Dulana - Agentic AI - What is CrewAI

crew ai

Here is a **clear, beginner-friendly explanation of *Crew AI***—the framework you've been hearing about for building **agentic AI systems**.

🚀 What is CrewAI?

CrewAI is a Python framework for building systems where **multiple AI agents work together**, each with a specific role, collaborating like a *team* ("a crew") to complete complex tasks.

Think of it like:

> 🧠 **Agents = AI teammates**
> **Crew = Team of agents**
> **Tasks = What the team has to accomplish together**

CrewAI handles the **coordination**, **delegation**, and **workflow** between agents.

🧩 Why do we need CrewAI?

When a single LLM struggles with a complex task (e.g., research → analyze → write → review → summarize), you can split the work into multiple agents:

- 🔎 **Research Agent**
- 🧮 **Analysis Agent**
- ✍️ **Writer Agent**
- 🕵️ **Reviewer Agent**

CrewAI allows these agents to communicate, pass results, refine outputs, and follow a workflow.

This is part of the modern **Agentic AI** trend.

🛠️ How CrewAI Works (Simple)

1. **Define Agents**
   Each agent has:
   - a role
   - a goal
   - a backstory
   - a toolset

2. **Define Tasks**
   Each task describes:
   - what the agent must do
   - expected output format
3. **Define a Process**
   - sequential
   - hierarchical
   - shared memory
4. **Run the Crew**
   CrewAI orchestrates how agents talk and pass information.

🧪 Tiny Example

```python
from crewai import Agent, Task, Crew
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4o")

researcher = Agent(
    role="Researcher",
    goal="Find accurate information",
    backstory="Expert in internet research",
    llm=llm
)

writer = Agent(
    role="Writer",
    goal="Create a concise summary",
    backstory="Technical writer",
    llm=llm
)

task1 = Task(
    description="Research the latest trends in Agentic AI.",
    agent=researcher
)

task2 = Task(
    description="Summarize the research into a clean paragraph.",
    agent=writer
)

crew = Crew(
    agents=[researcher, writer],
    tasks=[task1, task2]
)

result = crew.run()
print(result)
```

🎯 Key Features of CrewAI

Feature	Explanation
Multi-agent collaboration	Agents talk to each other and refine outputs
Agent memory	Short-term and long-term memory between tasks
Tool integration	Agents can use tools (APIs, search, code exec)
Structured workflows	Sequential, parallel, or hierarchical task flow
Supports all LLMs	OpenAI, Anthropic, Gemini, local models, Ollama

---

🧠 Where CrewAI Fits in Agentic AI Stack

CrewAI sits at the **Orchestration Layer**:

```scss
LLM (GPT, Llama, Claude)
        │
Agents (Roles, Tools)
        │
CrewAI (Workflow)
        │
Application (RAG, automation, pipeline)
```

It is similar to LangGraph but simpler and designed for **team-of-agents** workflows.

📌 When Should You Use CrewAI?

Use CrewAI if your task requires:

- multi-step reasoning
- role specialization
- collaboration between agents
- complex workflows (research → analysis → coding → testing → summary)
- tool usage (browser, python, API calls)

If you want a **simple multi-agent system** → use CrewAI.
If you want **fine-grained control, loops, graphs** → LangGraph.

🎁 If you want, I can prepare:

✅ A **CrewAI slide** for your Agentic AI presentation
✅ A **step-by-step tutorial**
✅ Comparison between CrewAI and LangGraph
✅ Full project template with custom agents and tools
Just tell me!

(agent) PS D:\New folder\Agentic-AI> cd "Agentic-AI\Langgraph-Bootcamp\01_Introduction"
cd : Cannot find path 'D:\New folder\Agentic-AI\Agentic-AI\Langgraph-Bootcamp\01_Introduction' because it does not exist.
At line:1 char:1
+ cd "Agentic-AI\Langgraph-Bootcamp\01_Introduction"
+ ~~~~~~
    + CategoryInfo   : ObjectNotFound: (D:\New folder\A...01_Introduction:String) [Set-Location], ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.SetLocationCommand

(agent) PS D:\New folder\Agentic-AI> why is this

The error happens because your command created a **duplicate folder name** in the path:

PowerShell thinks you're trying to go to:

```vbnet
D:\New folder\Agentic-AI\Agentic-AI\Langgraph-Bootcamp\01_Introduction
```

...but the actual folder structure is probably:

```vbnet
D:\New folder\Agentic-AI\Langgraph-Bootcamp\01_Introduction
```

So when you run:

```ps
cd "Agentic-AI\Langgraph-Bootcamp\01_Introduction"
```

and you're **already inside**:

```ps
D:\New folder\Agentic-AI>
```

PowerShell appends the path and becomes:

```vbnet
D:\New folder\Agentic-AI\Agentic-AI\Langgraph-Bootcamp\01_Introduction
```

➡️ **That folder does not exist → error.**

✅ How to fix

Instead of:

```ps
cd "Agentic-AI\Langgraph-Bootcamp\01_Introduction"
```

Use:

```ps
cd ".\Langgraph-Bootcamp\01_Introduction"
```

or full path:

```ps
cd "D:\New folder\Agentic-AI\Langgraph-Bootcamp\01_Introduction"
```

🔍 Verify folder names

Run:

```ps
ls
```

This will show you the folders inside **`Agentic-AI`**. You should see:

```
Langgraph-Bootcamp
```

If you want, send me the result of **`ls`**, and I will tell you the exact cd command.

Microsoft Windows [Version 10.0.26200.7171]
(c) Microsoft Corporation. All rights reserved.

D:\New folder>conda active agent
'conda' is not recognized as an internal or external command,
operable program or batch file.

D:\New folder>cd "D:\New folder\Agentic-AI\Langgraph-Bootcamp\02_Reflection Agent"

D:\New folder\Agentic-AI\Langgraph-Bootcamp\02_Reflection Agent>cond info --envs
'cond' is not recognized as an internal or external command,
operable program or batch file.

D:\New folder\Agentic-AI\Langgraph-Bootcamp\02_Reflection Agent>conda info --envs
'conda' is not recognized as an internal or external command,
operable program or batch file.

```
D:\New folder\Agentic-AI\Langgraph-Bootcamp\02_Reflection Agent>conda info
'conda' is not recognized as an internal or external command,
operable program or batch file.

D:\New folder\Agentic-AI\Langgraph-Bootcamp\02_Reflection Agent>
```

Your issue is **very simple**:

➡️ **PowerShell cannot find Conda because your Conda is not added to PATH.**
That's why every time you run `conda`, it says:

```csharp
'conda' is not recognized…
```

This happens mostly when:

- Miniconda/Anaconda was installed **for "Just Me"**
- You skipped the step **Add conda to PATH** (default: OFF)
- You installed Anaconda but are using **PowerShell**, not Anaconda Prompt

Let's fix it.

✅ Fix 1 — Use the built-in Anaconda Prompt

This is the simplest.

1. Click **Start Menu**
2. Search **Anaconda Prompt**
3. Open it
4. THEN run:

```cmd
conda activate agent
```

If it works → your installation is fine, PowerShell just doesn't know about Conda.

✅ Fix 2 — Initialize Conda in PowerShell

Open PowerShell and run:

```ps
"C:\ProgramData\anaconda3\Scripts\conda.exe" init powershell
```

OR if you installed Miniconda:

```ps
"C:\Users\<YOUR_USERNAME>\miniconda3\Scripts\conda.exe" init powershell
```

Then **close PowerShell and reopen**.

Now try:

```ps
conda info
```

If you don’t know where conda is installed, do this:

```ps
where conda
```

If it shows nothing → Conda is NOT in your PATH.

✅ Fix 3 — Add Conda to PATH manually

Check the default installation paths.

If you installed Anaconda:

```makefile
C:\ProgramData\anaconda3
C:\ProgramData\anaconda3\Scripts
C:\ProgramData\anaconda3\Library\bin
```

If you installed Miniconda:

```makefile
C:\Users\<YourName>\miniconda3
C:\Users\<YourName>\miniconda3\Scripts
C:\Users\<YourName>\miniconda3\Library\bin
```

Add these to your **System PATH**:

1. Press **Win + R**
2. Type **sysdm.cpl**
3. Go to **Advanced** → **Environment Variables**
4. Edit PATH → Add the folders above
5. Restart PowerShell

Now try:

```ps
conda --version
```

🔎 If nothing works, check if Conda is installed

Run:

```ps
dir "C:\ProgramData\"
```

and

```ps
dir "C:\Users\<yourname>\"
```

Look for:

- **`anaconda3`**
- **`miniconda3`**

If not found → Conda is not installed on this machine.

👉 Send me the output of this:

```ps
where conda
```

and

```ps
dir "C:\ProgramData\"
```

I'll tell you exactly what to do next.

i want to do it in cmd

Perfect — doing it in **CMD** is the easiest because Conda integrates cleanly with it.

Right now **`conda`** is not recognized, which means **Conda is not initialized for CMD OR not in PATH**.

Let's fix it step by step.

✅ STEP 1 — Find where Conda is installed

Run this in CMD:

```cmd
dir C:\ /s /b | findstr conda
```

OR check the common locations:

Try:

```cmd
dir "C:\ProgramData"
dir "C:\Users\%USERNAME%"
```

Look for a folder named:

- **`anaconda3`**
- **`miniconda3`**

Tell me which one you have and the exact path.

✅ STEP 2 — Initialize Conda for CMD

Once you know the installation folder, run this:

If you have Anaconda:

```cmd
"C:\ProgramData\anaconda3\Scripts\conda.exe" init cmd.exe
```

If you have Miniconda:

```cmd
"C:\Users\%USERNAME%\miniconda3\Scripts\conda.exe" init cmd.exe
```

You should see:

```bash
No writing was done
```

or

```php-template
Modified C:\Users\<you>\.condarc
```

Then **close CMD and open a new CMD window.**

Now test:

```cmd
conda info
conda env list
```

❗ If the command above gives ERROR ("file not found")

That means Conda is **not installed**, or installed in a different location.

In that case, you must run:

```cmd
where conda
```

If this returns nothing → Conda is not in PATH.

🎯 STEP 3 — Activate your environment

Once Conda is working:

```cmd
conda activate agent
```

Then run your LangGraph commands.

📌 Quick Summary (CMD version)

Goal	Command
Find Conda	**`where conda`**
Initialize for CMD	**`conda init cmd.exe`**
Activate env	**`conda activate agent`**

👉 Please send me the output of this from CMD:

```
cmd

where conda
```

With that, I can tell you the exact fix.

from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
import datetime
from langchain_openai import ChatOpenAI
from schema import AnswerQuestion, ReviseAnswer
from langchain_core.output_parsers.openai_tools import PydanticToolsParser,
JsonOutputToolsParser
from langchain_core.messages import HumanMessage
what each one do show me with exapmles

Nice, you're in the fun parts now 😄
Let's go through **each import one by one** with **simple explanations + mini examples**.

1️⃣ `ChatPromptTemplate` – build prompts for chat models

```python
from langchain.prompts import ChatPromptTemplate
```

What it is:
A helper to create **structured prompts** for chat models (with variables like `{question}`), instead of manually concatenating strings.

Example:

```python
from langchain.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant. Today is {today}."),
    ("human", "Answer this question clearly: {question}")
])

# Later when calling it:
filled = prompt.format(
    today="2025-12-09",
    question="What is Agentic AI?"
)

print(filled)
```

`filled` will be a list of formatted messages (system + human) with placeholders replaced.

In a full chain:

```python
from langchain_openai import ChatOpenAI

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

chain = prompt | llm   # LCEL pipe operator

response = chain.invoke({
    "today": "2025-12-09",
    "question": "What is Agentic AI?"
})

print(response.content)
```

2️⃣ `MessagesPlaceholder` – slot for message history

```python
from langchain.prompts import MessagesPlaceholder
```

What it is:
A special placeholder inside a **`ChatPromptTemplate`** where you can **inject a list of messages** (e.g., chat history).

Example:

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage, AIMessage

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful tutor."),
    MessagesPlaceholder("history"),   # 👈 here goes previous messages
    ("human", "Now answer this: {question}")
])

history = [
    HumanMessage(content="Hi, can you help me with LangChain?"),
    AIMessage(content="Sure! What would you like to know?")
]

filled = prompt.invoke({
    "history": history,
    "question": "What is a prompt template?"
})

print(filled)
```

This lets your model see the **past conversation**, not just the latest question.

3️⃣ `datetime` – Python standard library for time/date

```python
import datetime
```

What it is:
Built-in Python module for working with dates and times. Often used to pass the **current time** into the prompt.

Example:

```python
import datetime

now = datetime.datetime.now()
today_str = now.strftime("%Y-%m-%d %H:%M")

print(today_str)  # e.g., '2025-12-09 15:32'
```

Used with LangChain:

```python
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant. Current time: {now}."),
    ("human", "Summarize today's news about AI.")
])

inputs = {"now": today_str}
```

4️⃣ `ChatOpenAI` – wrapper around OpenAI chat models

```python
from langchain_openai import ChatOpenAI
```

What it is:
A LangChain **LLM class** that talks to OpenAI's chat models (like `gpt-4o`), and works nicely with prompts/runnables.

Example:

```python
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(
    model="gpt-4o-mini",
    temperature=0.2
)

response = llm.invoke("Explain RAG in one paragraph.")
print(response.content)
```

Used in a chain:

```python
chain = prompt | llm
result = chain.invoke({"question": "What is CrewAI?"})
```

5️⃣ `AnswerQuestion` & `ReviseAnswer` – your Pydantic schemas

```python
from schema import AnswerQuestion, ReviseAnswer
```

These are **not built-in LangChain**, they come from **your own `schema.py` file** in that project.

In the LangGraph bootcamp, they're usually **Pydantic models** used for **tool outputs** or **structured LLM results**.

Likely something like:

```python
# schema.py (example)
from pydantic import BaseModel

class AnswerQuestion(BaseModel):
    answer: str
    sources: list[str]

class ReviseAnswer(BaseModel):
    improved_answer: str
    critique: str
```

Then you can tell the LLM:

> "Return your answer as an `AnswerQuestion` object."

And with output parsers (next section) you get a real typed Python object, not just raw text.

6️⃣ `PydanticToolsParser` – parse tools output into Pydantic models

```python
from langchain_core.output_parsers.openai_tools import PydanticToolsParser
```

What it is:
An output parser that:

1. Takes **OpenAI tool-calls style JSON** output
2. Parses it into **Pydantic models** (like your `AnswerQuestion`, `ReviseAnswer`)

Used when you do **tool-style / structured outputs**.

Example:

```python
from langchain_core.output_parsers.openai_tools import PydanticToolsParser
from langchain_openai import ChatOpenAI
from schema import AnswerQuestion

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

```
    temperature=0,
    # Important: tell model about the tool/schema
    tools=[AnswerQuestion]   # depends on exact setup in your repo
)

parser = PydanticToolsParser(tools=[AnswerQuestion])

messages = [
    {"role": "user", "content": "What is LangGraph? Explain briefly."}
]

raw_response = llm.invoke(messages)
parsed = parser.invoke(raw_response)

# parsed will be something like: [AnswerQuestion(answer="...", sources=[...])]
print(parsed[0].answer)
```

So you move from **LLM JSON** → **nice Python objects**.

7️⃣ `JsonOutputToolsParser` – same idea but returns plain JSON

```python
from langchain_core.output_parsers.openai_tools import JsonOutputToolsParser
```

What it is:
Similar to **`PydanticToolsParser`**, but instead of Pydantic objects, it gives you **plain Python dicts / lists** (JSON-style).

Example:

```python
from langchain_core.output_parsers.openai_tools import JsonOutputToolsParser

parser = JsonOutputToolsParser()

raw_response = llm.invoke(messages)
parsed = parser.invoke(raw_response)

print(parsed)
# e.g. [{'type': 'AnswerQuestion', 'args': {'answer': '...', 'sources': ['...']}}]
```

Use this if you don't want Pydantic models and just need raw JSON.

8️⃣ `HumanMessage` – represent a human chat message

```python
from langchain_core.messages import HumanMessage
```

What it is:
LangChain's internal **message type** for content from the **user** (you).

Instead of just passing strings around, LangChain uses classes:

- **`HumanMessage`** – from the user

- **`AIMessage`** – from the model
- **`SystemMessage`** – system instructions
- etc.

Example:

```python
from langchain_core.messages import HumanMessage
from langchain_openai import ChatOpenAI

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

messages = [
    HumanMessage(content="Explain what an agent is in Agentic AI.")
]

response = llm.invoke(messages)
print(response.content)
```

Used with **`MessagesPlaceholder`** for chat history:

```python
history = [
    HumanMessage(content="Hi"),
    HumanMessage(content="Explain LangGraph briefly.")
]
```

Putting it all together (mini reflection-style flow)

Here's a tiny sketch similar to your **Reflection Agent** notebook:

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage
from langchain_core.output_parsers.openai_tools import PydanticToolsParser
from schema import AnswerQuestion

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert AI tutor."),
    MessagesPlaceholder("history"),
    ("human", "{question}")
])

parser = PydanticToolsParser(tools=[AnswerQuestion])

history = [HumanMessage(content="Hi, help me learn Agentic AI.")]

chain = prompt | llm | parser

result = chain.invoke({
    "history": history,
    "question": "What is a reflection agent in LangGraph?"
})
```

```
answer_obj = result[0]   # AnswerQuestion instance
print(answer_obj.answer)
```

If you want, next we can:

- open your actual **`schema.py`**
- walk through **AnswerQuestion** and **ReviseAnswer** line by line
- and map them to how the reflection node is using them.

how can i do this in gemini ai

Nice, so the idea is:

> 🧾 You were using **OpenAI + tools + PydanticToolsParser**
> 🟣 Now you want to do the **same thing with Gemini**.

With Gemini, the easiest way is to **skip `PydanticToolsParser`** and instead use:

```python
llm.with_structured_output(YourPydanticModel)
```

LangChain will make Gemini return your Pydantic model directly.

1️⃣ Basic Gemini setup in LangChain

```python
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash-exp",  # or "gemini-1.5-pro" if available in your env
    temperature=0.2
)
```

Make sure you have:

```bash
export GOOGLE_API_KEY="your-key-here"
```

(or set in **`.env`** / system env vars).

2️⃣ Using your `AnswerQuestion` Pydantic schema with Gemini

Assume your `schema.py` looks like:

```python
# schema.py
from pydantic import BaseModel

class AnswerQuestion(BaseModel):
    answer: str
    sources: list[str]

class ReviseAnswer(BaseModel):
    improved_answer: str
    critique: str
```

✅ Gemini: structured output with Pydantic

```python
from langchain_google_genai import ChatGoogleGenerativeAI
from schema import AnswerQuestion, ReviseAnswer

base_llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash-exp",
    temperature=0
)

# One LLM that always returns AnswerQuestion
answer_llm = base_llm.with_structured_output(AnswerQuestion)

# Another that always returns ReviseAnswer
revise_llm = base_llm.with_structured_output(ReviseAnswer)
```

Now:

```python
result: AnswerQuestion = answer_llm.invoke(
    "Explain what LangGraph is in 3–4 sentences and list any sources you assume."
)

print(type(result))          # <class 'schema.AnswerQuestion'>
print(result.answer)         # the explanation text
print(result.sources)        # a list of strings
```

And for revision:

```python
rev: ReviseAnswer = revise_llm.invoke(
    {
        "improved_answer": "LangGraph is ...",  # Gemini will see this as plain prompt text
    }
)
print(rev.improved_answer)
print(rev.critique)
```

You can also pass a single string prompt; the model just needs enough instruction in the prompt to know what to fill.

3️⃣ Using it with `ChatPromptTemplate` + `MessagesPlaceholder`

Here's how you plug it into your reflection-style setup:

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage
from langchain_google_genai import ChatGoogleGenerativeAI
from schema import AnswerQuestion, ReviseAnswer

base_llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash-exp",
    temperature=0
)

answer_llm = base_llm.with_structured_output(AnswerQuestion)
revise_llm = base_llm.with_structured_output(ReviseAnswer)

# 1) Prompt for answering
answer_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert AI tutor."),
    MessagesPlaceholder("history"),
    ("human", "Answer the student's question: {question}")
])

# 2) Prompt for reflection / revision
revise_prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a strict reviewer. Improve the answer and critique it."),
    ("human", "Question: {question}\nDraft answer: {draft}")
])

# Chains
answer_chain = answer_prompt | answer_llm
revise_chain = revise_prompt | revise_llm

history = [HumanMessage(content="Hi, help me understand Agentic AI.")]

# First: get an answer
ans: AnswerQuestion = answer_chain.invoke({
    "history": history,
    "question": "What is a reflection agent in LangGraph?"
})

print("Draft:", ans.answer)

# Then: send to reviewer
rev: ReviseAnswer = revise_chain.invoke({
    "question": "What is a reflection agent in LangGraph?",
    "draft": ans.answer
})

print("Improved:", rev.improved_answer)
print("Critique:", rev.critique)
```

No **`PydanticToolsParser`**, no **`JsonOutputToolsParser`** – Gemini + LangChain handles the structured output directly.

4️⃣ So what changes vs your OpenAI setup?

With OpenAI (tools style)

You did something like:

```python
llm = ChatOpenAI(..., tools=[AnswerQuestion])
parser = PydanticToolsParser(tools=[AnswerQuestion])
chain = prompt | llm | parser
```

With Gemini

Do this instead:

```python
llm = ChatGoogleGenerativeAI(...)
answer_llm = llm.with_structured_output(AnswerQuestion)
chain = prompt | answer_llm
```

So:

- ❌ You don't need `PydanticToolsParser`
- ✅ You use `.with_structured_output(PydanticModel)`
- ✅ You get the Pydantic model instance directly from `.invoke()`

If you paste your **exact** `schema.py` here, I can write the **exact Gemini version** of your Reflection Agent code (generation_chain + reflection_chain) using those models.

from pydantic import BaseModel,Field
from typing import List
what does do these show me with expales

Let's break them down **simply + clearly**, with **examples** you can run.

✅ 1. `from pydantic import BaseModel, Field`

🧠 What is Pydantic?

Pydantic is a Python library that allows you to define **data models** with:

- validation
- type safety
- automatic conversion
- easy JSON parsing
- perfect for **LLM structured output**

🔹 `BaseModel`

`BaseModel` is the **base class** for all Pydantic models.

You create your own models by **inheriting** from it.

Example:

```python
from pydantic import BaseModel

class User(BaseModel):
    name: str
    age: int
```

Usage:

```python
u = User(name="Alice", age=25)
print(u.name)    # Alice
print(u.age)     # 25
```

🔥 Pydantic VALIDATES your data automatically

Try this:

```python
User(name="Alice", age="25")
```

Output:

```ini
age=25  # Pydantic automatically converts string to int
```

But this fails:

```python
User(name="Alice", age="twenty")
```

Error:

```csharp
ValueError: invalid literal for int()
```

So Pydantic helps ensure your LLM outputs match the expected schema.

🔹 `Field`

Field lets you add **extra metadata**, like:

- descriptions
- default values
- constraints (min, max, regex, etc.)

Example:

```python
from pydantic import BaseModel, Field

class Product(BaseModel):
    name: str = Field(..., description="Name of the product")
    price: float = Field(..., gt=0, description="Price must be positive")
```

Usage:

```python
p = Product(name="Laptop", price=1200)
print(p)
```

If you try:

```python
Product(name="Laptop", price=-10)
```

Error:

```nginx
price must be greater than 0
```

🔥 Example combining BaseModel + Field

Perfect for structured LLM output:

```python
from pydantic import BaseModel, Field

class AnswerQuestion(BaseModel):
    answer: str = Field(..., description="Main answer text")
    sources: list[str] = Field(default_factory=list, description="List of sources")
```

Usage:

```python
obj = AnswerQuestion(
    answer="LangGraph lets you create agent workflows.",
    sources=["https://langchain.com"]
)
```

```
print(obj.answer)
print(obj.sources)
```

✅ 2. `from typing import List`

`List` is used for **type hinting** – telling Python and Pydantic that a field holds a list of items.

Example:

```python
from typing import List
from pydantic import BaseModel

class Student(BaseModel):
    name: str
    marks: List[int]    # list of integers
```

Usage:

```python
s = Student(name="Dulama", marks=[85, 90, 77])
print(s.marks)
```

If you try:

```python
Student(name="Dulama", marks=[85, "abc"])
```

Error:

```pgsql
value is not a valid integer
```

Pydantic validates every element of the list.

🎯 When combined:

This is what you use in your LangGraph project:

```python
from pydantic import BaseModel, Field
from typing import List

class ReviseAnswer(BaseModel):
    improved_answer: str = Field(..., description="Improved answer text")
    critique: str = Field(..., description="Short critique")
    suggestions: List[str] = Field(default_factory=list, description="List of improvements")
```

This enables:

✔️ Structured output
✔️ Auto validation
✔️ Perfect compatibility with Gemini or OpenAI tool calling
✔️ Clean JSON-like objects in Python

📌 Quick Summary

Item	Purpose	Example
`BaseModel`	Create data classes	`class User(BaseModel): ...`
`Field`	Add descriptions, constraints	`Field(..., gt=0)`
`List`	Type hint for a list	`marks: List[int]`

If you want, I can now show:

✅ How to make Gemini RETURN this Pydantic model
or
✅ How to use this in a LangGraph node (reflection or generation)

↳
improved_answer: str = Field(..., description="Improved answer text")
why there is 3 dots

Great question — this confuses almost everyone at first.

The **three dots** `...` in Pydantic **do not mean "etc."**
They mean something very specific:

✅ `...` means "this field is REQUIRED"

In Pydantic, `...` is a special marker called **Ellipsis**, used to indicate that a value **must** be provided.

Example:

```python
from pydantic import BaseModel, Field

class User(BaseModel):
    name: str = Field(..., description="User's full name")
```

Here:

- `name` **must be provided**
- if missing → validation error

🧪 Example: Required vs Not Required

✔️ Required field (uses `...`)

```python
from pydantic import BaseModel, Field

class Product(BaseModel):
    name: str = Field(..., description="Product name")    # REQUIRED
    price: float = Field(..., description="Price")        # REQUIRED
```

Try:

```python
Product(price=100)
```

You get:

```csharp
ValidationError: field 'name' is required
```

❌ Optional or has a default (NO `...`)

```python
from pydantic import BaseModel, Field

class Product(BaseModel):
    name: str = Field(default="Unknown", description="Product name")
    price: float = 0.0
```

Now this works:

```python
Product()
```

Output:

```ini
name='Unknown' price=0.0
```

🎯 Why does LangChain + Pydantic use `...`?

Because when you want *structured output from LLMs*, every field should be **required** so the LLM always fills it.

Example from your code:

```python
class ReviseAnswer(BaseModel):
    improved_answer: str = Field(..., description="Improved answer text")
    critique: str = Field(..., description="Short critique")
```

Here:

- `improved_answer` must be returned
- `critique` must be returned

Otherwise, the parser will fail.

🧠 Quick Summary

Symbol	Meaning
`...`	field is **required** (no default)
None	field can be null / optional
Value (like `"abc"` or `0`)	field has a default

✔️ Examples for clarity

REQUIRED field:

```python
answer: str = Field(...)
```

Has default value:

```python
answer: str = Field("unknown")
```

Optional field:

```python
from typing import Optional

answer: Optional[str] = None
```

If you'd like, I can rewrite your `schema.py` with correct required/optional fields and explain each one.

