask(__name__)

@app.route("/saluta/<nome>")
def saluta(nome):           # <-- Flask passa automaticamente il va
    return f"Ciao {nome}!"

# Quando visiti /saluta/Mario → Flask chiama saluta("Mario")
# Tu sai che "nome" è una stringa perché è definito nella route
```

8. Event handler in GUI (es. tkinter)

Python

```
import tkinter as tk

def cliccato(event):          # event viene passato automaticamente da
    print("Hai cliccato alle coordinate:", event.x, event.y)

finestra = tk.Tk()
bottone = tk.Button(finestra, text="Cliccami")
bottone.bind("<Button-1>", cliccato)  # tkinter passa l'evento
bottone.pack()
finestra.mainloop()
```

Riassunto finale: la regola d'oro

In **tutti** questi casi:

Funzione/metodo	Cosa passa automaticamente alla lambda/funzione?	Come lo sai?
sorted(..., key=)	Un elemento della lista	Documentazione + tu conosci la lista
map(lambda x: ..., lista)	Un elemento della lista	Documentazione
filter(lambda x: ..., lista)	Un elemento della lista	Documentazione
df.apply(lambda riga: ..., axis=1)	Una riga del DataFrame	Documentazione pandas
LangGraph routing lambda	Lo state del grafo	Documentazione LangGraph + tu definisci lo State

Il segreto è sempre lo stesso:

Non sei tu a chiamare la lambda e a passare il valore.

È la funzione/framework che la chiama per te, passando il valore che ha promesso nella documentazione.

Una volta capita questa idea, tutto diventa prevedibile e sicuro!

Se vuoi, possiamo fare un piccolo esercizio insieme: ti do una funzione "madre" inventata e tu provi a scrivere la lambda corretta. Così fissiamo il concetto per sempre 😊