```
from pydantic import BaseModel,Field
from typing import List

class Reflection(BaseModel):
    missing: str = Field(description="Critique of what is missing.")
    superfluous: str = Field(description="Critique of what is superfluous.")

class AnswerQuection(BaseModel):
    """Answer the quection."""

    answer:str = Field(
        description="~250 woed detailed answer to the quection")
    search_queries: List[str] = Field(
        deascription = "1-3 search queries for researching improvements to address the critique of your current answer."
    )
    reflection: Reflection = Field(
        description="Your reflection on the initial answer."
    )

class ReviseAnswer(AnswerQuection):
    """Revise your original answer to your quection."""

    references: List[str] = Field(
        description="Citations motivating your updated answer."
    )


    explain this code
```

Here is a **clean, simple, and complete explanation** of your code — **what each line does, why it's written this way, and how the models are meant to be used** (especially in LangGraph / Gemini structured output).

✅ First, the full code (corrected typos)

Your code has spelling mistakes (`deascription`, `quection`, `woed`).
I'll explain the intention *and* show the corrected version.

But first—let's explain the original code.

🔍 1. Imports

```python
from pydantic import BaseModel, Field
```

```
from typing import List
```

✔️ `BaseModel`

This is the parent class for all Pydantic models.
It gives:

- type validation
- automatic JSON parsing
- helpful `.dict()` and `.json()` methods
- compatibility with LangChain structured output

✔️ `Field`

Adds metadata such as:

- description
- default values
- validators
- required fields

✔️ `List`

Means the field is a list of some type (`List[str]`, `List[int]`, etc.).

🔍 2. Reflection model

```
python

class Reflection(BaseModel):
    missing: str = Field(description="Critique of what is missing.")
    superfluous: str = Field(description="Critique of what is superfluous.")
```

✔️ What is this?

This Pydantic model holds a **reflection/critique** about an answer.

Fields:

Field	Type	Meaning
`missing`	`str`	What the answer failed to include
`superfluous`	`str`	What the answer included unnecessarily

What this object looks like:

Example object:

```json
{
  "missing": "The answer does not explain LangGraph state updates.",
  "superfluous": "Too much detail about the history of transformers."
}
```

This is usually used in a **reflection agent** step.

🔍 3. AnswerQuestion model

```python
class AnswerQuection(BaseModel):
    """Answer the quection."""

    answer: str = Field(
        description="~250 woed detailed answer to the quection"
    )

    search_queries: List[str] = Field(
        deascription="1-3 search queries for researching improvements to address the critique of
your current answer."
    )

    reflection: Reflection = Field(
        description="Your reflection on the initial answer."
    )
```

✔️ What it represents

This is the **structured output** of your **Generation Agent**.

It includes:

Field	Type	Meaning
`answer`	`str`	A ~250-word answer generated by the LLM
`search_queries`	`List[str]`	Search phrases to help improve the answer
`reflection`	`Reflection`	The critique (object defined above)

✔️ Example of a valid `AnswerQuestion` object

```json
{
  "answer": "LangGraph is a framework for building agent workflows...",
  "search_queries": [
    "LangGraph state management",
    "reflection agents LangChain",
    "agent workflows python examples"
  ],
  "reflection": {
    "missing": "No mention of how nodes are defined.",
    "superfluous": "Too much detail about LCEL origins."
```

```
    }
}
```

This is EXACTLY the structured JSON that Gemini or OpenAI tools mode would return.

🔍 4. ReviseAnswer model

```
python

class ReviseAnswer(AnswerQuection):
    """Revise your original answer to your quection."""

    references: List[str] = Field(
        description="Citations motivating your updated answer."
    )
```

✔️ What is happening?

`ReviseAnswer` inherits from **`AnswerQuestion`**.
This means:

➡️ **It contains all fields from AnswerQuestion**, plus one more:

Field	Type	Meaning
`references`	List[str]	URLs, citations, sources used to revise the answer

Why inheritance?

Because revising the answer requires:

- the new improved **`answer`**
- the updated **`search_queries`**
- the new **`reflection`**
- AND *additional references*

So the structure is:

```
sql

ReviseAnswer
├── answer
├── search_queries
├── reflection
└── references
```

✔️ Example of a valid `ReviseAnswer` output

```json
{
  "answer": "Updated, clearer explanation of LangGraph...",
  "search_queries": [
    "LangGraph reactive workflows",
    "LangGraph memory patterns"
  ],
  "reflection": {
    "missing": "Still needs examples of multi-node graphs.",
    "superfluous": "Removed unnecessary LCEL explanation."
  },
  "references": [
    "https://docs.langchain.com/docs/langgraph",
    "https://python.langchain.com/docs"
  ]
}
```

This would be the final output after the reflection + revision step.

🔥 Putting Everything Together (How Used in Agents)

Step 1 → Generation Model returns `AnswerQuestion`

Step 2 → Reflection Model critiques the answer (`reflection` object)

Step 3 → Revision Model returns `ReviseAnswer`

(Updated answer + sources)

This is exactly the architecture of:

- LangGraph "Reflection Agent"
- OpenAI "Critic → Reviser" chain
- Gemini structured output sequence

🧠 Why it is designed this way?

Because:

- LLMs are bad at long reasoning in one pass
- Reflection forces the model to improve
- Structured output ensures consistent quality
- Pydantic guarantees type safety
- LangGraph allows cycling between generation → critique → revision

This is **one of the most powerful agent patterns** you'll learn.

✔️ If you want, I can:

✅ Correct all spelling errors
✅ Convert this to **Gemini-friendly structured output**
✅ Write the full LangGraph nodes and edges
✅ Show a working chain using `.with_structured_output()`

Just tell me!

multiple line comments in python

In Python, there is *no official* multi-line comment syntax like `/* ... */` in C, **but** there are **3 correct ways** to write multi-line comments depending on what you want to do.

✅ 1. Multi-line comment using triple quotes (`""" ... """`)

This is the most common method.

```python
"""
This is a multi-line comment in Python.
It can span many lines.
Python ignores it unless it is the first line of a function/class.
"""
```

Python treats triple-quoted strings as **string literals**, and if they are not assigned or used, they behave like comments.

✅ 2. Multi-line comment using multiple `#`

Officially, Python comments use `#`.

```python
# This is a comment
# This is another comment
# This is a third line
```

This is the most explicit and recommended by PEP8.

✅ 3. Disable blocks of code temporarily

Developers often “comment out” code by:

```python
"""
x = 5
y = 10
```

```
print(x + y)
"""
```

This works because triple-quoted strings are ignored.

⚠️ IMPORTANT: When triple quotes DO NOT act like comments

If a triple-quoted string appears at the **top of a function, class, or module**, Python treats it as a **docstring**, not a comment.

Example:

```python
def add(a, b):
    """This describes the function. This is a docstring, not a comment."""
    return a + b
```

This string becomes part of the function's documentation.

🧠 Summary

Method	Example	Recommended?
Triple quotes (`""" ... """`)	multi-line string literal	Yes (for quick multi-line comments)
Multiple **#**	**`# line 1`**	Yes (official comment syntax)
Triple quotes to comment out code	**`""" print('hi') """`**	Yes, for temporary disabling

---

✔️ Quick Examples

Multi-line comment:

```python
"""
This is a multi-line comment
explaining my LangGraph reflection agent.
"""
```

Comment block:

```python
# Step 1: Create agent
# Step 2: Create prompt
# Step 3: Run graph
```

If you want, I can show you how to document your LangGraph nodes using docstrings too!

```
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
import datetime
from langchain_openai import ChatOpenAI
from schema import AnswerQuestion, ReviseAnswer
from langchain_core.output_parsers.openai_tools import PydanticToolsParser,
JsonOutputToolsParser
from langchain_core.messages import HumanMessage

pydantic_parser = PydanticToolsParser(tools=[AnswerQuestion])

parser = JsonOutputToolsParser(return_id=True)

# Actor Agent Prompt
actor_prompt_template = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """You are expert AI researcher.
Current time: {time}

1. {first_instruction}
2. Reflect and critique your answer. Be severe to maximize improvement.
3. After the reflection, **list 1-3 search queries separately** for researching improvements. Do not
include them inside the reflection.
""",
        ),
        MessagesPlaceholder(variable_name="messages"),
        ("system", "Answer the user's question above using the required format."),
    ]
).partial(
    time=lambda: datetime.datetime.now().isoformat(),
)

first_responder_prompt_template = actor_prompt_template.partial(
    first_instruction="Provide a detailed ~250 word answer"
)

llm = ChatOpenAI(model="gpt-4o")

first_responder_chain = first_responder_prompt_template | llm.bind_tools(tools=
[AnswerQuestion], tool_choice='AnswerQuestion')

validator = PydanticToolsParser(tools=[AnswerQuestion])

# Revisor section

revise_instructions = """Revise your previous answer using the new information.
    - You should use the previous critique to add important information to your answer.
```

- You MUST include numerical citations in your revised answer to ensure it can be verified.
- Add a "References" section to the bottom of your answer (which does not count towards the word limit). In form of:
  - [1] https://example.com
  - [2] https://example.com
- You should use the previous critique to remove superfluous information from your answer and make SURE it is not more than 250 words.

"""

revisor_chain = actor_prompt_template.partial(
first_instruction=revise_instructions
) | llm.bind_tools(tools=[ReviseAnswer], tool_choice="ReviseAnswer")

response = first_responder_chain.invoke({
"messages": [HumanMessage("AI Agents taking over content creation")]
})s

print(response)
convert this in to gemini

Nice, this is a perfect place to switch to Gemini + structured output 😎
I'll show you the **Gemini version of your code** and then highlight what changed.

✅ Gemini version of your code

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
import datetime
from langchain_google_genai import ChatGoogleGenerativeAI
from schema import AnswerQuestion, ReviseAnswer
from langchain_core.messages import HumanMessage

# -------------------------
# Shared prompt template
# -------------------------
actor_prompt_template = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """You are expert AI researcher.
Current time: {time}

1. {first_instruction}
2. Reflect and critique your answer. Be severe to maximize improvement.
3. After the reflection, **list 1-3 search queries separately** for researching improvements. Do not include them inside the reflection.
""",
        ),
        MessagesPlaceholder(variable_name="messages"),
        ("system", "Answer the user's question above using the required format."),
    ]
).partial(
    time=lambda: datetime.datetime.now().isoformat(),
)
```

```python
# First responder prompt (initial answer)
first_responder_prompt_template = actor_prompt_template.partial(
    first_instruction="Provide a detailed ~250 word answer"
)

# -------------------------
# LLM: Gemini base
# -------------------------
# Make sure GOOGLE_API_KEY is set in your env
# export GOOGLE_API_KEY="your-key-here"
llm_base = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash-exp",   # or "gemini-1.5-pro" if that's what you have
    temperature=0
)

# LLM that returns AnswerQuestion as a Pydantic object
first_responder_llm = llm_base.with_structured_output(AnswerQuestion)

# LLM that returns ReviseAnswer as a Pydantic object
revisor_llm = llm_base.with_structured_output(ReviseAnswer)

# -------------------------
# Chains
# -------------------------

# First responder chain: generate initial answer + reflection + search queries
first_responder_chain = first_responder_prompt_template | first_responder_llm

# Revisor instructions
revise_instructions = """Revise your previous answer using the new information.
    - You should use the previous critique to add important information to your answer.
        - You MUST include numerical citations in your revised answer to ensure it can be
verified.
        - Add a "References" section to the bottom of your answer (which does not count towards
the word limit). In form of:
            - [1] https://example.com
            - [2] https://example.com
    - You should use the previous critique to remove superfluous information from your answer
and make SURE it is not more than 250 words.
"""

# Revisor chain: returns ReviseAnswer object
revisor_chain = actor_prompt_template.partial(
    first_instruction=revise_instructions
) | revisor_llm

# -------------------------
# Example usage
# -------------------------
if __name__ == "__main__":
    # Step 1: first responder
    first_result: AnswerQuestion = first_responder_chain.invoke({
        "messages": [HumanMessage(content="AI Agents taking over content creation")]
    })

    print("=== First Answer ===")
    print(first_result.answer)
    print("\nReflection (missing):", first_result.reflection.missing)
    print("Reflection (superfluous):", first_result.reflection.superfluous)
    print("Search queries:", first_result.search_queries)

    # Step 2: revisor (you would usually add web search etc. in between)
    revised_result: ReviseAnswer = revisor_chain.invoke({
        "messages": [HumanMessage(
            content=f"Question: AI Agents taking over content creation\n"
                    f"Previous answer: {first_result.answer}\n"
                    f"Reflection: missing={first_result.reflection.missing}, "
                    f"superfluous={first_result.reflection.superfluous}"
        )]
    })

    print("\n=== Revised Answer ===")
```

```
    print(revised_result.answer)
    print("\nReferences:", revised_result.references)
```

🔄 What changed from your OpenAI version?

1. Swapped LLM class

Before:

```python
from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="gpt-4o")
```

Now:

```python
from langchain_google_genai import ChatGoogleGenerativeAI
llm_base = ChatGoogleGenerativeAI(model="gemini-2.0-flash-exp", temperature=0)
```

---

2. No `bind_tools`, no `PydanticToolsParser`, no `JsonOutputToolsParser`

With OpenAI tools you did:

```python
first_responder_chain = first_responder_prompt_template | llm.bind_tools(
    tools=[AnswerQuestion],
    tool_choice='AnswerQuestion'
)

validator = PydanticToolsParser(tools=[AnswerQuestion])
```

With Gemini + LangChain, you instead use:

```python
first_responder_llm = llm_base.with_structured_output(AnswerQuestion)
first_responder_chain = first_responder_prompt_template | first_responder_llm
```

- `.with_structured_output(AnswerQuestion)` tells Gemini:

  > "Return an object that matches this Pydantic model"

- You directly get a **Pydantic instance** back from `.invoke()`, no extra parser needed.

Same idea for the revisor:

```python
revisor_llm = llm_base.with_structured_output(ReviseAnswer)
revisor_chain = actor_prompt_template.partial(
```

```
    first_instruction=revise_instructions
) | revisor_llm
```

---

3. Structured output is now automatic

With OpenAI + tools:

- The LLM returns a **tool call + JSON**
- `PydanticToolsParser` converts that JSON → Pydantic model

With Gemini + `.with_structured_output`:

- The LLM is prompted behind the scenes to match your schema
- You directly get `AnswerQuestion` / `ReviseAnswer` Python objects

Example:

```python
first_result: AnswerQuestion = first_responder_chain.invoke({...})
print(first_result.answer)
print(first_result.search_queries)
print(first_result.reflection.missing)
```

If you paste your actual `schema.py` here, I can tune this snippet 100% to your exact field names and fix any typos (like `AnswerQuection` vs `AnswerQuestion`).

from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
import datetime
from langchain_google_genai import ChatGoogleGenerativeAI
from schema import AnswerQuestion, ReviseAnswer
from langchain_core.messages import HumanMessage
what do these each one with exapm=le

Sure — here is a **clear explanation of each import**, what it does, **why you need it**, and **examples showing exactly how it works**.

✅ 1. `ChatPromptTemplate`

```python
from langchain.prompts import ChatPromptTemplate
```

✔️ What it does

`ChatPromptTemplate` lets you **build structured prompts** for chat models.
Instead of writing one big long string, you create a prompt made of:

- system messages
- human messages
- variables
- templates

✔️ Example

```python
from langchain.prompts import ChatPromptTemplate

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful tutor."),
    ("human", "Explain {topic} in simple words.")
])

filled = prompt.format(topic="Agentic AI")
print(filled)
```

Output is a list of messages LangChain will send to the LLM:

```arduino
[
  SystemMessage("You are a helpful tutor."),
  HumanMessage("Explain Agentic AI in simple words.")
]
```

You then pass this into an LLM.

✅ 2. `MessagesPlaceholder`

```python
from langchain.prompts import MessagesPlaceholder
```

✔️ What it does

Allows you to insert **previous chat messages** or any list of **`HumanMessage`** / **`AIMessage`** into a prompt.

Useful for:

- maintaining conversation history
- multi-turn agents
- LangGraph where each step feeds previous messages

✔️ Example

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert AI assistant."),
    MessagesPlaceholder("history"),
    ("human", "{question}")
])

history = [
    HumanMessage(content="Hi, can you teach me about agents?"),
    HumanMessage(content="What is reflection?"),
]

res = prompt.format(
    history=history,
    question="Explain LangGraph in one paragraph."
)

print(res)
```

Now the LLM gets ALL previous messages in context.

✅ 3. `datetime`

```python
import datetime
```

✔️ What it does

Built-in Python module for working with **dates and time**.

Used in your agent prompt as:

```python
time=lambda: datetime.datetime.now().isoformat()
```

✔️ Example

```python
import datetime

now = datetime.datetime.now().isoformat()
print(now)
```

Output:

```makefile
2025-12-08T15:20:54.123456
```

Used in prompts when you want the agent to know the current time.

✅ 4. `ChatGoogleGenerativeAI`

```python
from langchain_google_genai import ChatGoogleGenerativeAI
```

✔️ What it does

This is the LangChain LLM wrapper for **Google Gemini models**.

It allows you to use models like:

- **`gemini-2.0-flash-exp`**
- **`gemini-1.5-pro`**
- **`gemini-1.0-pro`**

✔️ Example

```python
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash-exp",
    temperature=0.2
)

response = llm.invoke("Explain RAG in one paragraph.")
print(response.content)
```

This sends the message to Gemini and prints the answer.

✅ 5. `AnswerQuestion` & `ReviseAnswer` (your Pydantic models)

```python
from schema import AnswerQuestion, ReviseAnswer
```

✔️ What they do

These are **custom data models** you wrote to force Gemini to output structured JSON-like objects.

Example:

```python
class AnswerQuestion(BaseModel):
    answer: str
    search_queries: List[str]
    reflection: Reflection
```

✔️ Example use with Gemini structured output

```python
llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash-exp")

structured_llm = llm.with_structured_output(AnswerQuestion)

result = structured_llm.invoke("Explain LangGraph.")
print(result.answer)
print(result.search_queries)
```

Gemini now **returns a Python object**, not plain text.

✅ 6. `HumanMessage`

```python
from langchain_core.messages import HumanMessage
```

✔️ What it does

Represents a message **coming from the user**.
LLMs expect messages with roles: system, assistant, human.

✔️ Example

```python
from langchain_core.messages import HumanMessage

msg = HumanMessage(content="What is an AI agent?")
```

✔️ Using it in a chain

```python
messages = [HumanMessage(content="Explain CrewAI")]
response = llm.invoke(messages)
print(response.content)
```

This is the correct format for LangChain chat models.

🎉 Summary Table

Import	Purpose	Small Example
`ChatPromptTemplate`	Build multi-message prompts	`prompt.format(topic="Agentic AI")`
`MessagesPlaceholder`	Insert chat history into prompt	`history=[HumanMessage(...)]`
`datetime`	Get current time for prompts	`datetime.now().isoformat()`
`ChatGoogleGenerativeAI`	Use Gemini models in LangChain	`llm.invoke("Hello")`

Import	Purpose	Small Example
AnswerQuestion / ReviseAnswer	Structured model for LLM output	`llm.with_structured_output(Model)`
HumanMessage	Represents user messages	`HumanMessage("Hi")`

If you want, I can create a **diagram showing how all these pieces fit together in your reflection agent**, or rewrite your entire agent pipeline in clean Gemini-friendly code.

↳
from_messages(
what is this

`from_messages` is a **constructor** used by `ChatPromptTemplate` to build a **chat-style prompt** from a list of messages.

Let's break it down clearly and simply👇

✅ What is `from_messages`?

It is a method that lets you create a `ChatPromptTemplate` using **multiple messages**, each with a **role** and **content**.

Syntax:

```python
ChatPromptTemplate.from_messages([
    ("system", "You are a helpful AI."),
    ("human", "Explain {topic}.")
])
```

This creates a prompt containing:

1. a **system message**
2. a **human message**
3. optional chat history
4. or any other message placeholders

✅ Why do we use `from_messages`?

Because LLMs like OpenAI and Gemini expect messages in this format:

- system → instructions

- human → user input
- assistant → AI responses

LangChain uses **`from_messages`** so you can build a **structured, multi-part prompt** instead of one long string.

🔍 Detailed Example

```python
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a coding expert."),
    ("human", "Explain how a for loop works."),
])
```

When you call:

```python
print(prompt.format())
```

It emits:

```scss
SystemMessage("You are a coding expert.")
HumanMessage("Explain how a for loop works.")
```

This is exactly what the LLM receives internally.

🧠 Example with a variable

```python
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a math tutor."),
    ("human", "Solve this problem: {question}")
])
```

Then at runtime:

```python
filled = prompt.format(question="What is 5 + 7?")
```

Produces:

```scss
SystemMessage("You are a math tutor.")
HumanMessage("Solve this problem: What is 5 + 7?")
```

🔥 Example with message history (using `MessagesPlaceholder`)

```python
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a conversation bot."),
    MessagesPlaceholder("history"),
    ("human", "{new_question}")
])
```

Then you do:

```python
history = [
    HumanMessage("Hello!"),
    HumanMessage("Can you teach me about LangChain?")
]

final = prompt.format(
    history=history,
    new_question="What is LCEL?"
)
```

This injects the **`history`** messages into the prompt.

🧱 Summary

Piece	Meaning
`ChatPromptTemplate.from_messages()`	Build a multi-message prompt
Roles (**`"system"`**, **`"human"`**)	Define who is speaking
Supports variables	**`{question}`** gets replaced at runtime
Works with LangChain LLMs	Produces correct message format

If you want, I can show how **`from_messages`** fits into your **reflection agent pipeline** or draw a diagram of how prompts flow into the LLM.

↳

HumanMessage
  what is this onw

Got it — you are asking:

> **"What is this one?"** → **`HumanMessage`**

Here is a **clear, simple explanation with examples**.

✅ What is `HumanMessage`?

`HumanMessage` is a **LangChain message object** that represents **a message sent by a human (the user)** in a chat conversation.

It is part of LangChain's internal message system:

LangChain Message Type	Meaning
`SystemMessage`	Instructions / rules
`HumanMessage`	User input
`AIMessage`	Model output
`ToolMessage`	Tool result
`MessagesPlaceholder`	Slot for history

🧠 Why use `HumanMessage`?

Because chat-based LLMs (OpenAI, Gemini, Anthropic) expect input in **message format**, not plain strings.

Instead of writing:

```python
llm.invoke("Explain RAG")
```

LangChain lets you send **proper chat messages**:

```python
from langchain_core.messages import HumanMessage

llm.invoke([HumanMessage(content="Explain RAG")])
```

This closely matches how LLMs process role-based messages.

🔥 Basic Example

```python
from langchain_core.messages import HumanMessage
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash-exp")

messages = [
    HumanMessage(content="Explain AI agents in simple terms.")
]

response = llm.invoke(messages)
```

```
print(response.content)
```

Here:

- **`HumanMessage("...")`** tells the model **the user said this**.
- **`response`** will be an **`AIMessage`** object.

🎯 Example inside your agent

Your agent code uses:

```python
"messages": [HumanMessage("AI Agents taking over content creation")]
```

This means:

➡️ "The user asked: *AI Agents taking over content creation*"

LangChain will convert this into the format Gemini expects:

```json
{
  "role": "user",
  "content": "AI Agents taking over content creation"
}
```

🧱 Example with multiple steps (history)

```python
history = [
    HumanMessage(content="Hi"),
    HumanMessage(content="What are agents?"),
    HumanMessage(content="Explain reflection.")
]
```

Putting it in the prompt:

```python
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are an expert."),
    MessagesPlaceholder("history"),
    ("human", "Now answer: {question}")
])
```

Runtime:

```python
prompt.format(
    history=history,
    question="What is LangGraph?"
)
```

Now the model receives the entire conversation.

🧠 Summary

`HumanMessage` is:

- a **wrapper** for user text
- used to build chat conversations
- required in LangChain for structured multi-turn agents
- used by LangGraph steps
- compatible with Gemini / OpenAI / Claude

If you want, I can also explain:

- `AIMessage`
- `SystemMessage`
- how they interact
- why LangGraph nodes often output lists of messages

Just ask!

actor_prompt_template = ChatPromptTemplate([
    (
        "system",
        """
            You are expert AI researcher.
            Current time : {time}

            1. {first_instruction}
            2. Reflect and critique your answer. Be serve to maximize improvement.
            3. After the reflection , *** list 1-3 search queries separately *** for researchinf
improvements. Do not include the inside the reflection.

        """
    ),
    MessagesPlaceholder(variable_name="messages"),
    ("system","Answer the user's question above using the required format.")
]).partial(time = lambda: datetime.datetime.now().isoformat())
explain this whats going here

Let's break this down **line by line** in the simplest possible way.
This is an extremely important part of your agent — it controls **how the model thinks and what format it must output**.

Here is the code:

```python
actor_prompt_template = ChatPromptTemplate([
    (
        "system",
        """
            You are expert AI researcher.
            Current time : {time}

            1. {first_instruction}
            2. Reflect and critique your answer. Be serve to maximize improvement.
            3. After the reflection , *** list 1-3 search queries separately *** for researchinf
improvements. Do not include the inside the reflection.

        """
    ),
    MessagesPlaceholder(variable_name="messages"),
    ("system","Answer the user's quection above using the required format.")
]).partial(time = lambda: datetime.datetime.now().isoformat())
```

✅ 1. `ChatPromptTemplate([...])`

`ChatPromptTemplate` is used to build a **structured conversation prompt**.

It takes a list of messages:

- system messages
- human messages
- placeholders for past messages

This prompt will eventually be passed to the LLM.

✅ 2. First message: System message

```python
(
    "system",
    """
        You are expert AI researcher.
        Current time : {time}

        1. {first_instruction}
        2. Reflect and critique your answer. Be serve to maximize improvement.
        3. After the reflection , *** list 1-3 search queries separately *** for researching
improvements. Do not include them inside the reflection.

    """
),
```

✔️ What this does:

This is a **system / instruction message** telling the AI how to behave.

Key parts:

- `You are expert AI researcher.`
  → Sets the "role" of the AI.
- `Current time : {time}`
  → A variable that will be replaced later.
- `1. {first_instruction}`
  → This is a placeholder that will later contain something like *"Provide a detailed ~250 word answer"* or *"Revise your previous answer"*.
- **Step 2: Reflect and critique the answer**
  → Forces the AI to evaluate its own work.
- **Step 3: List 1–3 search queries**
  → These are used for next steps in the reflection pipeline.

This entire block is a **dynamic instruction template**.

✅ 3. `MessagesPlaceholder(variable_name="messages")`

```python
MessagesPlaceholder(variable_name="messages")
```

✔️ What this does:

It tells LangChain:

> "At runtime, insert all previous chat messages right here."

For example:

```python
"messages": [
    HumanMessage("Explain AI Agents"),
    AIMessage("Sure…"),
    HumanMessage("Now explain reflection")
]
```

This makes the agent **multi-turn**.

✅ 4. Final system instruction

```python
("system", "Answer the user's question above using the required format.")
```

This is another system message reinforcing:

> "Follow the required format. Don't freestyle."

This prevents junk output.

⭐ 5. `.partial(time = lambda: datetime.datetime.now().isoformat())`

This is VERY important.

✔️ What `.partial()` does:

It **pre-fills variables** in the prompt template.

Your template includes a placeholder:

```bash
{time}
```

`partial()` automatically fills it every time you call the prompt with:

```python
lambda: datetime.datetime.now().isoformat()
```

Example:

```python
2025-12-09T11:10:54.392182
```

Without `.partial()`

You would have to pass **`"time": datetime.datetime.now()`** every time manually.

🧠 Putting it all together

This code creates a **universal prompt** that:

1. Injects dynamic instructions (**`first_instruction`**)
2. Injects the current time (**`time`**)
3. Inserts conversation history (**`messages`**)
4. Forces the AI to:

- give an answer
- critique itself
- list search queries

5. Ensures the output is in a structured format

This is PERFECT for **reflection agents**.

🔥 Example of how this prompt is used:

For first responder agent:

```
python

first_responder_prompt = actor_prompt_template.partial(
    first_instruction="Provide a detailed ~250 word answer."
)
```

For revision agent:

```
python

revisor_prompt = actor_prompt_template.partial(
    first_instruction="Revise your answer using critiques and citations."
)
```

The **same base prompt** → different purposes.

🎉 Summary

Component	Meaning
`ChatPromptTemplate([...])`	Build a structured prompt
First system message	Defines AI role + instructions + placeholders
`MessagesPlaceholder("messages")`	Insert chat history at runtime
Last system message	Forces answer format
`.partial(time = ...)`	Automatically add current time

If you want, I can draw a **flow diagram** showing how this template integrates into the whole Gemini reflection agent pipeline.

MessagesPlaceholder(variable_name="messages"), whats going here what is variable_name here

Great — let's break this line down **very clearly** because it's one of the most important concepts in LangChain prompts:

✅ The code in question

```python
MessagesPlaceholder(variable_name="messages")
```

🚀 What is MessagesPlaceholder?

`MessagesPlaceholder` is a special LangChain object used inside a **`ChatPromptTemplate`** to tell LangChain:

> "At this position in the prompt, insert a list of chat messages."

This allows you to dynamically inject **conversation history** or any sequence of **`HumanMessage`** / **`AIMessage`**.

🎯 What does `variable_name="messages"` mean?

It means:

- This placeholder expects a **variable** called **`"messages"`** when you call the prompt.
- Whatever you pass into the prompt as **`messages=`** will be inserted at this position.

Example:

```python
prompt.invoke({
    "messages": [
        HumanMessage("Hello!"),
        HumanMessage("Explain LangGraph"),
    ],
    "question": "What is reflection?"
})
```

The placeholder expands to:

```makefile
Human: Hello!
Human: Explain LangGraph
```

Placed *inside the prompt exactly where **`MessagesPlaceholder`** was written.*

🔍 Why do we need a variable name?

Because **`ChatPromptTemplate`** can have **many placeholders**, and LangChain must know:

- **Which variable should fill which placeholder**

- **Where in the prompt the list of messages goes**

For example:

```python
MessagesPlaceholder("history")
MessagesPlaceholder("feedback")
("human", "{question}")
```

You might call:

```python
prompt.invoke({
    "history": [...],
    "feedback": [...],
    "question": "Explain RAG"
})
```

Without a variable name, LangChain would not know what messages to insert.

🧠 Visual Explanation

Your template:

```python
[
    ("system", "...instructions..."),
    MessagesPlaceholder(variable_name="messages"),
    ("system", "Answer the user's question...")
]
```

If you use:

```python
messages=[
    HumanMessage("What are AI agents?"),
    HumanMessage("Explain reflection agents.")
]
```

Then the filled prompt becomes:

```vbnet
System: ...instructions...

Human: What are AI agents?
Human: Explain reflection agents.

System: Answer the user's question...
```

🧪 Full Example

🔹 Define prompt

```python
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful assistant."),
    MessagesPlaceholder("messages"),
    ("human", "{final_question}")
])
```

🔹 Call prompt

```python
prompt.invoke({
    "messages": [
        HumanMessage(content="Hi, teach me ML."),
        HumanMessage(content="Now teach me LangChain.")
    ],
    "final_question": "What is LCEL?"
})
```

🔹 Output prompt sent to LLM

```vbnet
System: You are a helpful assistant.

Human: Hi, teach me ML.
Human: Now teach me LangChain.

Human: What is LCEL?
```

🧱 Summary

Concept	Meaning
`MessagesPlaceholder`	Tells LangChain to insert a list of messages here
`variable_name="messages"`	The name of the variable you pass in at runtime
Purpose	Supports multi-turn conversation / history injection
Example variable value	A list of **`HumanMessage`**, **`AIMessage`**, etc.

If you'd like, I can also explain:

- **`SystemMessage`**
- **`AIMessage`**
- How messages flow inside an **agent loop**
- How this connects to LangGraph state

Just tell me!

MessagesPlaceholder why is this after the system message

Great question — **the position of `MessagesPlaceholder` in the prompt matters a lot**, and most people misunderstand *why it is placed right after the system message.*

Let's explain this clearly and visually.

✅ Why is `MessagesPlaceholder` placed *after* the system message?

Because your prompt is intended to look like a **real conversation**:

```makefile
System: instructions for the AI
⬇
History: previous interactions
⬇
System/human: final instructions or question
```

This ordering is **intentional** and very important.

🧠 Reason 1 — System instructions should ALWAYS come first

The **system message** defines:

- who the AI is
- how it should behave
- what format to follow

Example:

```sql
System: You are an expert AI researcher
```

You ALWAYS want this message to come **before** any chat history, so the AI sees these instructions first.

🧠 Reason 2 — History needs to be injected into the middle of the prompt

`MessagesPlaceholder` inserts **previous messages** (conversation history).

You want the model to see:

1. **System rules**

2. **Conversation so far**
3. **Final instruction**

This exactly mirrors a real chat interface.

If you put history *before* the system message, the AI would interpret the system instructions as just another chat message — BAD.

If you put history *after* the final question, the AI will get confused because the older messages appear *after* the new ones — also BAD.

🧠 Reason 3 — The final instruction must appear LAST

After all context is given, you want to end with:

```arduino
("system", "Answer the user's question using the required format.")
```

This ensures the final thing the AI sees is the format requirement — VERY IMPORTANT in structured generation.

Imagine your prompt structure:

```yaml
SYSTEM: Instructions
⬇
MESSAGESPLACEHOLDER: Chat history
⬇
SYSTEM again: Required output format
⬇
LLM produces answer
```

This is the optimal order.

🔍 Let's visualize your full prompt

Your template:

```python
[
    ("system", "You are expert AI researcher. ..."),
    MessagesPlaceholder("messages"),
    ("system", "Answer the user's question above using the required format.")
]
```

What the LLM actually sees (example)

```sql
System: You are expert AI researcher.
System: Current time: 2025-12-09
```

```
System: Instructions...
------------------------------------
Human: Hi, explain AI agents
AI: Sure...
Human: Explain LangGraph
------------------------------------
System: Answer the user's question using the required format.
```

This structure ensures:

- The model understands its role
- The model sees the historical conversation
- The model understands the final instruction before output

🎯 Summary Table

Element	Why it is there	Why placed in that order
System message (top)	Define AI role, rules	Must come first for correct behavior
MessagesPlaceholder	Insert conversation history	History belongs **after rules**, before final task
Final system message	Enforce required output format	Must come last so model does not forget instructions

---

🚀 Analogy

Imagine you are a researcher assistant:

1. Your boss gives you instructions → **system message**
2. You review all previous emails → **messages placeholder**
3. Your boss gives final task: "Now answer this in this format" → **final system message**

This is exactly how this prompt works.

If you want, I can also explain:

- Why some agent frameworks put MessagesPlaceholder at the bottom
- How to design perfect prompts for reflection agents
- How LangGraph handles messages internally

Just tell me!

```
import json
from typing import List, Dict, Any
from langchain_core.messages import AIMessage, BaseMessage, ToolMessage, HumanMessage
```

from langchain_community.tools import TavilySearchResults
explain these with examples

Here is a **clear explanation of each import**, what it does, and **simple examples** you can run.
These are important when building **agent loops, tool-calling, and message histories** in LangChain / LangGraph.

✅ 1. `import json`

✔️ What it does

Python's built-in module for working with JSON:

- Convert JSON string → Python dict
- Convert dict → JSON string
- Save JSON files
- Load JSON files

✔️ Example

```python
import json

data = {"name": "Dulama", "role": "AI researcher"}

# Convert to JSON string
json_str = json.dumps(data)
print(json_str)

# Convert back to Python dict
loaded = json.loads(json_str)
print(loaded["name"])
```

✅ 2. `from typing import List, Dict, Any`

These are **type hints** used for documenting the expected types of variables.

✔️ `List`

Represents a list of elements.

```python
from typing import List

names: List[str] = ["Alice", "Bob", "Charlie"]
```

✔️ `Dict`

Represents a dictionary.

```python
from typing import Dict

student: Dict[str, int] = {"math": 95, "science": 88}
```

✔️ `Any`

Represents a variable of *any* type.

```python
from typing import Any

value: Any = 42
value = "hello"
```

Useful when the structure is unknown (LLM outputs, tool responses, etc.).

✅ 3. `AIMessage`, `BaseMessage`, `ToolMessage`, `HumanMessage`

```python
from langchain_core.messages import AIMessage, BaseMessage, ToolMessage, HumanMessage
```

LangChain uses **message classes** to represent chat messages.

These are essential in **LangGraph** because each node deals with lists of messages.

🔵 HumanMessage

Represents **a message from the user**.

Example:

```python
from langchain_core.messages import HumanMessage

msg = HumanMessage(content="Explain LangGraph.")
print(msg)
```

LLM sees this as:

```json
{"role": "user", "content": "Explain LangGraph."}
```

🤖 AIMessage

Represents **the response from the AI model**.

Example:

```
python

from langchain_core.messages import AIMessage

msg = AIMessage(content="LangGraph is a framework for agent workflows.")
print(msg)
```

LLM sees it as:

```
json

{"role": "assistant", "content": "..."}
```

🔧 ToolMessage

This represents **the response of a tool** (like web search, calculations, database lookup).

When the LLM calls a tool, the tool's output is wrapped in **`ToolMessage`**.

Example:

```
python

from langchain_core.messages import ToolMessage

tool_output = ToolMessage(
    content="Search results: AI agents are evolving...",
    tool_call_id="search_1"
)

print(tool_output)
```

Used in Agent loops:

1. AIMessage: "Call tool search with query X"
2. ToolMessage: tool responds
3. AIMessage: model continues with tool result

🧱 BaseMessage

The **parent class** for all message types.

You rarely use it directly — but useful for type hints.

Example:

```python
from langchain_core.messages import BaseMessage

def print_message(msg: BaseMessage):
    print(msg.type, msg.content)

print_message(HumanMessage(content="Hi"))
print_message(AIMessage(content="Hello"))
```

Output:

```nginx
human Hi
ai Hello
```

🔍 Where these messages are used?

Example: History list

```python
history: List[BaseMessage] = [
    HumanMessage(content="Explain AI agents"),
    AIMessage(content="AI agents are..."),
    HumanMessage(content="Explain reflection agents")
]
```

You then plug this into a `MessagesPlaceholder` in your prompt.

✅ 4. Tavily search tool

```python
from langchain_community.tools import TavilySearchResults
```

✔️ What it does

A ready-made **web search tool** (Tavily API).
Useful in reflection agents, retrieval agents, and autonomous agents.

Example:

```python
from langchain_community.tools import TavilySearchResults

search = TavilySearchResults(max_results=3)

results = search.invoke({"query": "Latest research on Agentic AI"})
print(results)
```

The result is a list of:

- titles
- URLs
- summary snippets

🎉 Summary Table

Import	Meaning	Example
`json`	Convert to/from JSON	`json.loads("...")`
`List`	List type hint	`names: List[str]`
`Dict`	Dictionary type hint	`scores: Dict[str, int]`
`Any`	Any type	`x: Any`
`HumanMessage`	User message	`HumanMessage("Hi")`
`AIMessage`	Model response	`AIMessage("Hello")`
`ToolMessage`	Tool call response	wrapping tool outputs
`BaseMessage`	Parent class for messages	type hints
`TavilySearchResults`	Web search tool	`search.invoke({"query":"..."})`

If you want, I can show you a **full LangGraph example** that uses:

- HumanMessage
- AIMessage
- ToolMessage
- Tavily Search
- Reflection + Revision agent

Just tell me!

can you show me the example of history is is list

Absolutely — here are **clear, simple examples** showing how **history is a list** of messages (`HumanMessage`, `AIMessage`, `ToolMessage`) and how it is used in prompts and agent loops.

✅ 1. Basic Example of History (List of Messages)

```python
from langchain_core.messages import HumanMessage, AIMessage

history = [
    HumanMessage(content="Hi, can you help me study AI?"),
    AIMessage(content="Of course! What topic do you want to learn?"),
    HumanMessage(content="Explain LangGraph.")
]

print(history)
```

This is a **list** containing message objects:

```arduino
[
  HumanMessage("Hi, can you help me study AI?"),
  AIMessage("Of course! ..."),
  HumanMessage("Explain LangGraph.")
]
```

✅ 2. Using history with MessagesPlaceholder

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage, AIMessage

prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful tutor."),
    MessagesPlaceholder("history"),
    ("human", "{new_question}")
])

history = [
    HumanMessage(content="Hi"),
    AIMessage(content="Hello! How can I help?"),
    HumanMessage(content="Explain Agents."),
    AIMessage(content="Agents are ...")
]

result = prompt.format(
    history=history,
    new_question="Explain LangGraph"
)

print(result)
```

LLM sees this:

```makefile
System: You are a helpful tutor.

Human: Hi
AI: Hello! How can I help?
Human: Explain Agents.
AI: Agents are ...

Human: Explain LangGraph
```

✅ 3. History list in an agent state

In LangGraph, history is usually stored in the **state**:

```
python

from langchain_core.messages import BaseMessage, HumanMessage, AIMessage
from typing import List

state = {
    "messages": [
        HumanMessage("What is RAG?"),
        AIMessage("RAG means retrieval augmented generation."),
        HumanMessage("Explain vector embeddings.")
    ]
}

print(state["messages"])
```

Yes — the value of **`"messages"`** is a **list**.

✅ 4. Adding new messages to history

```
python

history = []

history.append(HumanMessage("What is LangChain?"))
history.append(AIMessage("LangChain is a framework for LLM applications."))

print(history)
```

Output:

```
csharp

[HumanMessage("What is LangChain?"), AIMessage("LangChain is a framework...")]
```

✅ 5. History with ToolMessage (tool calling)

```
python

from langchain_core.messages import HumanMessage, AIMessage, ToolMessage

history = [
    HumanMessage("Search for LangGraph documentation."),
    AIMessage(content="", tool_calls=[{"id": "1", "name": "search", "args": {"query":
"LangGraph"}}]),
    ToolMessage(content="Found 12 documents...", tool_call_id="1"),
    AIMessage("Here is the summary...")
]
```

This is how **agent loops** store tool interactions.

🧠 6. History passed into prompt inside LangChain chain

```python
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash-exp")

messages = [
    HumanMessage("Define agentic AI."),
    AIMessage("Agentic AI means systems that act with autonomy."),
]

response = llm.invoke(messages + [HumanMessage("Give 3 examples.")])
```

Yes — again, **messages is a list**.

🎉 Summary

Concept	Meaning
History	A **list of messages**
Each item	A **`HumanMessage`**, **`AIMessage`**, or **`ToolMessage`**
Used for	Multi-turn chat, LangGraph agent loops
Inserted via	**`MessagesPlaceholder("history")`**

If you want, I can also show:

- how LangGraph's **state** stores messages
- how to append messages correctly inside a node
- how Gemini structured output fits into message history

Just tell me!

```
# Function to execute search queries from AnswerQuestion tool calls
def execute_tools(state: List[BaseMessage]) -> List[BaseMessage]:
    last_ai_message: AIMessage = state[-1]

    # Extract tool calls from the AI message
    if not hasattr(last_ai_message, "tool_calls") or not last_ai_message.tool_calls:
        return []

    # Process the AnswerQuestion or ReviseAnswer tool calls to extract search queries
    tool_messages = []

    for tool_call in last_ai_message.tool_calls:
        if tool_call["name"] in ["AnswerQuestion", "ReviseAnswer"]:
            call_id = tool_call["id"]
```

```
        search_queries = tool_call["args"].get("search_queries", [])

        # Execute each search query using the tavily tool
        query_results = {}
        for query in search_queries:
            result = tavily_tool.invoke(query)
            query_results[query] = result

        # Create a tool message with the results
        tool_messages.append(
            ToolMessage(
                content=json.dumps(query_results),
                tool_call_id=call_id
            )
        )

    return tool_messages
explain this clearly
```

Here is a **very clear, step-by-step explanation** of what this function does, *why it exists*, and *how it works inside an agent loop or LangGraph*.

✅ What the function does (high level)

This function looks at the **last AIMessage** in the conversation history.
If that message contains **tool calls** (like **`AnswerQuestion`** or **`ReviseAnswer`**),
then it:

1. Extracts the **search queries** from the tool call
2. Runs them with the **Tavily search tool**
3. Wraps the results in a **`ToolMessage`**
4. Returns a list of messages to add back to the conversation state

This allows the agent to perform **actual web search** after generating search queries.

🧠 Why do we need this function?

Because of this flow:

```sql
AIMessage: "Call tool AnswerQuestion with search_queries=[...]"
→ Agent executes the tool
→ Tool returns results inside ToolMessage
→ Agent continues answering using the tool results
```

LangGraph requires you to manually run the tool logic if you're not using built-in agent node types.

This function is **that tool executor**.

🔎 Let's break down the code line-by-line

📌 1. Function signature

```
python

def execute_tools(state: List[BaseMessage]) -> List[BaseMessage]:
```

- `state` = a list of messages (history)
- returns another list of messages (tool outputs)

So this function is used inside a LangGraph node that updates conversation history.

📌 2. Get the last AI message

```
python

last_ai_message: AIMessage = state[-1]
```

- The last message **must** be from the AI
- This message contains the **tool call instructions**

Example last message:

```
json

{
  "role": "assistant",
  "tool_calls": [
      {"id": "123", "name": "AnswerQuestion", "args": {"search_queries": ["AI agent research"]}}
  ]
}
```

📌 3. Check if tools were requested

```
python

if not hasattr(last_ai_message, "tool_calls") or not last_ai_message.tool_calls:
    return []
```

If the AI did **not** ask to call a tool:

- do nothing
- return no tool messages

Example where this happens:

```vbnet
AI: "Hello, how can I help?"
```

This has **no tool call**, so no search to perform.

📌 4. Prepare empty list to store ToolMessages

```python
tool_messages = []
```

Eventually we will return a list of tool outputs like:

```python
[
    ToolMessage(content="...", tool_call_id="123")
]
```

📌 5. Loop through each tool call

```python
for tool_call in last_ai_message.tool_calls:
```

Example tool calls:

```python
[
  {"id": "1", "name": "AnswerQuestion", "args": {"search_queries": [...]}},
  {"id": "2", "name": "ReviseAnswer", "args": {"search_queries": [...]}},
]
```

📌 6. Only handle AnswerQuestion or ReviseAnswer tools

```python
if tool_call["name"] in ["AnswerQuestion", "ReviseAnswer"]:
```

Meaning:

- Ignore other tool calls
- Only extract search queries for these two tools

📌 7. Extract required fields

```python
call_id = tool_call["id"]
search_queries = tool_call["args"].get("search_queries", [])
```

- **`call_id`** links tool output back to the original tool call
- **`search_queries`** is a list like:

```python
["agentic AI examples", "LLM tool calling explained"]
```

📌 8. Execute Tavily search for each query

```python
query_results = {}
for query in search_queries:
    result = tavily_tool.invoke(query)
    query_results[query] = result
```

This does:

- Run Tavily search tool for each query
- Store results in a dictionary indexed by query text

Example output:

```python
{
  "AI agent research": [{"url": "...", "content": "..."}],
  "LangGraph tutorials": [{"url": "...", "content": "..."}]
}
```

📌 9. Build a ToolMessage

```python
tool_messages.append(
    ToolMessage(
        content=json.dumps(query_results),
        tool_call_id=call_id
    )
)
```

This creates a message telling the AI:

> “Here are the results from the tool call you requested.”

Every tool call must respond with a ToolMessage.

The LLM receives:

```php-template
<tool_output>
```

and can continue answering.

📌 10. Return all tool messages

```python
return tool_messages
```

These get appended to the conversation state.

The agent then continues with:

- AIMessage (tool call)
- ToolMessage (tool results)
- AIMessage (LLM processes results)

🔥 Example of full flow

Step 1 — AI produces search queries

```json
AIMessage:
{
  "tool_calls": [
    {
      "id": "1",
      "name": "AnswerQuestion",
      "args": {
        "search_queries": ["agentic AI", "LangGraph workflows"]
      }
    }
  ]
}
```

Step 2 — execute_tools processes it

Produces:

```arduino
ToolMessage(
    tool_call_id="1",
    content="{ 'agentic AI': [...], 'LangGraph workflows': [...] }"
)
```

OpenAI

Step 3 — AI uses tool results

Next prompt contains the ToolMessage, so the LLM can produce a revised answer using actual search results.

🎉 Summary

Step	Meaning
Read last AI message	Check if it requested tools
Extract search queries	From tool call args
Run Tavily search	For each query
Build ToolMessage	Attach results tied to tool call ID
Return	Updated tool output messages

This function **executes real-world searches** for the agent and is essential for grounded answers and improved reflection.

If you want, I can show you:

- how to integrate this function into a full **LangGraph node**
- the **state graph** around it
- how Gemini structured output interacts with tool calls

Just say yes!

↳

def execute_tools(state: List\[BaseMessage\]) -> List\[BaseMessage\]:
what is this state

Great — you're asking:

> **"What is `state` here? Why is it a List[BaseMessage]? What does it represent?"**

This is a **core concept** in LangChain / LangGraph, especially in agent workflows.
Let me explain it clearly.

✅ What is `state`?

In this line:

```python
def execute_tools(state: List[BaseMessage]) -> List[BaseMessage]:
```

`state` represents the **entire conversation history** (up to this point).
It is a **list of messages** exchanged between:

- the **human**
- the **AI model**
- any **tools**

So `state` is basically:

```mathematica
[Message 1, Message 2, Message 3, ...]
```

Every new message (human, AI, or tool output) gets appended to this list.

🧠 Why is the type `List[BaseMessage]`?

Because **all LangChain messages inherit from `BaseMessage`**, including:

- `HumanMessage`
- `AIMessage`
- `ToolMessage`

So the type signature:

```python
List[BaseMessage]
```

means:

> "This is a list containing ANY message type."

🔍 Example of what state actually looks like

Before tool call

```python
state = [
    HumanMessage(content="Tell me about AI agents."),
    AIMessage(content="Sure, I'll answer that..."),
    HumanMessage(content="Give me search queries too."),
    AIMessage(
        content="",
```

```
        tool_calls=[
            {
                "id": "1",
                "name": "AnswerQuestion",
                "args": {
                    "search_queries": [
                        "What are AI agents?",
                        "LangGraph agent workflow"
                    ]
                }
            }
        ]
    )
]
```

This represents a multi-turn conversation.

🔦 Why does `execute_tools` need `state`?

Because it needs to look at **the last AI message**:

```python
last_ai_message = state[-1]
```

Why?

- The last AI message contains the **tool calls** (like AnswerQuestion, ReviseAnswer)
- Those tool calls contain **search queries**
- The agent must run the tools (Tavily search) to produce **ToolMessages**

So **`state`** contains **everything the function needs**.

🔨 What `execute_tools` does with state

1. Look at the last message:

```python
last_ai_message = state[-1]
```

2. Check if that message contains tool calls:

```python
last_ai_message.tool_calls
```

3. Extract search queries:

```python
search_queries = tool_call["args"].get("search_queries")
```

4. Run the search tool:

```python
tavily_tool.invoke(query)
```

5. Return new messages:

```python
return [ToolMessage(...)]
```

🎯 State in LangGraph

If you build an agent with LangGraph, your state often looks like:

```python
class State(TypedDict):
    messages: List[BaseMessage]
```

Then each node modifies the state:

- LLM node adds an `AIMessage`
- Tool executor node adds a `ToolMessage`
- Human input node adds a `HumanMessage`

Everything is stored inside the `state["messages"]` list.

🧱 Visual Diagram

```scss
┌──────────┐
│  State   │
│ messages │ ──────────────────────────────┐
└──────────┘                               │
                                           ▼
                               ┌───────────────────┐
                               │  execute_tools()  │
                               └───────────────────┘
                          reads state[-1] (AIMessage)
                                    │
                                    ▼
                       extracts tool calls → runs search
                                    │
                                    ▼
                         returns ToolMessage(s) back
                                    │
                                    ▼
                 state.messages.append(ToolMessage(...))
```

🎉 Summary

Concept	Meaning
state	The entire conversation history (list of messages)
List[BaseMessage]	Any message type (AI, human, tool)
execute_tools()	Reads last AI message → runs tools → returns ToolMessages
Used in LangGraph	Nodes read & write to state for multi-step processing

---

If you want, I can show:

- how to define a **state graph** properly
- how to create an **LLM** → **Tool** → **LLM** loop
- how Gemini tool-calls differ from OpenAI tool-calls

Just tell me!

i want to know what is like list or dict or variable

Ahhh okay — now I understand what you really want:

> **You want to know what `state` *is* in simple terms — is it a list? a dict? a variable?**

Let's explain it super clearly 👇

✅ In your code: `state` is a *list*.

Specifically:

```python
state: List[BaseMessage]
```

This means:

- `state` is a **variable**
- containing a **list**
- and the list contains **messages** (HumanMessage, AIMessage, ToolMessage...)

✔️ So:

```scss
state → variable
List[...] → type (Python list)
BaseMessage → type of items inside the list
```

🧠 What does the list look like?

Example:

```python
state = [
    HumanMessage(content="Hi"),
    AIMessage(content="Hello! How can I help?"),
    HumanMessage(content="Explain AI agents"),
    AIMessage(content="AI agents are..."),
]
```

This is a **list** of messages.

🧾 What the Python list actually is

A list is written using `[]`.

Example:

```python
[1, 2, 3]
["a", "b", "c"]
[HumanMessage(...), AIMessage(...)]
```

So yes — **`state`** is a normal Python **list**.

🧱 Why is it not a dict?

A dictionary looks like:

```python
{"key": "value"}
```

Your code uses:

```python
state: List[BaseMessage]
```

This is **NOT a dict**.

It is a **list**, because the order of messages matters:

1. Human said something
2. AI replied
3. Tool responded
4. AI replied again

A dict does not preserve this action order in a meaningful way, but a list does.

🎯 Check the type

You can test:

```python
print(type(state))
```

Output:

```javascript
<class 'list'>
```

And each element:

```python
print(type(state[0]))
```

Example:

```javascript
<class 'langchain_core.messages.human.HumanMessage'>
```

🧩 Summary

Item	What it is
`state`	A **variable** that stores conversation history
`List[...]`	Python **list** (ordered collection)
`BaseMessage`	Parent class for AIMessage, HumanMessage, ToolMessage
Structure	**`state = [HumanMessage(), AIMessage(), ...]`**

So in simple terms:

> **`state` is just a list of messages. That's it.**

If you want...

I can also explain:

- Why LangGraph chooses state as a list
- How to convert state to a dict-based state
- Types of Python variables (list, dict, tuple, set...) in simple English

Just tell me!

can i replace it using any word right

Yes — **you can replace the variable name `state` with ANY word you want**, **as long as** you update it everywhere you use it.

For example:

This:

```python
def execute_tools(state: List[BaseMessage]):
    last_ai_message = state[-1]
```

Can be changed to:

```python
def execute_tools(history: List[BaseMessage]):
    last_ai_message = history[-1]
```

Or:

```python
def execute_tools(messages: List[BaseMessage]):
    last_ai_message = messages[-1]
```

Or even:

```python
def execute_tools(chat_log: List[BaseMessage]):
    last_ai_message = chat_log[-1]
```

🧠 BUT IMPORTANT:

The name doesn't matter —
the TYPE matters.

It must still be:

✔️ A **list**
✔️ Containing **BaseMessage** objects
(HumanMessage, AIMessage, ToolMessage)

So all of these are valid:

```python
conversation: List[BaseMessage]
history: List[BaseMessage]
msgs: List[BaseMessage]
chat_state: List[BaseMessage]
```

Python does not care what variable name you choose.

🎯 Summary

Variable name	Allowed?	Notes
`state`	✔️ yes	common in LangGraph
`history`	✔️ yes	very clear meaning
`messages`	✔️ yes	also very common
`banana`	✔️ yes 😂	works, but not recommended

As long as you **change the name everywhere**, it's completely fine.

If you want, I can show you:

- How to rename **`state`** everywhere in a LangGraph agent
- The recommended naming conventions (state vs history vs messages)

Just tell me!

is this pydantic or what

Ahh now I understand —
you are asking:

> **"Is this (the `state: List[BaseMessage]`) part Pydantic or something else?"**

➡️ **Answer: No — this is NOT Pydantic.**
It is **only Python typing + LangChain message objects.**

Let's break it down clearly 👇

✅ What you wrote is NOT Pydantic

```python
state: List[BaseMessage]
```

This is just:

✔️ A normal Python variable (`state`)

✔️ Annotated with a Python type hint (`List[BaseMessage]`)

✔️ Using LangChain message classes (`BaseMessage`)

There is **no Pydantic** involved here.

🧩 Why does it look confusing?

Because Pydantic also uses type hints:

```python
class Person(BaseModel):
    name: str
    age: int
```

But in your case, there is **no BaseModel** and no class, so it is NOT Pydantic.

Your function is pure Python:

```python
def execute_tools(state: List[BaseMessage]):
    ...
```

This is **just type annotation**, like:

```python
x: int = 5
names: List[str] = ["a", "b", "c"]
```

🧠 What is actually happening?

`state`

A normal Python **variable**.

`List[...]`

A Python **type hint** from **`typing`**.

`BaseMessage`

A LangChain class.

So:

```css
→ NOT a Pydantic model
→ NOT a Pydantic field
→ Just a function argument with a type annotation
```

🔍 Example without type hint

You could write:

```python
def execute_tools(state):
    ...
```

And it would work exactly the same.

The type hint is optional, just helps developers understand:

> "state should be a list of messages"

🧱 How Pydantic WOULD look (for comparison)

Pydantic always involves:

```python
from pydantic import BaseModel
```

Example:

```python
class Answer(BaseModel):
    text: str
    sources: List[str]
```

This is Pydantic.

Your code does **not** use **`BaseModel`**, so it is **not** Pydantic.

🎯 Summary

Code	Is it Pydantic?	Why
`state: List[BaseMessage]`	❌ No	Just a type hint
Function parameters	❌ No	Normal Python
LangChain message classes	❌ No	LangChain message system
Pydantic models like **`AnswerQuestion`**	✅ Yes	Because they inherit **`BaseModel`**

If you want, I can also explain:

- What exactly **is** Pydantic
- When to use Pydantic vs normal Python objects
- How Gemini uses Pydantic for structured output

Just let me know!

if not hasattr(last_ai_message, "tool_calls") or not last_ai_message.tool_calls:
  return []
can you explain how this work

Absolutely — let's break this line down **slowly and clearly**, because it is one of the most important checks in an agent loop.

Here is the code:

```python
if not hasattr(last_ai_message, "tool_calls") or not last_ai_message.tool_calls:
    return []
```

This line decides:

> **"Did the AI request any tool calls? If not → do nothing."**

Let's understand each part.

✅ Step 1 — What is `last_ai_message`?

It is the **latest AI message** in the conversation state:

```python
last_ai_message = state[-1]
```

This AIMessage may or may not contain tool calls.

Example with tool calls:

```json
{
  "content": "",
  "tool_calls": [
    {"name": "AnswerQuestion", "args": {...}}
  ]
}
```

Example without tool calls:

```json
{
  "content": "Hello, how can I help?",
}
```

✅ Step 2 — Understanding `hasattr(last_ai_message, "tool_calls")`

```python
hasattr(last_ai_message, "tool_calls")
```

This returns:

- **True** → if the message *has a field* named **`"tool_calls"`**
- **False** → if the message does NOT have that field

Example:

```python
msg = AIMessage(content="hello")
hasattr(msg, "tool_calls")   # False
```

If the model **didn't call a tool**, there is no **`.tool_calls`** attribute.

✅ Step 3 — Understanding `not last_ai_message.tool_calls`

If **`"tool_calls"`** exists but the list is **empty**, this also means:

➡️ No tools were requested.

Example:

```python
msg.tool_calls = []   # empty list
```

Then:

```python
not msg.tool_calls    # True
```

🎯 Putting it all together

```python
if not hasattr(last_ai_message, "tool_calls")
    or
   not last_ai_message.tool_calls:
```

This means:

> **If the message does NOT have tool_calls, OR tool_calls is empty → do nothing.**

Cases where this condition is True:

last_ai_message	Has tool_calls?	Empty?	Return?
`"Hello!"`	❌ No	—	Return `[]`
`{ "tool_calls": [] }`	✔️ Yes	✔️ Empty	Return `[]`
`{ "tool_calls": [...] }`	✔️ Yes	❌ Not empty	DO NOT return

📚 Example 1 — AI does NOT ask for tool call

```python
AIMessage(content="Hi")
```

Check:

- Does it have `.tool_calls`? → ❌ No
- Condition is true → `return []`

Meaning:
Do nothing because AI didn't ask for tools.

📚 Example 2 — AI asks for tool call

```python
AIMessage(tool_calls=[{"name": "AnswerQuestion", "args": {...}}])
```

Check:

- Has attribute? → ✔️ Yes
- Is it empty? → ❌ No
- Condition is false → continue processing

So the function will run the tool.

🔥 Why do we return `[]`?

Because this function is supposed to return a list of **ToolMessages**.

If the model didn't request any tool calls:

- There are no tool messages to return
- Returning an empty list means "no update to conversation"

This prevents errors and keeps the agent loop clean.

🧠 Summary

This check:

```python
if not hasattr(last_ai_message, "tool_calls") or not last_ai_message.tool_calls:
    return []
```

Means:

➡️ **If the model did not request a tool → stop and return nothing.**

This prevents running tool logic when it's not needed.

If you want, I can rewrite it in easier-to-read code or add print-debug versions so you can see how it behaves.

hasattr what is this how it works

`hasattr()` is a **built-in Python function** that checks whether an object has a specific **attribute**.

Think of it like asking:

> "Does this object contain this field or property?"

✅ Syntax

```python
hasattr(object, "attribute_name")
```

It returns:

- **True** → if the object *has* that attribute
- **False** → if the object *does NOT have* that attribute

✅ Simple Example

```python
class Person:
    def __init__(self):
        self.name = "Alice"

p = Person()

print(hasattr(p, "name"))      # True
print(hasattr(p, "age"))       # False
```

Why?

- **`Person`** has an attribute **`name`** → True
- It does NOT have **`age`** → False

🧠 Why use `hasattr`?

Because sometimes an object only *sometimes* contains a field.

Example:
An AIMessage **may or may not** have **`tool_calls`**.

So you check:

```python
hasattr(last_ai_message, "tool_calls")
```

If this returns False → there is NO tool call.

🔥 Example with AIMessage

Case 1: No tool calls

```python
msg = AIMessage(content="Hello")
hasattr(msg, "tool_calls")  # False
```

Case 2: Has tool calls

```python
msg = AIMessage(
    content="",
    tool_calls=[{"name": "search", "args": {"query": "AI"}}]
)
hasattr(msg, "tool_calls")  # True
```

🧪 Example with hasattr + getattr

Sometimes you use both:

```python
if hasattr(msg, "tool_calls"):
    print(getattr(msg, "tool_calls"))
```

`getattr()` retrieves the attribute value.

🎯 Why your code uses it

This line:

```python
if not hasattr(last_ai_message, "tool_calls") or not last_ai_message.tool_calls:
    return []
```

Means:

1. If the message does NOT HAVE a **`tool_calls`** attribute → do nothing
2. OR if it has **`tool_calls`** but they are empty → do nothing

This prevents errors and unnecessary tool execution.

🚀 Visual Understanding

Think of it like checking if a student has a notebook:

```python
has_notebook = hasattr(student, "notebook")
```

If they don't have it → you can't ask them to write in it.

🎉 Summary

Function	Meaning
`hasattr(obj, "x")`	Does **`obj`** have attribute **`"x"`**?
Returns	True/False
Useful for	Checking optional fields like **`tool_calls`**

If you want, I can show how Python objects store attributes internally or how **`getattr`** and **`setattr`** work.

↳
🔥 **Example with AIMessage** ============================ Case 1: No tool calls python Copy code `msg = AIMessage(content="Hello") hasattr(msg, "tool_calls") # False`
is this one dict or what

Great — now you are asking specifically:

> **“Is `AIMessage` a dict or what? What type of object is it?”**

Let’s answer clearly.

✅ `AIMessage` is NOT a dict.

`AIMessage` is a **Python class** (an object), created by LangChain to represent an LLM response.

It looks like this internally:

```python
AIMessage(content="Hello")
```

This is an **instance of a class**, not a dictionary.

You can check:

```python
type(msg)
```

Output:

```javascript
<class 'langchain_core.messages.ai.AIMessage'>
```

So it’s **not**:

- ❌ a dict (`{}`)

- ❌ a list (`[]`)
- ❌ a tuple

It is a **custom Python object with attributes**.

🔍 How does AIMessage store its data?

It stores fields like:

```python
msg.content             # "Hello"
msg.tool_calls          # maybe exists, maybe not
msg.response_metadata   # optional
```

Because it's a class, we use:

```python
msg.content
```

NOT:

```python
msg["content"]     # ❌ this would be for dicts
```

🧠 Why does `hasattr(msg, "tool_calls")` work?

Because **`AIMessage`** sometimes has **`tool_calls`**, sometimes not.

Example 1 — **without tool calls**

```python
msg = AIMessage(content="Hello")

hasattr(msg, "tool_calls")
# False → attribute does not exist
```

Example 2 — **with tool calls**

```python
msg = AIMessage(
    content="",
    tool_calls=[{"name": "search", "args": {"query": "AI"}}]
)

hasattr(msg, "tool_calls")
# True → attribute exists
```

OpenAI

🧱 So what exactly *is* `AIMessage`?

✔️ A Python class

✔️ Supports attributes

✔️ Used for chat history

✔️ Represents the LLM's role ("assistant")

Think of it like:

```python
class AIMessage:
    content: str
    tool_calls: Optional[List[dict]] = None
```

But more complex internally.

🧪 Compare with a dict

AIMessage (class)

```python
msg.content
msg.tool_calls
msg.additional_kwargs
```

dict

```python
msg["content"]
msg["tool_calls"]
```

These are completely different.

🎯 Summary

Thing	Explanation
`AIMessage`	A **Python object/class**, NOT a dict
Access fields	Using dot notation (**`msg.content`**)
Why hasattr?	Because the object might not have certain attributes
Example	**`hasattr(msg, "tool_calls")`**

If you want...

I can show you:

- The full internal structure of AIMessage
- Difference between AIMessage, HumanMessage, ToolMessage
- How messages are stored in LangGraph state

Just tell me!

search_queries = tool_call["args"].get("search_queries",[]) what does do this

Great question — this line is **very common in Python**, and understanding it will help you a lot.

Here's the code:

```python
search_queries = tool_call["args"].get("search_queries", [])
```

Let's break it down **exactly** and **simply** 👇

✅ 1. `tool_call["args"]` — this is a dictionary

In an OpenAI tool call, the structure looks like:

```python
tool_call = {
    "id": "1",
    "name": "AnswerQuestion",
    "args": {
        "answer": "...",
        "search_queries": ["AI agents", "LangGraph reflection"]
    }
}
```

So:

- **`tool_call["args"]`** is a **dictionary**
- Inside **`"args"`** you have fields given by the AI model

Example:

```python
args = tool_call["args"]
print(args)
```

Output:

```python
{
  "answer": "...",
  "search_queries": ["AI agents", "LangGraph reflection"]
}
```

✅ 2. `.get("search_queries", [])`

`.get()` is a **dictionary method**:

```python
dict.get(key, default_value)
```

Meaning:

> If the dictionary contains **`key`**, return its value.
> Otherwise, return the **`default_value`**.

🔥 So the full line means:

```python
search_queries = tool_call["args"].get("search_queries", [])
```

✔️ Try to get `"search_queries"` from the args

✔️ If it exists → return it

✔️ If it's missing → return an empty list `[]`

🧠 Why do we need this?

Because sometimes the LLM may **not** include **`"search_queries"`** in the output.

Example:
Bad output:

```python
"args": {
  "answer": "some answer"
}
```

In that case:

```python
tool_call["args"]["search_queries"]  # ❌ ERROR (KeyError)
```

But with `.get()`:

```python
tool_call["args"].get("search_queries", [])
```

It safely returns:

```python
[]
```

So your program does NOT crash.

🔍 Visual table

`tool_call["args"]` contains	Result
`"search_queries": ["a", "b"]`	`["a", "b"]`
`"search_queries": []`	`[]`
No `"search_queries"` key	`[]` (default)

🧪 Example 1: Key exists

```python
args = {"search_queries": ["AI", "agents"]}
search_queries = args.get("search_queries", [])
print(search_queries)
```

Output:

```css
['AI', 'agents']
```

🧪 Example 2: Key missing

```python
args = {"answer": "something"}
search_queries = args.get("search_queries", [])
print(search_queries)
```

Output:

```css
[]
```

No error — safe.

🧱 Summary

Part	Meaning
`tool_call["args"]`	A dictionary of arguments generated by the AI
`.get("search_queries", [])`	Safe way to access **`"search_queries"`** without errors
If key missing	Returns **`[]`** instead of crashing
What it produces	A list of search query strings

If you want, I can also show how **`args`** look in an actual **`AIMessage.tool_calls`** and how you debug them in real agent runs.

query_results = {}
for query in search_queries:
result = tavily_tool.invoke(query)
query_results[query] = result

tool_messages.append(
ToolMessage(
content = json.dumps(query_results),
tool_call_id=call_id

how these works

Let's break this down **slowly and clearly**, because this part is what actually **executes the search queries and returns the tool output** to the AI inside your agent loop.

Here is the code:

```python
query_results = {}
for query in search_queries:
    result = tavily_tool.invoke(query)
    query_results[query] = result

tool_messages.append(
    ToolMessage(
        content=json.dumps(query_results),
        tool_call_id=call_id
```

```
    )
)
```

✅ STEP 1 — `query_results = {}`

This creates an **empty dictionary**.

This will eventually hold:

```arduino
{
  "query1": result1,
  "query2": result2
}
```

It's a mapping of:

```nginx
search_query_text → search_result_from_tavily
```

✅ STEP 2 — Loop through each search query

```python
for query in search_queries:
```

If search_queries =

```python
["what is agentic ai", "langgraph reflection agent"]
```

Then the loop does:

1️⃣ **`query = "what is agentic ai"`**
2️⃣ **`query = "langgraph reflection agent"`**

✅ STEP 3 — Execute the Tavily search tool

```python
result = tavily_tool.invoke(query)
```

This calls the Tavily search API (via LangChain tool).

For example:

```python
result = tavily_tool.invoke("what is agentic ai")
```

It returns something like:

```python
[
  {"title": "Agentic AI Overview", "url": "...", "content": "..."},
  {"title": "New Research", "url": "...", "content": "..."}
]
```

So now:

```ini
result = list of search results
```

✅ STEP 4 — Store results in the dictionary

```python
query_results[query] = result
```

This means:

- The **query text** becomes the **key**
- The **search result** becomes the **value**

Example result after running 2 queries:

```python
query_results = {
    "what is agentic ai": [...search results list...],
    "langgraph reflection agent": [...search results list...]
}
```

This structure is easy for the AI to read.

🔥 STEP 5 — Convert dictionary → JSON string

```python
json.dumps(query_results)
```

Because **`ToolMessage`** requires **string content**, not dict.

It becomes:

```json
{
  "what is agentic ai": [ ... ],
  "langgraph reflection agent": [ ... ]
}
```

🎯 STEP 6 — Create a ToolMessage

```python
tool_messages.append(
    ToolMessage(
        content=json.dumps(query_results),
        tool_call_id=call_id
    )
)
```

What is happening here?

A ToolMessage is created that contains:

- **the JSON search results**
- **the tool_call_id**, which links this response back to the original AI tool call

So the agent sees:

```scss
AIMessage (tool call)
ToolMessage (tool results)
```

This allows the LLM to continue processing with **actual search results**.

🧠 Why do we need `tool_call_id`?

Because the AI may call **multiple tools** in one message.

Example:

```ini
tool_call_id = "12345"
```

So the model knows:

> "This ToolMessage is the response to my previous tool call with id 12345."

This is required by OpenAI / LangChain tool-calling logic.

🧱 Final Summary (Step by Step)

Step	Code	Meaning
1	`query_results = {}`	Create empty dict
2	`for query in search_queries:`	Loop through each query
3	`result = tavily_tool.invoke(query)`	Run Tavily search
4	`query_results[query] = result`	Store result under that query
5	`json.dumps(query_results)`	Convert to JSON string
6	`ToolMessage(..., tool_call_id=call_id)`	Tool output wrapped for AI

🔥 A real example

Assume:

```python
search_queries = ["ai agents", "langgraph reflection"]
```

After running the code:

```python
query_results = {
    "ai agents": [ ... results ... ],
    "langgraph reflection": [ ... results ... ]
}
```

The LLM receives a ToolMessage like:

```json
{
  "tool_call_id": "1",
  "content": "{ 'ai agents': [...], 'langgraph reflection': [...] }"
}
```

Now the AI can use this data to improve its answer.

If you want, I can show:

- How this integrates into a LangGraph node
- A full working example with Gemini
- How the LLM continues processing after the ToolMessage

Just say the word!

query_results[query] = result here [query] numbers right

Great question — **and NO, `[query]` is *not* numbers here.**
Let's explain this clearly.

✅ What is `query_results` here?

```python
query_results = {}
```

This is a **dictionary**, not a list.

💡 **Dictionaries use *keys*, not numbers.**
Keys can be:

- strings
- numbers
- tuples
- etc.

In this case, the keys are **query strings**.

✅ So what does `query_results[query] = result` mean?

It means:

> "Store **`result`** in the dictionary under the key **`query`**."

Example:

```python
query = "what is agentic ai"
result = ["doc1", "doc2"]

query_results[query] = result
```

This produces:

```python
{
  "what is agentic ai": ["doc1", "doc2"]
}
```

👉 **NOT using numbers**
👉 **Using the search query STRING as the key**

🔍 Visualizing it

❌ NOT this (list indexing):

```python
query_results[0] = result
```

✔️ THIS (dictionary key):

```python
query_results["ai agents"] = ["link1", "link2"]
```

So the dictionary ends up looking like:

```python
{
  "ai agents": [...search results...],
  "langgraph reflection": [...search results...]
}
```

🧠 Why use the query string as the key?

Because the AI can read the results like:

```css
results["ai agents"]
```

This makes the output **clear and labeled**.

If the dictionary used numbers (like 0, 1), the AI wouldn't know which result belongs to which query.

🧪 Example with two queries

```python
query_results = {}

query = "agentic ai"
result = ["paper1", "paper2"]
query_results[query] = result

query = "langgraph workflows"
result = ["url1", "url2"]
query_results[query] = result
```

Final dictionary:

```python
{
  "agentic ai": ["paper1", "paper2"],
```

```
  "langgraph workflows": ["url1", "url2"]
}
```

🎯 Summary

Code	Meaning
`query_results = {}`	Create empty dictionary
`query_results[query] = result`	Add entry where the **key is the query string**
Keys	NOT numbers — they are the search query strings
Purpose	LLM can clearly see which results match which query

If you'd like, I can also show:

- how lists (`[]`) differ from dicts (`{}`)
- how keys and values work
- print-debug examples so you can see the exact structure during runtime

Just say yes!

can you give structure of tool message

Yes! Here is the **exact structure of a `ToolMessage`** in LangChain — explained simply, with diagrams and examples.

✅ What is a ToolMessage?

A **`ToolMessage`** is a message sent **from a tool back to the AI model** after the model requested a tool call.

Example flow:

```sql
AIMessage  → Tool Call
ToolMessage → Tool Result
AIMessage  → Continues answer with tool data
```

So **`ToolMessage`** is the **tool's response**.

✅ ToolMessage Structure (Basic)

A **`ToolMessage`** has TWO main fields:

Field	Meaning
`content`	The tool output (string, usually JSON)
`tool_call_id`	ID of the tool call this response belongs to

Minimal structure:

```python
ToolMessage(
    content="...",          # tool response as string
    tool_call_id="12345"    # must match the tool call ID from AIMessage
)
```

🧠 WHY do we need `tool_call_id`?

Because the AI may call multiple tools at once.

Example AIMessage:

```json
{
  "tool_calls": [
    {"id": "1", "name": "search", "args": {...}},
    {"id": "2", "name": "calculator", "args": {...}}
  ]
}
```

So the agent must send back:

- ToolMessage for id **"1"**
- ToolMessage for id **"2"**

The model uses `tool_call_id` to match results to the right request.

📦 FULL INTERNAL STRUCTURE (LangChain)

If you inspect a ToolMessage:

```python
msg = ToolMessage(
    content="{'result': 'some data'}",
    tool_call_id="abc123"
)

print(msg)
```

Internally it stores:

```python
python

{
  "type": "tool",                  # message type
  "content": "...",                # string content
  "tool_call_id": "abc123",        # required linking
  "additional_kwargs": {}          # internal metadata (optional)
}
```

So the internal structure is basically a **dict**, but you access it as **object attributes**.

🔥 REAL EXAMPLE (your agent)

```python
python

query_results = {
  "ai agents": [...],
  "langgraph workflows": [...]
}

tool_msg = ToolMessage(
    content=json.dumps(query_results),
    tool_call_id="1"
)
```

This becomes:

```json
json

{
  "type": "tool",
  "content": "{\"ai agents\": [...], \"langgraph workflows\": [...]}",
  "tool_call_id": "1"
}
```

The AI then sees **ToolMessage** → **tools_response**, and uses the results.

🧪 Full flow example

Step 1 — AIMessage (tool call)

```json
json

{
  "tool_calls": [
    {
      "id": "1",
      "name": "search",
      "args": {"query": "agentic AI"}
    }
  ]
}
```

Step 2 — ToolMessage (response)

```json
{
  "tool_call_id": "1",
  "content": "{\"agentic AI\": [ ...results... ]}"
}
```

Step 3 — AI continues using tool output

```vbnet
AI: Based on the search results, here is the improved answer...
```

🧱 Summary Table

Component	Purpose
`tool_call_id`	Links tool result to the specific tool call
`content`	JSON string containing tool output
Message type	Always **`"tool"`**
Created by	You (in your execute_tools() function)
Used by	The next AI step to continue reasoning

⭐ FINAL TAKEAWAY

A **`ToolMessage`** is **not** a dict —
it is a **LangChain message object** with this structure:

```go
ToolMessage {
   content: string,
   tool_call_id: string,
   type: "tool"
}
```

It behaves **like a message** in the chat history.

If you want, I can also show:

- how to print a ToolMessage cleanly
- how Gemini handles tool messages
- how ToolMessages are used inside a LangGraph node

Just say the word!

{'capital of sri lanka,give single word answer': [{'url': 'https://en.wikipedia.org/wiki/Capital_of_Sri_Lanka', 'content': 'Wikipedia\nThe Free Encyclopedia\n\n## Contents\n\n# Capital of Sri Lanka\n\nThe current legislative capital of Sri Lanka is Sri Jayawardenepura Kotte and the executive and judicial capital is Colombo. Over the course of the island\'s history, the national capital has been in several locations other than Sri Jayawardenepura Kotte.\n\n| Part of a series on the | [...] | Royal Residences | Tambapaṇṇī Upatissagāma Vijithapura Anuradhapura Sigiriya Rohana Polonnaruwa Dambadeniya Yapahuwa Kurunegala Gampola Dedigama Sri Jayawardenepura Kotte Rayigama") Sitawaka Kandy |\n| Related dynasties | Kalinga "Kalinga (historical region)") Vanga Shakya Vedda Pandyan Mauryan Madurai Nayak |\n| Other | Sinhapura Sinhabahu Titles") Crown Jewels") Royal Standards") Pretenders") Abhisheka Vijayabā Kollaya | [...] | Epochs | Mahāvaṃsa Vijaya Tambapanni Anuradhapura Chola conquest of Anuradhapura Polonnaruwa Jaffna Dambadeniya Gampola Kotte Crisis of the Sixteenth Century Sitawaka Kandy Portuguese Ceylon Dutch Ceylon British Ceylon Kandyan Wars Uva Rebellion Matale rebellion Independence movement Dominion of Ceylon Civil war Aragalaya |\n| Topics | Chronicles") Monarchs Demographic") Economic Education Military") Sexual Minorities |'}, {'url': 'https://en.wikipedia.org/wiki/Sri_Lanka', 'content': "Jayawardenepura Kotte is the legislative capital of Sri Lanka, while the largest city, Colombo, is the administrative and judicial capital which is the nation's political, financial and cultural centre. Kandy is the second-largest city and also the capital of the last native kingdom of Sri Lanka. The majority of the population speak Sinhala, while Tamil is the second most-spoken language. They are spoken by approximately 17 million and 5 million people respectively. [...] | Capital | Sri Jayawardenepura Kotte (legislative) Colombo (executive and judicial) 6°56′N 79°52′E\ufeff / \ufeff6.933°N 79.867°E\ufeff / 6.933; 79.867 |\n| Largest city | Colombo |\n| Official languages and recognised national languages | Sinhala Tamil |\n| Recognised language | English |\n| Ethnic groups (2024) | 74.1% Sinhalese 12.3% Sri Lankan Tamils 10.5% Sri Lankan Moors 2.8% Indian Tamils 0.3% other |"}], 'who is anura kumara dissanayake,give single word answer': [{'url': 'https://en.wikipedia.org/wiki/Anura_Kumara_Dissanayake', 'content': 'Anura Kumara Dissanayake (born 24 November 1968), commonly referred to by his initials AKD, is a Sri Lankan politician who has served as the tenth president of Sri Lanka since 2024. Dissanayake is the first Sri Lankan president to be elected in a second round of vote counting, and the first not to be a member of the traditional political parties of Sri Lanka. He is also the ninth executive president of Sri Lanka, a constitutional distinction that separates the executive presidency established [...] 1. ^ Mallawarachi, Bharatha. "Who is Anura Kumara Dissanayake, Sri Lanka\'s new Marxist president?". AP News. Archived from the original on 9 October 2024. Retrieved 1 November 2024.\n2. ^ "Anura Kumara Dissanayake: who is Sri Lanka\'s new leftist president?". The Guardian. Archived from the original on 24 September 2024. Retrieved 1 November 2024. [...] | |\n| Personal details |\n| Born | Dissanayaka Mudiyanselage Anura Kumara Dissanayake (1968-11-24) 24 November 1968 (age 57) Dewahuwa, Ceylon |\n| Party | National People\'s Power |\n| Other political affiliations | Janatha Vimukthi Peramuna |\n| Spouse | Mallika Dissanayake |\n| Children | 1 |\n| Residence(s) | 462/20, Pannipitiya Road, Pelawatte, Battaramulla |\n| Education | Thambuttegama Central College |\n| Alma mater | University of Kelaniya |\n| Occupation | Politician |'}, {'url': 'https://www.britannica.com/biography/Anura-Kumara-Dissanayake', 'content': 'Anura Kumara Dissanayake (born November 24, 1968, Thambuttegama, Ceylon [now Sri Lanka]) is the ninth executive president of Sri Lanka (2024–) and leader of the leftist People's Liberation Front (Janatha Vimukthi Peramuna; JVP). Dissanayake's election marks a major turning point in Sri Lankan politics. Unlike most previous presidents, he does not come from a political family. Additionally, the JVP has a controversial past because of its violent Marxist insurrections in the 1970s and '80s, for [...] Since assuming office, Dissanayake has both proclaimed a "new era of renaissance" for Sri Lanka and stated that "we have deeply understood that we are going to get a challenging country....We don't believe that a government, a single party or an individual would be able to resolve this deep crisis."\n\nEthan Teekah'}], 'best engineering university in sri lanka,give single word answer': [{'url': 'https://edurank.org/engineering/lk/', 'content': "The best cities to study Engineering in Sri Lanka based on the number of universities and their ranks are Peradeniya, Moratuwa, Colombo, and Nugegoda.\n\n## Engineering subfields in Sri Lanka\n\nlogo [...] We don't distinguish between undergraduate and graduate programs nor do we adjust for current majors offered. You can find information about granted degrees on a university page but always double-check with the university website.\n\n## 1. University of Peradeniya\n\nSri Lanka Flag\n\nFor Engineering\n\nUniversity of Peradeniya logo\n\n## 2. University of Moratuwa\n\nSri Lanka Flag\n\nFor Engineering\n\nUniversity of Moratuwa logo\n\n## 3. University of Colombo\n\nSri Lanka Flag\n\nFor Engineering [...] Rajarata University of Sri Lanka logo\n\n## 9. Sabaragamuwa University of Sri Lanka\n\nSri Lanka Flag\n\nFor Engineering\n\nSabaragamuwa University of Sri Lanka logo\n\n## 10. Wayamba University of Sri Lanka\n\nSri Lanka Flag\n\nFor Engineering\n\nWayamba University of Sri Lanka logo\n\n## 11. South Eastern University of Sri Lanka\n\nSri Lanka Flag\n\nFor Engineering\n\nSouth Eastern University of Sri Lanka logo\n\n## 12. Uva Wellassa University\n\nSri Lanka Flag\n\nFor Engineering\n\nUva Wellassa University logo"}, {'url': 'https://www.scimagoir.com/rankings.php?sector=Higher+educ.&country=LKA&area=2200', 'content': '| | | | | |\n --- --- \n| | 1 (2540) | University of Moratuwa | LKA | University |\n| | 2 (2895) | University of Ruhuna | LKA | University |\n| | 3 (3141) | University of Peradeniya | LKA | University |\n| | 4 (3479) | University of Jaffna | LKA | University |\n| | 5 (3693) | University of Sri Jayewardenepura | LKA | University |\n|\n\nUniversity\nUniversity\nUniversity\nUniversity\nUniversity\nUniversity\nScimago Lab\nScopus [...] You must have one of these extentions in you browser\n\nSCImago Journal & Country Rank\nSCImago Institutions Rankings\nSCImago Media Rankings\nSCImago Iber\nSCImago Research Centers Ranking\nSCImago Graphica\nEdiciones Profesionales de la Información \nSCImago Lab\nSCImago Lab\nScimago Institutions Rankings\nopen menu\nclose menu'}]}

Great — what you posted is **exactly what the ToolMessage content looks like after you do:**

```python
content = json.dumps(query_results)
```

Let me explain **what this structure is** and **how to understand it**.

✅ What you posted is a Python dictionary → converted to JSON string

It has this shape:

```arduino
{
  "query1": [results],
  "query2": [results],
  "query3": [results]
}
```

Where:

- each **key** is the **search query text**
- each **value** is a **list of search result objects**
- each search result object contains:
  - `url`
  - `content`

This is exactly what we expected.

🔍 Let's rewrite your output in cleaner form

ToolMessage content (decoded from JSON):

```python
{
  "capital of sri lanka,give single word answer": [
    {
      "url": "https://en.wikipedia.org/wiki/Capital_of_Sri_Lanka",
      "content": "... extracted text ..."
    },
    {
      "url": "https://en.wikipedia.org/wiki/Sri_Lanka",
      "content": "... extracted text ..."
    }
  ],

  "who is anura kumara dissanayake,give single word answer": [
    {
      "url": "https://en.wikipedia.org/wiki/Anura_Kumara_Dissanayake",
      "content": "..."
    },
    {
      "url": "https://www.britannica.com/biography/Anura-Kumara-Dissanayake",
      "content": "..."
    }
  ],

  "best engineering university in sri lanka,give single word answer": [
    {
      "url": "https://edurank.org/engineering/lk/",
      "content": "..."
    },
    {
      "url": "https://www.scimagoir.com/...rankings",
      "content": "..."
    }
  ]
}
```

This is exactly how `query_results` is supposed to look.

🧠 Why does the dictionary look like this?

Because of this code:

```python
query_results[query] = result
```

✔️ `query` = your search query string

Example:

```arduino
"capital of sri lanka,give single word answer"
```

✔️ `result` = list of search results from Tavily

So the dictionary becomes:

```python
{
  "capital of sri lanka,give single word answer": [ ...list of results... ]
}
```

This is a **mapping of query → results**, which is perfect for the AI.

🔥 Why does the ToolMessage content look messy?

Because:

1. It is **JSON-encoded**
2. Tavily returns **large text**
3. JSON inside a print looks ugly

But this is exactly what the AI needs.

🎯 What the AI will do with this

When the ToolMessage is added to the conversation, the AI will see:

- all extracted summaries
- URLs
- content

Then it can:

- verify facts
- revise its answer
- add citations

- correct mistakes

🧱 Structure Summary

Your ToolMessage content is a:

✔️ Python dict

converted to

✔️ JSON string

containing

✔️ keys = search queries

✔️ values = list of Tavily search results

where each result has

✔️ url

✔️ content

Perfect and expected.

👍 If you want...

I can show:

- how to pretty-print this JSON
- how to extract specific fields
- how the AI processes this in the revision step
- how to display it in a cleaner table

Just tell me!

execute_tools can you give sample of this output

Sure — here are **clear, realistic examples** of what `execute_tools()` returns in different situations.

Remember:

```python
def execute_tools(state: List[BaseMessage]) -> List[BaseMessage]:
```

It **always returns a list of ToolMessage objects**.

✅ Case 1 — AI did NOT request tool calls

Example last AIMessage:

```python
AIMessage(content="Hello! How can I help?")
```

`execute_tools()` output:

```python
[]
```

Why?
Because there is no **`tool_calls`** field.

✅ Case 2 — AI requested ONE tool call with ONE search query

AIMessage:

```python
AIMessage(
    content="",
    tool_calls=[
        {
            "id": "1",
            "name": "AnswerQuestion",
            "args": {
                "search_queries": ["capital of sri lanka"]
            }
        }
    ]
)
```

Tavily result (simplified):

```python
{"url": "https://...", "content": "Sri Jayawardenepura Kotte ..."}
```

🔥 execute_tools output:

```python
[
    ToolMessage(
        content='{"capital of sri lanka": [{"url": "https://...", "content": "Sri Jayawardenepura Kotte ..."}]}',
        tool_call_id="1"
    )
]
```

It always returns a **list** containing **ToolMessage**.

✅ Case 3 — ONE tool call with MULTIPLE search queries

AIMessage asked:

```python
"search_queries": [
    "capital of sri lanka",
    "who is anura kumara dissanayake"
]
```

Assume Tavily returns 2 results per query.

🔥 execute_tools output:

```python
[
    ToolMessage(
        content=
        '{"capital of sri lanka": [ ... ], '
        '"who is anura kumara dissanayake": [ ... ]}',
        tool_call_id="1"
    )
]
```

Same pattern:

- dictionary of **`{query: results}`**
- JSON encoded
- packed inside ToolMessage

✅ Case 4 — MULTIPLE tool calls inside the same AIMessage

AI produced:

```python
tool_calls = [
  {
    "id": "1",
    "name": "AnswerQuestion",
    "args": {"search_queries": ["query1"]}
```

```
  },
  {
    "id": "2",
    "name": "ReviseAnswer",
    "args": {"search_queries": ["query2", "query3"]}
  }
]
```

🔥 execute_tools output:

```
python

[
    ToolMessage(
        content='{"query1": [...]}',
        tool_call_id="1"
    ),

    ToolMessage(
        content='{"query2": [...], "query3": [...]}',
        tool_call_id="2"
    )
]
```

A **separate ToolMessage per tool call**.

🎉 FINAL — The output ALWAYS looks like this

A list of ToolMessages:

```
arduino

[
  ToolMessage(
     content="<JSON string>",
     tool_call_id="<id>"
  ),
  ToolMessage(
     content="<JSON string>",
     tool_call_id="<id>"
  )
]
```

Each ToolMessage contains:

Field	Meaning
`content`	JSON string of **`{query: tavily_results}`**
`tool_call_id`	ID that links this message to the original tool call

💡 If you want

I can also show:

- how the AI uses this ToolMessage in the next step

- how to format the JSON to look pretty
- how to debug inside LangGraph
- how Gemini handles ToolMessages differently

Just tell me!

```
results = execute_tools(test_state)

print("Raw results:", results)

print("==================================================")

if results:
    parsed_content = json.loads(results[0].content)
    print("Parsed content:", parsed_content)
    print("==================================================")
```

Let's break down **exactly what this code does**, step-by-step, and what the output looks like.

✅ 1. Call the function

```python
results = execute_tools(test_state)
```

- **`test_state`** is your list of messages
- It includes the **last AIMessage** that may (or may not) contain **`tool_calls`**
- **`execute_tools()`** returns:

Either:

❌ No tool calls → an empty list:

```python
[]
```

✔️ Or a list of ToolMessage objects:

```python
[
    ToolMessage(
        content='{"capital of sri lanka": [...], "best engineering university": [...]}',
        tool_call_id='1'
```

```
    )
]
```

✅ 2. Print raw results

```python
print("Raw results:", results)
```

This prints the **ToolMessage objects**, not the JSON inside.

Example output:

```css
Raw results: [ToolMessage(content="{'capital....}", tool_call_id='1')]
```

It will look messy because ToolMessage is an object.

✅ 3. Check if results exist

```python
if results:
```

This means:

- If list is **not empty** → process it
- If list is **empty** → skip parsing

✅ 4. Parse JSON content

```python
parsed_content = json.loads(results[0].content)
```

What's happening:

- `results[0]` → the first ToolMessage
- `.content` → the JSON string inside it
- `json.loads()` → convert the JSON string back into a Python dict

So now:

```python
parsed_content = {
    "capital of sri lanka,give single word answer": [...],
    "who is anura kumara dissanayake,give single word answer": [...],
    "best engineering university in sri lanka,give single word answer": [...]
}
```

✅ 5. Print parsed content

```python
print("Parsed content:", parsed_content)
```

Now you will see a Python dict printed cleanly.

Example:

```bash
Parsed content: {
   'capital of sri lanka,give single word answer': [...],
   'who is anura ...': [...],
   'best engineering university ...': [...]
}
```

🔥 FULL OUTPUT EXAMPLE (Realistic)

If tool calls exist:

```bash
Raw results: [ToolMessage(content="{"capital of sri lanka":[{"url": "...", "content": "..."}],
"who is ...": [...]}",
 tool_call_id='1')]

=====================================================

Parsed content: {
  "capital of sri lanka": [
      {"url": "...", "content": "..."},
      {"url": "...", "content": "..."}
  ],
  "who is anura kumara dissanayake": [
      {"url": "...", "content": "..."},
      {"url": "...", "content": "..."}
  ]
}

=====================================================
```

If NO tool calls exist:

```markdown
Raw results: []
```

```
==========================================================
```

(No parsed content printed)

🎯 Summary

Step	Purpose
`execute_tools(test_state)`	Runs Tavily search and returns ToolMessages
Print raw results	Show ToolMessage objects
`json.loads(results[0].content)`	Convert JSON string → Python dict
Print parsed content	Show readable results

If you want, I can also:

- Show how to pretty-print JSON
- Show how to test with different fake states
- Show how to print each search result nicely

Just tell me!

how to get single query result

If you want to extract **just one query result** from the **`ToolMessage`** output, it's very easy — because after parsing the JSON, you get a normal **Python dictionary**:

```python
parsed_content = {
    "capital of sri lanka": [...],
    "who is anura kumara dissanayake": [...],
    "best engineering university in sri lanka": [...]
}
```

Each key is a **query string**,
each value is a **list of Tavily search results**.

Let's go step-by-step.

✅ 1. Parse the ToolMessage content first

```python
tool_msg = results[0]                    # first ToolMessage
```

```
parsed = json.loads(tool_msg.content)          # convert JSON → dict
```

✅ 2. Get a single query result by using the key

Example:

```python
query = "capital of sri lanka,give single word answer"
single_result = parsed.get(query)
print(single_result)
```

This gives you the **list** of results for that query:

```json
[
  {"url": "...", "content": "..."},
  {"url": "...", "content": "..."}
]
```

✅ 3. Get ONLY the first result for that query

```python
first_result = parsed[query][0]
print(first_result)
```

Output:

```json
{"url": "...", "content": "..."}
```

✅ 4. Get only the URL from the first result

```python
url = parsed[query][0]["url"]
print(url)
```

✅ 5. Get only the content from the first result

```python
content = parsed[query][0]["content"]
print(content)
```

🔥 Full Working Example

```python
results = execute_tools(test_state)

if results:
    parsed = json.loads(results[0].content)

    # Choose one query key
    query_key = "capital of sri lanka,give single word answer"

    # All results for that query
    all_results = parsed.get(query_key)
    print("All results:", all_results)

    # First result
    first_result = all_results[0]
    print("First result:", first_result)

    # Individual fields
    print("URL:", first_result["url"])
    print("Content:", first_result["content"])
```

🎯 Summary

Goal	Code
Get results for one query	`parsed[query]`
Get first result only	`parsed[query][0]`
Get URL	`parsed[query][0]["url"]`
Get content	`parsed[query][0]["content"]`

If you want, I can show how to:

- Loop over all queries
- Extract the **best** result
- Clean the text
- Create a nice pretty print output

Just tell me!

from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
import datetime
from langchain_google_genai import ChatGoogleGenerativeAI
from schema import AnswerQuection, ReviseAnswer
#from langchain_core.messages import HumanMessage

from dotenv import load_dotenv

```
load_dotenv()


actor_prompt_template = ChatPromptTemplate([
    (
        "system",
        """
            You are expert AI researcher.
            Current time : {time}

            1. {first_instruction}
            2. Reflect and critique your answer. Be serve to maximize improvement.
            3. After the reflection , *** list 1-3 search queries separately *** for researchinf
improvements. Do not include the inside the reflection.

        """
    ),

    MessagesPlaceholder(variable_name="messages"),

    ("system","Answer the user's question above using the required format.")

]).partial(time = lambda: datetime.datetime.now().isoformat())


first_responder_prompt_template = actor_prompt_template.partial(first_instruction="provide a
detailed ~250 word answer")

llm=ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    temperature=0
)

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)

# Revisor instructions
revise_instructions = """Revise your previous answer using the new information.
    - You should use the previous critique to add important information to your answer.
        - You MUST include numerical citations in your revised answer to ensure it can be verified.
        - Add a "References" section to the bottom of your answer (which does not count towards the
word limit). In form of:
            - [1] https://example.com
            - [2] https://example.com
    - You should use the previous critique to remove superfluous information from your answer and
make SURE it is not more than 250 words.
"""

first_responder_chain = first_responder_prompt_template | first_responder_llm
revisor_chain = actor_prompt_template.partial(first_instruction=revise_instructions) | revisor_llm
```

```
from typing import List
from langchain_core.messages import BaseMessage,ToolMessage
from langgraph.graph import END,MessageGraph
from chains import revisor_chain, first_responder_chain
from tool import execute_tools

graph = MessageGraph()
MAX_ITERATIONS = 2

#nodes
graph.add_node("draft",first_responder_chain)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_chain)

#edges
graph.add_edge("draft","execute_tools")
graph.add_edge("execute_tools","revisor")


def event_loop(state: List[BaseMessage]) -> str:
    count_tool_visit =  sum(isinstance(item,ToolMessage) for item in state)
    num_of_iterations = count_tool_visit
    if num_of_iterations > MAX_ITERATIONS:
        return END
    return "execute_tools"

"""✔️ Final Answer
Yes, it is okay to write -> str because Python does not enforce it, but it is not 100% correct.
The return type should ideally allow END, which is not a string.
"""
graph.add_conditional_edges("revisor",event_loop)
graph.set_entry_point("draft")

app = graph.compile()

print(app.get_graph().draw_mermaid())
app.get_graph().print_ascii()

response = app.invoke("Write about how small business can leverage AI to grow")
print(response)
```

Nice, you've basically built a **mini reflection agent with tools in LangGraph** 🧠🔁
Let me walk through what this whole thing is doing, step by step, and point out a couple of small things to be aware of.

1️⃣ `chains.py` – the two agent "roles"

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
import datetime
from langchain_google_genai import ChatGoogleGenerativeAI
from schema import AnswerQuection, ReviseAnswer
from dotenv import load_dotenv

load_dotenv()
```

- Loads env vars (`GOOGLE_API_KEY` etc.).
- Uses Gemini (`ChatGoogleGenerativeAI`).
- `AnswerQuection` / `ReviseAnswer` are your **Pydantic schemas** for structured output.

  Note: you probably meant `AnswerQuestion` not `AnswerQuection` (typo is fine as long as schema name matches).

Actor / base prompt

```python
actor_prompt_template = ChatPromptTemplate([
    (
        "system",
        """
            You are expert AI researcher.
            Current time : {time}

            1. {first_instruction}
            2. Reflect and critique your answer. Be serve to maximize improvement.
            3. After the reflection , *** list 1-3 search queries separately *** for researchinf
improvements. Do not include the inside the reflection.

        """
    ),
    MessagesPlaceholder(variable_name="messages"),
    ("system","Answer the user's quection above using the required format.")
]).partial(time = lambda: datetime.datetime.now().isoformat())
```

What this does:

- **System message**: tells the model to:
  1. Follow `first_instruction`
  2. Reflect + critique its own answer
  3. Output 1–3 `search_queries` **separately**

- `MessagesPlaceholder("messages")`: where the chat history will be inserted.
- Second system: "Answer using the required format."
- `.partial(time=...)`: automatically fills `{time}` with current ISO timestamp.

  ⚠️ One thing: in LangChain, you normally use
  `ChatPromptTemplate.from_messages([...])`
  not `ChatPromptTemplate([...])`.
  If it's working, fine, but if you get errors, change to:

  ```python
  actor_prompt_template = ChatPromptTemplate.from_messages([...])
  ```

First responder prompt

```python
first_responder_prompt_template = actor_prompt_template.partial(
    first_instruction="provide a detailed ~250 word answer"
)
```

Same base prompt, but now:

- `first_instruction` = "write ~250 word answer".

So this prompt is for the **draft** answer.

LLM & structured outputs

```python
llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    temperature=0
)

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)
```

- Base Gemini model.
- `with_structured_output(AnswerQuection)` → Gemini will return a **Pydantic object**: `AnswerQuection(answer, search_queries, reflection)`
- `ReviseAnswer` extends that with e.g. `references`, etc.

Chains

```python
first_responder_chain = first_responder_prompt_template | first_responder_llm
```

- Input: **`{"messages": [...]}`**
- Output: an **`AnswerQuection`** instance.

```python
revisor_chain = actor_prompt_template.partial(
    first_instruction=revise_instructions
) | revisor_llm
```

- Same prompt template, but **`first_instruction`** now = **revise answer with citations**.
- Output: a **`ReviseAnswer`** instance.

So you have:

- **draft chain** → first_responder_chain
- **revision chain** → revisor_chain

2️⃣ `graph.py` – your LangGraph wiring

```python
from typing import List
from langchain_core.messages import BaseMessage, ToolMessage
from langgraph.graph import END, MessageGraph
from chains import revisor_chain, first_responder_chain
from tool import execute_tools
```

- **`MessageGraph`**: a graph where **state = List[BaseMessage]`.
- **`execute_tools`**: your function that takes the last **`AIMessage.tool_calls`**, runs Tavily, and returns **`ToolMessage`**(s).

Build the graph

```python
graph = MessageGraph()
MAX_ITERATIONS = 2

# nodes
graph.add_node("draft", first_responder_chain)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_chain)
```

You've created 3 nodes:

- **`"draft"`** → uses **`first_responder_chain`**
- **`"execute_tools"`** → runs your Tavily search logic
- **`"revisor"`** → uses **`revisor_chain`**

Edges – control flow

```python
graph.add_edge("draft", "execute_tools")
graph.add_edge("execute_tools", "revisor")
```

So the normal flow is:

`draft → execute_tools → revisor`

Conditional loop

```python
def event_loop(state: List[BaseMessage]) -> str:
    count_tool_visit = sum(isinstance(item, ToolMessage) for item in state)
    num_of_iterations = count_tool_visit
    if num_of_iterations > MAX_ITERATIONS:
        return END
    return "execute_tools"
```

What this does:

- Counts **how many ToolMessage(s)** are in the state (i.e., how many times tools have been used).
- If tool usage > **`MAX_ITERATIONS`** (2), stop the graph → return **`END`**.
- Otherwise, go back to **`"execute_tools"`**.

Then:

```python
graph.add_conditional_edges("revisor", event_loop)
```

So from **`"revisor"`** node:

- If you haven't hit max iterations → go to **`"execute_tools"`** again.
- If you have → go to **`END`**.

Basically:

```text
draft → execute_tools → revisor
                   ↑        └─(if more loops allowed)──┘
                   |
```

> That comment in your code:
>
> ```python
> """✔ Final Answer
> Yes, it is okay to write -> str because Python does not enforce it, but it is not
> 100% correct.
> ```

```
    The return type should ideally allow END, which is not a string.
    """
```

is correct: type-wise, **`event_loop`** really returns either **`"execute_tools"`** (a str) or **`END`** (a sentinel). You *can* annotate it as **`-> str`**, Python won't enforce it, but a more precise type would be something like:

```python
from typing import Union
def event_loop(state: List[BaseMessage]) -> Union[str, type(END)]:
    ...
```

or just leave it untyped.

Entry point & app

```python
graph.set_entry_point("draft")
app = graph.compile()

print(app.get_graph().draw_mermaid())
app.get_graph().print_ascii()
```

- Entry node = **`"draft"`** → first step is the first_responder.
- **`draw_mermaid()`** + **`print_ascii()`** visualise the graph.

Invoking the graph

```python
response = app.invoke("Write about how small business can leverage AI to grow")
print(response)
```

Here, you pass a **plain string** input.

MessageGraph will typically turn this into a **`HumanMessage`** internally like:

```python
[HumanMessage(content="Write about how small business can leverage AI to grow")]
```

Then flow:

1. **`"draft"`** node:
    - Uses **`first_responder_chain`**
    - Produces a structured **`AnswerQuection`**
    - LangGraph converts that into an **`AIMessage`** containing the tool-call / structured content.
2. **`"execute_tools"`** node:

- Looks at last **`AIMessage.tool_calls`**
- Runs Tavily on **`search_queries`**
- Returns **`ToolMessage`** with JSON results.

3. **`"revisor"`** node:
   - Uses **`revisor_chain`** (ReviseAnswer model)
   - Produces improved answer + references.
4. **`event_loop`**:
   - If ToolMessages ≤ MAX_ITERATIONS → go back to **`"execute_tools"`** again.
   - Else → end and return final messages.

The **`response`** you print at the end is the **final message list**, usually something like:

```python
[
  HumanMessage(...),
  AIMessage(... draft ...),
  ToolMessage(... tavily results ...),
  AIMessage(... revised answer ...),
  ...
]
```

3️⃣ What you might want to check / tweak

1. **Use `from_messages` for the prompt**
   If you get any errors on **`ChatPromptTemplate([...])`**, switch to:

   ```python
   actor_prompt_template = ChatPromptTemplate.from_messages([
       ("system", "..."),
       MessagesPlaceholder("messages"),
       ("system", "...")
   ])
   ```

2. **Schema name typo**
   `AnswerQuection` is fine if your **`schema.py`** uses that name, just make sure it's consistent everywhere.
3. **MAX_ITERATIONS logic**
   Right now you count total ToolMessages. That's okay.
   If you ever want "max *revisions*", you could adjust it.

If you tell me:

- what output you're currently getting
- or if you hit an error when running this

I can point out the exact line to fix and show you how to pretty-print the final revised answer only (not the whole message list).

```
(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>python
reflexion_graph.py
---
config:
  flowchart:
    curve: linear
---
graph TD;
        __start__([<p>__start__</p>]):::first
        draft(draft)
        execute_tools(execute_tools)
        revisor(revisor)
        __end__([<p>__end__</p>]):::last
        __start__ --> draft;
        draft --> execute_tools;
        execute_tools --> revisor;
        revisor -.-> draft;
        revisor -.-> execute_tools;
        revisor -.-> __end__;
        classDef default fill:#f2f0ff,line-height:1.2
        classDef first fill-opacity:0
        classDef last fill:#bfb6fc

              +-----------+
              | __start__ |
              +-----------+
                    *
                    *
                    *
                +-------+
                | draft |
                +-------+
             ***         ...
            *               .
          **                 ...
+---------------+               .
| execute_tools |            ...
+---------------+           .
             ***         ...
                *       .
                 **   ..
              +---------+
              | revisor |
              +---------+
                    .
                    .
                    .
              +---------+
```

```
          | __end__ |
          +---------+
Traceback (most recent call last):
  File "D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent\reflexion_graph.py", line
39, in <module>
    response = app.invoke("Write about how small business can leverage AI to grow")
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line
2683, in invoke
    for chunk in self.stream(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line
2331, in stream
    for _ in runner.tick(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\runner.py", line 146,
in tick
    run_with_retry(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\retry.py", line 40, in
run_with_retry
    return task.proc.invoke(task.input, config)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\utils\runnable.py", line 608,
in invoke
    input = step.invoke(input, config)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\runnables\base.py",
line 5711, in invoke
    return self.bound.invoke(
           ^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_core\language_models\chat_models.py", line 395, in invoke
    self.generate_prompt(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_core\language_models\chat_models.py", line 1025, in generate_prompt
    return self.generate(prompt_messages, stop=stop, callbacks=callbacks, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_core\language_models\chat_models.py", line 842, in generate
    self._generate_with_cache(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_core\language_models\chat_models.py", line 1091, in _generate_with_cache
    result = self._generate(
             ^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_google_genai\chat_models.py", line 940, in _generate
    request = self._prepare_request(
              ^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_google_genai\chat_models.py", line 1170, in _prepare_request
    system_instruction, history = _parse_chat_history(
                                  ^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
```

```
packages\langchain_google_genai\chat_models.py", line 404, in _parse_chat_history
    raise ValueError(
ValueError: Unexpected message with type <class
'langchain_core.messages.system.SystemMessage'> at the position 2.
During task with name 'draft' and id '54f86bc4-bb3e-34e6-b587-8b91bd135b77'
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1765302690.502481   24924 init.cc:232] grpc_wait_for_shutdown_with_timeout() timed
out.

(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>
```

Nice, you actually got the graph running — the error now is from **Gemini**, not LangGraph or your code structure.

Here's the key part of the error:

> **`ValueError: Unexpected message with type SystemMessage at the position 2.`**
> during task with name 'draft'

So let's decode this.

1️⃣ What's going wrong?

Your prompt template:

```python
actor_prompt_template = ChatPromptTemplate([
    (
        "system",
        """
            You are expert AI researcher.
            Current time : {time}

            1. {first_instruction}
            2. Reflect and critique your answer. Be serve to maximize improvement.
            3. After the reflection , *** list 1-3 search queries separately *** for researchinf
improvements. Do not include the inside the reflection.

        """
    ),

    MessagesPlaceholder(variable_name="messages"),

    ("system","Answer the user's quection above using the required format.")

]).partial(time = lambda: datetime.datetime.now().isoformat())
```

When LangChain renders this for Gemini, it becomes a list of messages like:

1. **SystemMessage** → your big system prompt
2. **(from MessagesPlaceholder)** → history (Human/AI/Tool messages)
3. **SystemMessage** → **`"Answer the user's question above using the required format."`**

🔴 **Problem:**

`langchain_google_genai` (Gemini wrapper) only allows **one system message at the very beginning**.
It does **not** allow another `SystemMessage` later in the list (at position 2), so it crashes.

That's exactly what this line means:

```
Unexpected message with type SystemMessage at the position 2.
```

2️⃣ How to fix it

You need to **remove that second system message** and fold its content into the first system block.

✅ Fixed `actor_prompt_template`

Also switch to `from_messages` (recommended):

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
import datetime

actor_prompt_template = ChatPromptTemplate.from_messages([
    (
        "system",
        """
You are expert AI researcher.
Current time: {time}

You must follow this process:

1. {first_instruction}
2. Reflect and critique your answer. Be severe to maximize improvement.
3. After the reflection, list 1–3 search queries separately for researching improvements.
   - Do NOT include the search queries inside the reflection.
4. Finally, answer the user's question above using the required format.
        """.strip()
    ),
    MessagesPlaceholder(variable_name="messages"),
])
# auto-fill time
actor_prompt_template = actor_prompt_template.partial(
    time=lambda: datetime.datetime.now().isoformat()
)
```

Notice:

- Only **one** `"system"` message now.
- That last instruction *"Answer the user's question..."* is merged into the first system block.
- `MessagesPlaceholder` stays where it is.

Your other code can stay as-is:

```python
first_responder_prompt_template = actor_prompt_template.partial(
    first_instruction="provide a detailed ~250 word answer"
)

llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
```

```
    temperature=0
)

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)

first_responder_chain = first_responder_prompt_template | first_responder_llm

revisor_chain = actor_prompt_template.partial(
    first_instruction=revise_instructions
) | revisor_llm
```

And the graph part doesn't need to change.

3️⃣ Why this fixes it

Now Gemini sees messages like:

1. **`SystemMessage`** → (big instructions, including "answer using format...")
2. **`HumanMessage`** / **`AIMessage`** / **`ToolMessage`** from history (via **`MessagesPlaceholder`**)

✅ No second **`SystemMessage`** at position 2 → **no error**.

If you want, next step I can help you:

- print **only the final revised answer** from **`response`**
- or add a small debug node to see what **`AnswerQuection`** / **`ReviseAnswer`** actually look like.

```
(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>python
reflexion_graph.py
---
config:
  flowchart:
    curve: linear
---
graph TD;
        __start__([<p>__start__</p>]):::first
        draft(draft)
        execute_tools(execute_tools)
        revisor(revisor)
        __end__([<p>__end__</p>]):::last
        __start__ --> draft;
        draft --> execute_tools;
        execute_tools --> revisor;
        revisor -.-> draft;
        revisor -.-> execute_tools;
        revisor -.-> __end__;
        classDef default fill:#f2f0ff,line-height:1.2
        classDef first fill-opacity:0
        classDef last fill:#bfb6fc
```

```
            +-----------+
            | __start__ |
            +-----------+
                  *
                  *
                  *
              +-------+
              | draft |
              +-------+
            ***        ...
           *              .
         **                ...
+---------------+              .
| execute_tools |           ...
+---------------+          .
            ***        ...
               *      .
                **   ..
            +---------+
            | revisor |
            +---------+
                  .
                  .
                  .
            +---------+
            | __end__ |
            +---------+
Retrying langchain_google_genai.chat_models._chat_with_retry.<locals>._chat_with_retry in 2.0 seconds as it raised ResourceExhausted: 429 You exceeded your current quota, please check your plan and billing details. For more information on this error, head to: https://ai.google.dev/gemini-api/docs/rate-limits. To monitor your current usage, head to: https://ai.dev/usage?tab=rate-limit.
* Quota exceeded for metric: generativelanguage.googleapis.com/generate_content_free_tier_requests, limit: 0, model: gemini-2.0-flash
* Quota exceeded for metric: generativelanguage.googleapis.com/generate_content_free_tier_requests, limit: 0, model: gemini-2.0-flash
* Quota exceeded for metric: generativelanguage.googleapis.com/generate_content_free_tier_input_token_count, limit: 0, model: gemini-2.0-flash
Please retry in 10.378343641s. [links {
  description: "Learn more about Gemini API quotas"
  url: "https://ai.google.dev/gemini-api/docs/rate-limits"
}
, violations {
}
violations {
}
violations {
}
```

```
, retry_delay {
  seconds: 10
}
].
Traceback (most recent call last):
  File "D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent\reflexion_graph.py", line 39, in <module>
    response = app.invoke("Write about how small business can leverage AI to grow")
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2683, in invoke
    for chunk in self.stream(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2331, in stream
    for _ in runner.tick(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\runner.py", line 146, in tick
    run_with_retry(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\retry.py", line 40, in run_with_retry
    return task.proc.invoke(task.input, config)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\utils\runnable.py", line 608, in invoke
    input = step.invoke(input, config)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\runnables\base.py", line 5711, in invoke
    return self.bound.invoke(
           ^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\language_models\chat_models.py", line 395, in invoke
    self.generate_prompt(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\language_models\chat_models.py", line 1025, in generate_prompt
    return self.generate(prompt_messages, stop=stop, callbacks=callbacks, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\language_models\chat_models.py", line 842, in generate
    self._generate_with_cache(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\language_models\chat_models.py", line 1091, in _generate_with_cache
    result = self._generate(
             ^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_google_genai\chat_models.py", line 951, in _generate
    response: GenerateContentResponse = _chat_with_retry(
                                        ^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_google_genai\chat_models.py", line 196, in _chat_with_retry
    return _chat_with_retry(**kwargs)
```

```
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 336, in
wrapped_f
    return copy(f, *args, **kw)
        ^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 475, in __call__
    do = self.iter(retry_state=retry_state)
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 376, in iter
    result = action(retry_state)
         ^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 418, in
exc_check
    raise retry_exc.reraise()
       ^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 185, in reraise
    raise self.last_attempt.result()
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\concurrent\futures\_base.py", line 449, in result
    return self.__get_result()
        ^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\concurrent\futures\_base.py", line 401, in __get_result
    raise self._exception
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 478, in __call__
    result = fn(*args, **kwargs)
         ^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_google_genai\chat_models.py", line 194, in _chat_with_retry
    raise e
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_google_genai\chat_models.py", line 178, in _chat_with_retry
    return generation_method(**kwargs)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\google\ai\generativelanguage_v1beta\services\generative_service\client.py", line 835, in
generate_content
    response = rpc(
          ^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\gapic_v1\method.py",
line 131, in __call__
    return wrapped_func(*args, **kwargs)
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\retry\retry_unary.py",
line 293, in retry_wrapped_func
    return retry_target(
        ^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\retry\retry_unary.py",
line 153, in retry_target
    _retry_error_helper(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\retry\retry_base.py",
line 212, in _retry_error_helper
```

```
    raise final_exc from source_exc
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\retry\retry_unary.py",
line 144, in retry_target
    result = target()
             ^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\timeout.py", line 130,
in func_with_timeout
    return func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\grpc_helpers.py", line
78, in error_remapped_callable
    raise exceptions.from_grpc_error(exc) from exc
google.api_core.exceptions.ResourceExhausted: 429 You exceeded your current quota, please
check your plan and billing details. For more information on this error, head to:
https://ai.google.dev/gemini-api/docs/rate-limits. To monitor your current usage, head to:
https://ai.dev/usage?tab=rate-limit.
* Quota exceeded for metric:
generativelanguage.googleapis.com/generate_content_free_tier_requests, limit: 0, model: gemini-
2.0-flash
* Quota exceeded for metric:
generativelanguage.googleapis.com/generate_content_free_tier_requests, limit: 0, model: gemini-
2.0-flash
* Quota exceeded for metric:
generativelanguage.googleapis.com/generate_content_free_tier_input_token_count, limit: 0,
model: gemini-2.0-flash
Please retry in 8.17108247s. [links {
  description: "Learn more about Gemini API quotas"
  url: "https://ai.google.dev/gemini-api/docs/rate-limits"
}
, violations {
}
violations {
}
violations {
}
, retry_delay {
  seconds: 8
}
]
During task with name 'draft' and id 'bfcfb231-01e4-b80b-d324-30a5449d5f67'
WARNING: All log messages before absl::InitializeLog() is called are written to
```

Nice, your graph itself is fine now — this error is 100% coming from **Gemini's API**, not from LangGraph or your code logic.

Key part:

```
ResourceExhausted: 429 You exceeded your current quota
limit: 0, model: gemini-2.0-flash
```

So Google is basically saying:

- For **`gemini-2.0-flash`**, your **current project has 0 free-tier requests allowed**, and you don't have a paid quota set up for it.
- So **every call** to that model fails with 429.

1️⃣ What this means in practice

Your reflection graph is doing this:

1. **`draft`** node → calls Gemini (structured output)
2. **`execute_tools`** → Tavily etc.
3. **`revisor`** → calls Gemini again (structured output)

But Gemini rejects even the **first** call because:

- Free tier for that model is **0**
- Or your project has no free quota / billing for that model

So LangGraph keeps retrying and eventually throws that long stack trace.

Your code is okay. This is an **account/quota/config** issue.

2️⃣ Options to fix / continue

You've got a few choices; you can pick what's easiest right now.

✅ Option A — Switch to OpenAI in this project

Since you're already using OpenAI in other stuff, easiest is: swap Gemini → OpenAI in this file and keep the rest the same.

In **`chains.py`**, replace:

```python
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(
    model="gemini-2.0-flash",
    temperature=0
)
```

with something like:

```python
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(
    model="gpt-4o-mini",    # or "gpt-4.1-mini" / "gpt-4o"
```

```
    temperature=0
)
```

Everything else can stay:

```
python

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)
```

As long as your **`OPENAI_API_KEY`** is in **`.env`**, this should just work.

✅ Option B — Use a different Gemini model / project with quota

If you really want to stay on Gemini:

1. Go to Google AI Studio and check:
   - Which project your API key belongs to
   - Whether **free tier** is enabled
   - Whether **`gemini-2.0-flash`** has any free quota at all for your account
2. Try a model that is more likely to have free usage, e.g.:

```
python

llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",  # or whatever currently has free tier for you
    temperature=0
)
```

3. Make sure your env var is correct for the version of the SDK you're using:
   - Most recent: **`GOOGLE_API_KEY`**
     (You already load **`.env`**, so just confirm the key name.)

If the error still shows **`limit: 0`**, that project simply has **no free quota** → you'd need to enable **billing** for more usage.

✅ Option C — For now, run the graph with a cheap/dummy LLM

If you just want to test the graph wiring and tool execution (not answer quality), you can temporarily:

- Use a smaller/cheaper OpenAI model (like **`gpt-4o-mini`**)
- Or even stub out the LLM with a fake one (for local graph debugging)

But easiest for you right now is **Option A**: switch to OpenAI since you're already comfortable there.

3️⃣ Quick sanity check after switching

Once you change to **`ChatOpenAI`**:

```python
response = app.invoke("Write about how small business can leverage AI to grow")
print(response)
```

You should see a **list of messages** as the final output. To print only the final revised answer, you can later do something like:

```python
final_msg = response[-1]       # last message in the list
print(final_msg.content)       # if it's an AIMessage
```

(We can clean that up once it's actually running.)

If you tell me which path you want:

- **Switch to OpenAI**, or
- **Stay on Gemini and debug quotas**

I can give you the exact code edits for that choice.

```
Retrying langchain_google_genai.chat_models._chat_with_retry.<locals>._chat_with_retry in 2.0 seconds as it raised NotFound: 404 models/gemini-1.5-flash is not found for API version v1beta, or is not supported for generateContent. Call ListModels to see the list of available models and their supported methods..
Traceback (most recent call last):
  File "D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent\reflexion_graph.py", line 39, in <module>
    response = app.invoke("Write about how small business can leverage AI to grow")
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2683, in invoke
    for chunk in self.stream(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2331, in stream
    for _ in runner.tick(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\runner.py", line 146, in tick
    run_with_retry(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\retry.py", line 40, in run_with_retry
    return task.proc.invoke(task.input, config)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\utils\runnable.py", line 608, in invoke
    input = step.invoke(input, config)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\runnables\base.py",
```

```
line 5711, in invoke
    return self.bound.invoke(
           ^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_core\language_models\chat_models.py", line 395, in invoke
    self.generate_prompt(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_core\language_models\chat_models.py", line 1025, in generate_prompt
    return self.generate(prompt_messages, stop=stop, callbacks=callbacks, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_core\language_models\chat_models.py", line 842, in generate
    self._generate_with_cache(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_core\language_models\chat_models.py", line 1091, in _generate_with_cache
    result = self._generate(
             ^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_google_genai\chat_models.py", line 951, in _generate
    response: GenerateContentResponse = _chat_with_retry(

  ^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_google_genai\chat_models.py", line 196, in _chat_with_retry
    return _chat_with_retry(**kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 336, in
wrapped_f
    return copy(f, *args, **kw)
           ^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 475, in __call__
    do = self.iter(retry_state=retry_state)
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 376, in iter
    result = action(retry_state)
             ^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 418, in
exc_check
    raise retry_exc.reraise()
          ^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 185, in reraise
    raise self.last_attempt.result()
          ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\concurrent\futures\_base.py", line 449, in result
    return self.__get_result()
           ^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\concurrent\futures\_base.py", line 401, in __get_result
    raise self._exception
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\tenacity\__init__.py", line 478, in __call__
    result = fn(*args, **kwargs)
             ^^^^^^^^^^^^^^^^^^^
```

```
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_google_genai\chat_models.py", line 194, in _chat_with_retry
    raise e
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\langchain_google_genai\chat_models.py", line 178, in _chat_with_retry
    return generation_method(**kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-
packages\google\ai\generativelanguage_v1beta\services\generative_service\client.py", line 835, in
generate_content
    response = rpc(
               ^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\gapic_v1\method.py",
line 131, in __call__
    return wrapped_func(*args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\retry\retry_unary.py",
line 293, in retry_wrapped_func
    return retry_target(
           ^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\retry\retry_unary.py",
line 153, in retry_target
    _retry_error_helper(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\retry\retry_base.py",
line 212, in _retry_error_helper
    raise final_exc from source_exc
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\retry\retry_unary.py",
line 144, in retry_target
    result = target()
             ^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\google\api_core\timeout.py", line 130,
in func_with_timeout
04 models/gemini-1.5-flash is not found for API version v1beta, or is not supported for
generateContent. Call ListModels to see the list of available models and their supported methods.
During task with name 'draft' and id '35231766-3ec6-7795-fe14-510e43843a52'
WARNING: All log messages before absl::InitializeLog() is called are written to STDERR
E0000 00:00:1765303439.365907   17056 init.cc:232] grpc_wait_for_shutdown_with_timeout() timed
out.

(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>
```

Nice, this is actually a **different** problem now (your graph is fine).

Now the error is:

```
NotFound: 404 models/gemini-1.5-flash is not found for API version v1beta, or is not
supported for generateContent
```

So:

- Previously: **429** → quota issue (no free usage for `gemini-2.0-flash`)
- Now: **404** → the **model name / API version combo is not supported** by the `langchain_google_genai` version you have.

In simple terms:
👉 Your current `langchain_google_genai` + Google SDK combo doesn't know about `gemini-1.5-flash` for `v1beta`.

You *can* fight this by upgrading packages and matching Google's new model naming…
but for what you're doing (LangGraph reflection agent), the **simplest and cleanest path right now is: switch this file back to OpenAI.**

You already know OpenAI, and the bootcamp examples are OpenAI-based anyway.

✅ Fix: Switch to OpenAI in this project

In your `chains.py`, change **just the LLM part**.

1. Replace the import

```python
# REMOVE this:
# from langchain_google_genai import ChatGoogleGenerativeAI

# ADD this:
from langchain_openai import ChatOpenAI
```

2. Replace the LLM definition

Right now you have:

```python
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash",
    temperature=0
)
```

Change it to:

```python
llm = ChatOpenAI(
    model="gpt-4o-mini",   # or "gpt-4o" if you want
    temperature=0
)
```

That's it. The rest stays the same:

```python
first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)

first_responder_chain = first_responder_prompt_template | first_responder_llm
revisor_chain = actor_prompt_template.partial(
    first_instruction=revise_instructions
) | revisor_llm
```

3. Make sure your `.env` has the OpenAI key

In your **`.env`**:

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

And you're already calling **`load_dotenv()`**, so that's fine.

🔍 Why this solves it

- **`ChatOpenAI`** → fully supported by your current LangChain version.
- No Google v1 / v1beta / model-name drama.
- **`with_structured_output()`** works the same way.
- LangGraph **`MessageGraph`** doesn't care which provider you use, as long as it's a **`ChatModel`**.

So your flow stays exactly:

```text
draft (first_responder_chain using OpenAI)
  → execute_tools (Tavily)
  → revisor (revisor_chain using OpenAI)
  → conditional loop
```

But now with **no Gemini API issues** in the middle.

If you *really* want to stick to Gemini later

Then you'd need to:

- **`pip install -U langchain-google-genai google-generativeai`**
- Use the right model ID for the API version they support (sometimes **`gemini-1.5-flash-latest`** or similar)
- Make sure you're on the newer **`v1`** stack, not old PaLM-era **`v1beta`**

But that's extra setup, and right now you just want your reflection graph **working**.

If you paste your updated **`chains.py`** after switching to **`ChatOpenAI`**, I can quickly sanity-check it for you before you run again.

i dont have open api key

Got it — **if you don't have an OpenAI API key**, then there are **two options left**:

✅ Option A — Use Gemini, but with a model that actually works with your SDK

Right now your errors tell us:

❌ `gemini-2.0-flash` → quota = 0 in your Google project

❌ `gemini-1.5-flash` → not supported in your API version (404)

So we need a **model name that BOTH:**

- exists for your SDK version (**`v1beta`**)
- AND is available in your project's free tier

This depends on your **exact Google API key**, but usually one of these works:

Try these model names:

```
gemini-1.5-flash-8b
gemini-1.5-flash
gemini-1.5-flash-latest
gemini-pro
gemini-pro-vision
```

(These are older but widely supported in older SDK versions.)

Replace your LLM call:

```python
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(
    model="gemini-pro",
    temperature=0,
)
```

or:

```python
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-flash-8b",
    temperature=0,
)
```

➡️ This is your **best chance** to run without quota issues.

❗ How to know EXACTLY which models you have access to

Run this small script:

```python
from google.generativeai import list_models
import google.generativeai as genai

genai.configure(api_key="YOUR_KEY")

models = list_models()
for m in models:
    print(m.name, m.supported_generation_methods)
```

This will show:

- the models your **API key supports**
- AND which have **`generateContent`** capability

Then pick any model where:

```nginx
supported_generation_methods includes "generateContent"
```

Example output:

```css
gemini-pro ['generateContent']
gemini-pro-vision ['generateContent']
gemini-1.5-flash-8b ['generateContent']
```

Pick one of these in **`ChatGoogleGenerativeAI`**.

✅ Option B — Use Groq (FREE & unlimited) instead of OpenAI

Groq gives **FREE API keys** for:

- Llama 3 70B
- Llama 3 8B
- Mixtral

- Gemma

And **structured output works** exactly the same as OpenAI.

You can get a free API key here:

👉 https://console.groq.com ↗

Then install:

```nginx
pip install langchain-groq
```

And change your code:

```python
from langchain_groq import ChatGroq

llm = ChatGroq(
    model="llama3-70b-8192",
    temperature=0
)

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)
```

That’s it.
Everything else stays the same — LangGraph does not care which LLM you use.

⭐ Recommendation for you

Since:

- Gemini free quota is unpredictable
- Model naming is API-version-dependent
- You have *no* OpenAI key right now

👉 **The best solution is: Switch to Groq.**
It’s free, fast, and supports structured output.

If you prefer staying with Gemini, we can debug the model list.

If you want, just tell me:

"I want to use Groq"
or
"I want to continue with Gemini"

And I will update your full **`chains.py`** for you.

Which one do you choose?

```
models/embedding-gecko-001 ['embedText', 'countTextTokens']
models/gemini-2.5-flash ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.5-pro ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.0-flash-exp ['generateContent', 'countTokens', 'bidiGenerateContent']
models/gemini-2.0-flash ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.0-flash-001 ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.0-flash-exp-image-generation ['generateContent', 'countTokens',
'bidiGenerateContent']
models/gemini-2.0-flash-lite-001 ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.0-flash-lite ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.0-flash-lite-preview-02-05 ['generateContent', 'countTokens',
'createCachedContent', 'batchGenerateContent']
models/gemini-2.0-flash-lite-preview ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-exp-1206 ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.5-flash-preview-tts ['countTokens', 'generateContent']
models/gemini-2.5-pro-preview-tts ['countTokens', 'generateContent']
models/gemma-3-1b-it ['generateContent', 'countTokens']
models/gemma-3-4b-it ['generateContent', 'countTokens']
models/gemma-3-12b-it ['generateContent', 'countTokens']
models/gemma-3-27b-it ['generateContent', 'countTokens']
models/gemma-3n-e4b-it ['generateContent', 'countTokens']
models/gemma-3n-e2b-it ['generateContent', 'countTokens']
models/gemini-flash-latest ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-flash-lite-latest ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-pro-latest ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.5-flash-lite ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-2.5-flash-image-preview ['generateContent', 'countTokens',
'batchGenerateContent']
models/gemini-2.5-flash-image ['generateContent', 'countTokens', 'batchGenerateContent']
models/gemini-2.5-flash-preview-09-2025 ['generateContent', 'countTokens',
'createCachedContent', 'batchGenerateContent']
models/gemini-2.5-flash-lite-preview-09-2025 ['generateContent', 'countTokens',
'createCachedContent', 'batchGenerateContent']
models/gemini-3-pro-preview ['generateContent', 'countTokens', 'createCachedContent',
'batchGenerateContent']
models/gemini-3-pro-image-preview ['generateContent', 'countTokens', 'batchGenerateContent']
models/nano-banana-pro-preview ['generateContent', 'countTokens', 'batchGenerateContent']
models/gemini-robotics-er-1.5-preview ['generateContent', 'countTokens']
```

```
models/gemini-2.5-computer-use-preview-10-2025 ['generateContent', 'countTokens']
models/embedding-001 ['embedContent']
models/text-embedding-004 ['embedContent']
models/gemini-embedding-exp-03-07 ['embedContent', 'countTextTokens', 'countTokens']
models/gemini-embedding-exp ['embedContent', 'countTextTokens', 'countTokens']
models/gemini-embedding-001 ['embedContent', 'countTextTokens', 'countTokens',
'asyncBatchEmbedContent']
models/aqa ['generateAnswer']
models/imagen-4.0-generate-preview-06-06 ['predict']
models/imagen-4.0-ultra-generate-preview-06-06 ['predict']
models/imagen-4.0-generate-001 ['predict']
models/imagen-4.0-ultra-generate-001 ['predict']
models/imagen-4.0-fast-generate-001 ['predict']
models/veo-2.0-generate-001 ['predictLongRunning']
models/veo-3.0-generate-001 ['predictLongRunning']
models/veo-3.0-fast-generate-001 ['predictLongRunning']
models/veo-3.1-generate-preview ['predictLongRunning']
models/veo-3.1-fast-generate-preview ['predictLongRunning']
models/gemini-2.0-flash-live-001 ['bidiGenerateContent', 'countTokens']
models/gemini-live-2.5-flash-preview ['bidiGenerateContent', 'countTokens']
models/gemini-2.5-flash-live-preview ['bidiGenerateContent', 'countTokens']
models/gemini-2.5-flash-native-audio-latest ['countTokens', 'bidiGenerateContent']
models/gemini-2.5-flash-native-audio-preview-09-2025 ['countTokens', 'bidiGenerateContent']
```

Nice, this list is super useful 👌
It shows exactly which **Gemini models your API key can see**.

Key thing: for **LangChain's `ChatGoogleGenerativeAI`**, you should use the model name **without** the `models/` prefix.

So for example:

- In the list: `models/gemini-2.5-flash`
- In code: `"gemini-2.5-flash"`

1️⃣ Which model name to use in your code

For your reflection agent, pick one of these **chat models**:

From your list:

- `models/gemini-2.5-flash`
- `models/gemini-2.5-pro`
- `models/gemini-flash-latest`
- `models/gemini-pro-latest`
- `models/gemma-3-4b-it` (smaller, cheaper, but should also work)

Example in **`chains.py`**:

```python
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",   # 👈 no "models/" prefix
    temperature=0,
)
```

Or:

```python
llm = ChatGoogleGenerativeAI(
    model="gemini-flash-latest",
    temperature=0,
)
```

Then keep:

```python
first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)
```

2️⃣ But remember your earlier error: 429 (quota exceeded)

You previously had:

> **`ResourceExhausted: 429 You exceeded your current quota... limit: 0, model: gemini-2.0-flash`**

That means for that Google Cloud project:

- The **free tier for `generateContent` is 0**
- And if you haven't enabled **billing**, **all** text model calls will fail like that.

So even if you use **`"gemini-2.5-flash"`** now, you might still hit:

- **`429 You exceeded your current quota`**

unless:

- you enable **billing** for that project in Google AI Studio / Cloud Console, or
- you have some free-trial credits left.

I can't see your billing status from here, but the pattern of the error strongly suggests:

> No active free quota + no paid quota → 429 on every generateContent.

3️⃣ TL;DR what you should do right now

1. **Update your code to use a valid model name**, e.g.:

```python
llm = ChatGoogleGenerativeAI(
    model="gemini-2.5-flash",
    temperature=0,
)
```

2. If you still get **429** after that:
   - Log in to your Google AI Console / AI Studio
   - Check that:
     - The project with this API key has **billing enabled**, or
     - It still has **free trial credits**

Without some quota (free or paid), the reflection graph will always fail at the first LLM call.

If you paste your current **`chains.py`** LLM part (just that small section), I can rewrite it with the exact working model string based on this list.

how to check load_dotenv api key

To **verify that `load_dotenv()` actually loaded your API key**, you can check it directly from Python using **`os.environ`**.

Here is the **correct and simplest method** 👇

✅ Step 1 — import os

```python
import os
```

✅ Step 2 — print the environment variable

For Gemini:

```python
print(os.environ.get("GOOGLE_API_KEY"))
```

For OpenAI (if you use later):

```python
print(os.environ.get("OPENAI_API_KEY"))
```

For other keys:

```python
print(os.environ.get("TAVILY_API_KEY"))
```

🔍 Expected output

If the key is loaded correctly:

```
AIzaSyA.........
```

If it is **not** loaded:

```css
None
```

or an empty string:

```bash
''
```

🚨 If you get None, it means:

1. Your `.env` file is **not in the same folder** as your Python file
2. `.env` is named wrong (must be `.env`, not `env.txt`, `.env.txt`, or `.env.example`)
3. The variable name inside `.env` does not match (must be exactly):

```ini
GOOGLE_API_KEY=AIzaSy.......
```

4. You called `load_dotenv()` **before** the key exists

🧪 Quick test script you can run

Create a small file `test_env.py`:

```python
from dotenv import load_dotenv
import os

load_dotenv()

print("GOOGLE_API_KEY =", os.environ.get("GOOGLE_API_KEY"))
print("TAVILY_API_KEY =", os.environ.get("TAVILY_API_KEY"))
```

Run:

```nginx
python test_env.py
```

🛑 If your .env file is NOT being found

You can force-load a specific `.env` path:

```python
from dotenv import load_dotenv
load_dotenv(dotenv_path="D:/New folder/Agentic-AI/.env")
```

🎁 Bonus: how to debug all loaded environment variables

```python
import os
print(os.environ)
```

(This prints everything—don't share it publicly.)

If you want, send me the structure of your folders and your `.env` content (with the key blurred), and I'll tell you exactly what is wrong.

cant we use groq for this

YES — Groq works *perfectly* for your Reflection Agent, and it is actually the **BEST** option for you because:

- ✔️ You **don't need an OpenAI key**
- ✔️ You **avoid all Gemini quota issues (429)**
- ✔️ Groq models are **FREE**, extremely **fast**, and fully support `with_structured_output(PydanticModel)`

- ✔️ LangGraph works exactly the same

So **you can run your entire reflection agent using Groq** with only a tiny change.

✅ 1. Install Groq Python SDK + LangChain integration

```nginx
pip install groq langchain-groq
```

✅ 2. Get a free API key

Go here → https://console.groq.com↗
Copy your key:

```ini
GROQ_API_KEY=xxxxx
```

Put it in your `.env`.

✅ 3. Replace ONLY the LLM section in your code

In your **`chains.py`**, replace the Gemini imports:

```python
from langchain_google_genai import ChatGoogleGenerativeAI
```

with:

```python
from langchain_groq import ChatGroq
```

Then replace the LLM:

```python
llm = ChatGroq(
    model="llama3-70b-8192",    # ⭐ BEST for accuracy
    temperature=0
)
```

You can also use:

- **`"llama3-8b-8192"`** (cheaper, still great)
- **`"mixtral-8x7b-32768"`** (very strong)

✅ 4. Keep the structured-output logic the same

```python
first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)
```

Groq fully supports structured Pydantic output.

✅ 5. Everything else stays the SAME

Your LangGraph:

- **`draft → execute_tools → revisor`**
- tool calls
- Tavily search
- event loop (**`MAX_ITERATIONS`**)
- MessageGraph nodes

All of it works with Groq exactly the same.

🔥 Here is your working updated LLM block

```python
from langchain_groq import ChatGroq

llm = ChatGroq(
    model="llama3-70b-8192",
    temperature=0
)

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)
```

That’s it.
Now run:

```python
response = app.invoke("Write about how small businesses can leverage AI to grow")
print(response)
```

It will work without quota errors.

⭐ Why Groq is the best choice for you right now

Provider	Issues
Gemini	Rate-limit issues, quota = 0, model naming inconsistent
OpenAI	You don't have a key
Groq	**Free**, **fast**, supports structured output, perfect for LangGraph

Your agent will run much faster and more reliably on Groq.

If you want, I can also:

✅ rewrite your entire `chains.py` with Groq
✅ rewrite `reflexion_graph.py` cleanly
✅ add pretty-print output for final answer

Just say **"update full code for Groq"**.

```
Traceback (most recent call last):
  File "D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent\reflexion_graph.py", line 43, in <module>
    response = app.invoke("Write about how small business can leverage AI to grow")
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2683, in invoke
    for chunk in self.stream(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2331, in stream
    for _ in runner.tick(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\runner.py", line 146, in tick
    run_with_retry(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\retry.py", line 40, in run_with_retry
    return task.proc.invoke(task.input, config)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\utils\runnable.py", line 608, in invoke
    input = step.invoke(input, config)
            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\runnables\base.py", line 5711, in invoke
    return self.bound.invoke(
           ^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\language_models\chat_models.py", line 395, in invoke
    self.generate_prompt(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\language_models\chat_models.py", line 1025, in generate_prompt
    return self.generate(prompt_messages, stop=stop, callbacks=callbacks, **kwargs)
```

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\language_models\chat_models.py", line 842, in generate
    self._generate_with_cache(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\language_models\chat_models.py", line 1091, in _generate_with_cache
    result = self._generate(
             ^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_groq\chat_models.py", line 503, in _generate
    response = self.client.create(messages=message_dicts, **params)
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\groq\resources\chat\completions.py", line 322, in create
    return self._post(
           ^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\groq\_base_client.py", line 1225, in post
    return cast(ResponseT, self.request(cast_to, opts, stream=stream, stream_cls=stream_cls))
                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\groq\_base_client.py", line 917, in request
    return self._request(
           ^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\groq\_base_client.py", line 1020, in _request
    raise self._make_status_error_from_response(err.response) from None
groq.BadRequestError: Error code: 400 - {'error': {'message': 'The model `llama3-70b-8192` has been decommissioned and is no longer supported. Please refer to https://console.groq.com/docs/deprecations for a recommendation on which model to use instead.', 'type': 'invalid_request_error', 'code': 'model_decommissioned'}}
During task with name 'draft' and id 'bc702c01-4e79-4ea5-d820-cf75331ba11e'

(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>
```

Nice, this error is actually super simple to fix — your graph + Groq setup is fine, only the **model name** is the problem.

> **`The model llama3-70b-8192 has been decommissioned and is no longer supported.`**

Groq removed `llama3-70b-8192` and wants you to use one of their **new model IDs** instead.

✅ 1. Change the Groq model name

In your `chains.py` (or wherever you created `llm = ChatGroq(...)`), you probably have:

```python
from langchain_groq import ChatGroq

llm = ChatGroq(
    model="llama3-70b-8192",
    temperature=0,
)
```

Change it to one of the currently supported ones, for example:

```python
llm = ChatGroq(
    model="llama-3.3-70b-versatile",  # good general model
    temperature=0,
)
```

Other valid Groq models include (names may look like, depending on current docs):

- **`"llama-3.1-8b-instant"`**
- **`"llama-3.1-70b-versatile"`**
- **`"llama-3.2-3b-preview"`**
- **`"llama-3.2-1b-preview"`**

But the key thing is: **use the new `llama-3.x-*-*` names, not `llama3-70b-8192`.**

So, your updated block should look like:

```python
from langchain_groq import ChatGroq

llm = ChatGroq(
    model="llama-3.3-70b-versatile",  # or another valid one
    temperature=0,
)

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)
```

Everything else in your reflection graph remains the same.

✅ 2. Why this error happens

Groq's API error:

```json
{
  "message": "The model `llama3-70b-8192` has been decommissioned and is no longer supported.",
  "code": "model_decommissioned"
}
```

Just means:

- The **client library is fine**
- Your **API key is valid**
- But that particular **model ID is old** and has been turned off.

Once you swap to a current ID, the 400 error will disappear.

✅ 3. After you fix the model name

Run again:

```bash
python reflexion_graph.py
```

And your flow:

- **`draft`** → **`execute_tools`** → **`revisor`**
  should now work with Groq, no Gemini quota issues, no OpenAI key needed.

If you paste your current **`chains.py`** (just the top part with imports + **`llm`**), I can rewrite it line-by-line with a safe Groq model so you can copy–paste directly.

aiohappyeyeballs==2.4.6
aiohttp==3.11.12
aiosignal==1.3.2
aiosqlite==0.21.0
annotated-types==0.7.0
anyio==4.8.0
appnope==0.1.4
asttokens==3.0.0
attrs==25.1.0
cachetools==5.5.1
certifi==2025.1.31
charset-normalizer==3.4.1
comm==0.2.2
dataclasses-json==0.6.7
debugpy==1.8.13
decorator==5.2.1
distro==1.9.0
executing==2.2.0
filetype==1.2.0
frozenlist==1.5.0
google-ai-generativelanguage==0.6.15
google-api-core==2.24.1
google-api-python-client==2.160.0
google-auth==2.38.0
google-auth-httplib2==0.2.0

```
google-generativeai==0.8.4
googleapis-common-protos==1.66.0
grandalf==0.8
groq==0.19.0
grpcio==1.70.0
grpcio-status==1.70.0
h11==0.14.0
httpcore==1.0.7
httplib2==0.22.0
httpx==0.28.1
httpx-sse==0.4.0
idna==3.10
ipykernel==6.29.5
ipython==9.0.2
ipython_pygments_lexers==1.1.1
jedi==0.19.2
jiter==0.8.2
jsonpatch==1.33
jsonpointer==3.0.0
jupyter_client==8.6.3
jupyter_core==5.7.2
langchain==0.3.18
langchain-community==0.3.17
langchain-core==0.3.45
langchain-google-genai==2.0.9
langchain-groq==0.2.5
langchain-openai==0.3.5
langchain-text-splitters==0.3.6
langgraph==0.3.21
langgraph-checkpoint==2.0.21
langgraph-checkpoint-sqlite==2.0.6
langgraph-prebuilt==0.1.7
langgraph-sdk==0.1.51
langsmith==0.3.7
marshmallow==3.26.1
matplotlib-inline==0.1.7
msgpack==1.1.0
multidict==6.1.0
mypy-extensions==1.0.0
nest-asyncio==1.6.0
numpy==1.26.4
openai==1.62.0
orjson==3.10.15
packaging==24.2
parso==0.8.4
pexpect==4.9.0
platformdirs==4.3.7
prompt_toolkit==3.0.50
propcache==0.2.1
proto-plus==1.26.0
protobuf==5.29.3
```

```
psutil==7.0.0
ptyprocess==0.7.0
pure_eval==0.2.3
pyasn1==0.6.1
pyasn1_modules==0.4.1
pydantic==2.10.6
pydantic-settings==2.7.1
pydantic_core==2.27.2
Pygments==2.19.1
pyparsing==3.2.1
python-dateutil==2.9.0.post0
python-dotenv==1.0.1
PyYAML==6.0.2
pyzmq==26.3.0
regex==2024.11.6
requests==2.32.3
requests-toolbelt==1.0.0
rsa==4.9
six==1.17.0
sniffio==1.3.1
SQLAlchemy==2.0.38
stack-data==0.6.3
tenacity==9.0.0
tiktoken==0.8.0
tornado==6.4.2
tqdm==4.67.1
traitlets==5.14.3
typing-inspect==0.9.0
typing_extensions==4.12.2
uritemplate==4.1.1
urllib3==2.3.0
wcwidth==0.2.13
xxhash==3.5.0
yarl==1.18.3
zstandard==0.23.0
#pip install langchain-tavily==0.2.7 tavily-python
these are the availbe my conda enviroment
```

Nice, thanks for pasting the `pip freeze` – your environment looks totally fine for what you're trying to do:

- You already have:
  - `langchain`, `langgraph`
  - `langchain-groq`, `groq`
  - `langchain-google-genai`
  - `python-dotenv`
- So we **don't** need to install anything else right now.

The only real issue is:

1. You don't have an OpenAI key (so we skip that).
2. Gemini is annoying with quotas.
3. Your Groq error is just a **wrong / old model name** (`llama3-70b-8192` has been removed).

Let's fix the Groq part properly and ignore Gemini/OpenAI for now.

✅ Step 1 – Use a *current* Groq model name

Groq deprecated `llama3-70b-8192`. Use one of their new Llama 3.x models, for example:

- `"llama-3.3-70b-versatile"` – strong, general
- `"llama-3.1-8b-instant"` – cheaper/faster
  (Names can vary slightly over time, but these are the current style.)

In your `chains.py`, find where you set up the LLM (you previously had Gemini / then Groq).

Change that block to:

```python
from langchain_groq import ChatGroq

llm = ChatGroq(
    model="llama-3.3-70b-versatile",   # 👈 updated model id
    temperature=0,
)
```

Then keep:

```python
first_responder_llm = llm.with_structured_output(AnswerQuestion)
revisor_llm = llm.with_structured_output(ReviseAnswer)
```

Nothing else in that file needs to change.

✅ Step 2 – Make sure your GROQ_API_KEY is loaded

Since you already have `python-dotenv`, just ensure your `.env` (in the same folder as your script, or higher up) contains:

```env
GROQ_API_KEY=sk-xxxxxxxxxxxxxxxx
```

and at the top of your `chains.py` you already have:

```python
from dotenv import load_dotenv
load_dotenv()
```

If you want to double-check it's loaded:

```python
import os
print("GROQ_API_KEY =", os.environ.get("GROQ_API_KEY"))
```

It should print something, not **None**.

✅ Step 3 – Sanity check: your graph file is fine

Your **reflexion_graph.py** doesn't need any model-specific edits. It just imports the chains:

```python
from chains import revisor_chain, first_responder_chain
```

and defines the graph:

- **draft** → **execute_tools** → **revisor**
- conditional loop

That part was already working (all the errors were coming from the LLM call, not LangGraph).

✅ Summary of what you should do now

1. In **chains.py**:
    - Use **ChatGroq** with a modern model ID:

    ```python
    from langchain_groq import ChatGroq

    llm = ChatGroq(
        model="llama-3.3-70b-versatile",
        temperature=0,
    )
    ```

2. Confirm **.env** has **GROQ_API_KEY** and **load_dotenv()** is called.
3. Run again:

```bash
(agent2) python reflexion_graph.py
```

If you want, paste your **full `chains.py`** here and I'll rewrite it cleanly with Groq + your existing prompt, so you can just copy–paste and run.

```
Traceback (most recent call last):
  File "D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent\reflexion_graph.py", line 43, in <module>
    response = app.invoke("Write about how small business can leverage AI to grow")
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2683, in invoke
    for chunk in self.stream(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2330, in stream
    while loop.tick(input_keys=self.input_channels):
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\loop.py", line 435, in tick
    mv_writes, updated_channels = apply_writes(
                                  ^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\algo.py", line 317, in apply_writes
    if channels[chan].update(vals) and get_next_version is not None:
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\channels\binop.py", line 89, in update
    self.value = self.operator(self.value, value)
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\graph\message.py", line 36, in _add_messages
    return func(left, right, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\graph\message.py", line 173, in add_messages
    for m in convert_to_messages(right)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\messa m in messages]
            ^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\messages\utils.py", line 360, in _convert_to_message
    raise NotImplementedError(msg)
NotImplementedError: Unsupported message type: <class 'schema.AnswerQuection'>
For troubleshooting, visit:
https://python.langchain.com/docs/troubleshooting/errors/MESSAGE_COERCION_FAILURE

(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>
```

Nice, this is **a different kind of error now**, and it's actually about **LangGraph + structured output**, not Groq or models.

Key part:

```
NotImplementedError: Unsupported message type: <class 'schema.AnswerQuection'>
```

This is saying:

- Your LangGraph **MessageGraph** expects each node to output **messages** (`BaseMessage` like `AIMessage`, `HumanMessage`, `ToolMessage`).
- But your `"draft"` node (first_responder_chain) is returning a **Pydantic object**: `AnswerQuection`.
- LangGraph tries to merge it into the message list and goes: "I don't know how to turn this object into a message → NotImplementedError".

So the graph is fine, Groq is fine, but **the node output type is wrong**.

Let's fix that cleanly.

1️⃣ What's happening now (mentally)

Right now your `chains.py` is roughly like:

```python
from langchain_groq import ChatGroq
from schema import AnswerQuection, ReviseAnswer

llm = ChatGroq(
    model="llama-3.3-70b-versatile",
    temperature=0,
)

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)

first_responder_chain = first_responder_prompt_template | first_responder_llm
revisor_chain = actor_prompt_template.partial(
    first_instruction=revise_instructions
) | revisor_llm
```

`with_structured_output(...)` returns a **Pydantic model**, e.g.:

```python
AnswerQuection(
    answer="...",
    search_queries=["..."],
    reflection=Reflection(...)
)
```

But `MessageGraph` expects each node to return:

- a single `BaseMessage` or
- a list of `BaseMessage` objects

So we need to **wrap the Pydantic object into an `AIMessage`** before it hits the graph.

2️⃣ Fix: wrap structured output → `AIMessage`

Open your **`chains.py`** and:

(a) Import `AIMessage` and `RunnableLambda`

```python
from langchain_core.messages import AIMessage
from langchain_core.runnables import RunnableLambda
```

(b) Add a helper function to convert `AnswerQuection` → `AIMessage`

```python
from schema import AnswerQuection, ReviseAnswer

def answer_to_message(ans: AnswerQuection) -> AIMessage:
    """
    Convert the structured AnswerQuection into an AIMessage
    so LangGraph can work with it.
    """
    return AIMessage(
        content=ans.answer,  # what the model "says"
        additional_kwargs={
            "search_queries": ans.search_queries,
            "reflection": ans.reflection.model_dump(),
        },
    )
```

> **`additional_kwargs`** is just extra data attached to the message — we'll use **`search_queries`** from here in **`execute_tools`**.

(c) Update your chains

Now change your chains to:

```python
llm = ChatGroq(
    model="llama-3.3-70b-versatile",
    temperature=0,
)

first_responder_llm = llm.with_structured_output(AnswerQuection)
revisor_llm = llm.with_structured_output(ReviseAnswer)

first_responder_chain = (
    first_responder_prompt_template
    | first_responder_llm
    | RunnableLambda(answer_to_message)
)

revisor_chain = (
    actor_prompt_template.partial(first_instruction=revise_instructions)
    | revisor_llm
```

```
    | RunnableLambda(answer_to_message)
)
```

Now:

- **"draft"** node returns an **AIMessage**
- **"revisor"** node also returns an **AIMessage**
- The Pydantic object never leaks into the graph → error disappears

---

3️⃣ Update `execute_tools` to use `search_queries` from the message

Previously your **execute_tools** was reading **tool_calls**.
Now we're not using **tool_calls** at all — we put **search_queries** into **additional_kwargs**.

So in **tool.py**, change **execute_tools** to something like this:

```python
import json
from typing import List, Dict, Any
from langchain_core.messages import AIMessage, BaseMessage, ToolMessage
from langchain_community.tools import TavilySearchResults

tavily_tool = TavilySearchResults(max_results=3)  # or your existing config

def execute_tools(state: List[BaseMessage]) -> List[BaseMessage]:
    # last message should be the AI draft / revised answer
    last_ai_message = state[-1]
    if not isinstance(last_ai_message, AIMessage):
        return []

    # get search_queries from additional_kwargs
    search_queries = (
        last_ai_message.additional_kwargs.get("search_queries", [])
        if hasattr(last_ai_message, "additional_kwargs")
        else []
    )

    if not search_queries:
        return []

    query_results: Dict[str, Any] = {}

    for query in search_queries:
        result = tavily_tool.invoke(query)
        query_results[query] = result

    # We don't really have a real tool_call_id now, but our graph
    # only uses ToolMessage type (for counting), so a dummy id is fine.
    tool_msg = ToolMessage(
        content=json.dumps(query_results),
        tool_call_id="search-1",
    )

    return [tool_msg]
```

Your **event_loop** still works, because it just does:

```python
count_tool_visit = sum(isinstance(item, ToolMessage) for item in state)
```

So we're still producing `ToolMessage` objects; we just aren't using OpenAI-style `tool_calls` anymore.

4️⃣ What the flow looks like now

1. `"draft"` node:
   - Uses prompt + Groq
   - Returns `AnswerQuection`
   - `RunnableLambda(answer_to_message)` wraps that into `AIMessage` with `answer` as content and `search_queries` in `additional_kwargs`.
2. `"execute_tools"` node:
   - Looks at last `AIMessage.additional_kwargs["search_queries"]`
   - Calls Tavily for each query
   - Returns a `ToolMessage` with JSON search results.
3. `"revisor"` node:
   - Sees the entire history: Human → AI (draft) → ToolMessage (results)
   - Uses prompt + Groq + structured ReviseAnswer
   - Again wrapped to `AIMessage` by the same `answer_to_message`.
4. `event_loop`:
   - If ToolMessages <= MAX_ITERATIONS → goes back to `"execute_tools"`
   - Else → `END`

No more `Unsupported message type: AnswerQuection` because now **every node returns messages**.

5️⃣ Recap of what you need to change

- In `chains.py`:
  - Add `answer_to_message(...)`
  - Pipe both chains through `RunnableLambda(answer_to_message)`
- In `tool.py`:
  - Make `execute_tools` read `search_queries` from `AIMessage.additional_kwargs`
  - Return a list of `ToolMessage` (with dummy `tool_call_id` OK)

Once this is done, your `app.invoke(...)` should run without that NotImplementedError, and the Groq error you saw earlier is already fixed by using the new model name.

If you paste your current `chains.py` and `tool.py`, I can rewrite them exactly with these changes so you can just copy–paste.

```
Traceback (most recent call last):
  File "D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent\reflexion_graph.py", line 38, in <module>
    response = app.invoke('Write about how small business can leverage AI to grow')
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2683, in invoke
    for chunk in self.stream(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2330, in stream
    while loop.tick(input_keys=self.input_channels):
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\loop.py", line 435, in tick
    mv_writes, updated_channels = apply_writes(
                                  ^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\algo.py", line 317, in apply_writes
    if channels[chan].update(vals) and get_next_version is not None:
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\channels\binop.py", line 89, in update
    self.value = self.operator(self.value, value)
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\graph\message.py", line 36, in _add_messages
    return func(left, right, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\graph\message.py", line 173, in add_messages
    for m in convert_to_messages(right)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\messages\utils.py", line 382, in convert_to_messages
    return [_convert_to_message(m) for m in messages]
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\messages\utils.py", line 382, in <listcomp>
    return [_convert_to_message(m) for m in messages]
            ^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langchain_core\messages\utils.py", line 360, in _convert_to_message
    raise NotImplementedError(msg)
NotImplementedError: Unsupported message type: <class 'schema.AnswerQuestion'>
For troubleshooting, visit: https://python.langchain.com/docs/troubleshooting/errors/MESSAGE_COERCION_FAILURE

(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>
```

Nice, this error is finally **exactly** what it looks like:

```
Unsupported message type: <class 'schema.AnswerQuestion'>
```

LangGraph is saying:

> "Your node returned a **Pydantic object (AnswerQuestion)**, but I only know how to work with **messages** (HumanMessage, AIMessage, ToolMessage, etc.)."

So your graph wiring is basically correct, but the **node output type** is wrong.

Right now you have something like:

```python
graph.add_node("draft", first_responder_chain)
graph.add_node("revisor", revisor_chain)
```

But:

- **`first_responder_chain`** returns an **`AnswerQuestion`** Pydantic object
- **`revisor_chain`** returns a **`ReviseAnswer`** Pydantic object

Those can't be dropped directly into a **`MessageGraph`**.

We need to wrap those Pydantic objects in an **AIMessage** before the graph sees them.

✅ Fix: wrap chain outputs into AIMessage in your graph

You don't need to touch your **`schema.py`** or most of **`chains.py`**.
Just change **`reflexion_graph.py`** like this.

1. Import `AIMessage` and your Pydantic models

At the top of **`reflexion_graph.py`** add:

```python
from langchain_core.messages import AIMessage, BaseMessage, ToolMessage
from schema import AnswerQuestion, ReviseAnswer
```

(You already import **`BaseMessage`** & **`ToolMessage`**, keep them.)

2. Wrap the chains in node functions

Add these functions **before** you build the graph:

```python
from chains import revisor_chain, first_responder_chain

def draft_node(state: list[BaseMessage]) -> list[BaseMessage]:
    """
```

```
    Node for 'draft':
    - Takes the message history (state)
    - Calls first_responder_chain, which returns AnswerQuestion (Pydantic)
    - Wraps it into an AIMessage so LangGraph is happy
    """
    result: AnswerQuestion = first_responder_chain.invoke({"messages": state})

    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "search_queries": result.search_queries,
            "reflection": result.reflection.model_dump() if hasattr(result, "reflection") else
None,
        },
    )
    return [msg]


def revisor_node(state: list[BaseMessage]) -> list[BaseMessage]:
    """
    Node for 'revisor':
    - Takes the message history (state)
    - Calls revisor_chain, which returns ReviseAnswer (Pydantic)
    - Wraps it into an AIMessage
    """
    result: ReviseAnswer = revisor_chain.invoke({"messages": state})

    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "search_queries": result.search_queries,
            "reflection": result.reflection.model_dump() if hasattr(result, "reflection") else
None,
            "references": result.references,
        },
    )
    return [msg]
```

We're now **hiding** the Pydantic object inside **`AIMessage.additional_kwargs`**.
The graph only sees **`AIMessage`**, which is valid.

3. Register these functions as nodes instead of the raw chains

Change your node registrations from:

```python
graph = MessageGraph()
graph.add_node("draft", first_responder_chain)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_chain)
```

to:

```python
graph = MessageGraph()
graph.add_node("draft", draft_node)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_node)
```

That's the key fix.
Now every node returns a list of **messages**, not Pydantic models.

---

4. What about `execute_tools`?

Right now, your **`execute_tools(state)`** probably looks at **`last_ai_message.tool_calls`**.
Since we're no longer using tools / tool_calls with Groq, that condition will just fail, and it will return **`[]`**.
That's fine for now (no error, just no searches).

If you want it to actually use the **`search_queries`** from the Pydantic output, you can later change it to something like:

```python
def execute_tools(state: list[BaseMessage]) -> list[BaseMessage]:
    last_ai = state[-1]
    if not isinstance(last_ai, AIMessage):
        return []

    search_queries = last_ai.additional_kwargs.get("search_queries", [])
    if not search_queries:
        return []

    query_results = {}
    for q in search_queries:
        result = tavily_tool.invoke(q)
        query_results[q] = result

    return [
        ToolMessage(
            content=json.dumps(query_results),
            tool_call_id="search-1",
        )
    ]
```

But that's **optional** for fixing your current error.

🔁 After making these changes

Run:

```bash
(agent2) python reflexion_graph.py
```

Now:

- **`"draft"`** returns **`AIMessage`**
- **`"execute_tools"`** returns **`ToolMessage`** (or **`[]`**)
- **`"revisor"`** returns **`AIMessage`**

So LangGraph **will not complain** about **`schema.AnswerQuestion`** anymore.

If you paste your current **`reflexion_graph.py`**, I can rewrite it fully with this change so you can just copy–paste.

```
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
import datetime
from langchain_groq import ChatGroq
from schema import AnswerQuestion, ReviseAnswer
#from langchain_core.messages import HumanMessage

from dotenv import load_dotenv

load_dotenv()


actor_prompt_template = ChatPromptTemplate.from_messages([
    (
        "system",
        """
You are expert AI researcher.
Current time: {time}

You must follow this process:

1. {first_instruction}
2. Reflect and critique your answer. Be severe to maximize improvement.
3. After the reflection, list 1â€“3 search queries separately for researching improvements.
   - Do NOT include the search queries inside the reflection.
4. Finally, answer the user's question above using the required format.
        """.strip()
    ),
    MessagesPlaceholder(variable_name="messages"),
])
# auto-fill time
actor_prompt_template = actor_prompt_template.partial(
    time=lambda: datetime.datetime.now().isoformat()
)



first_responder_prompt_template = actor_prompt_template.partial(first_instruction="provide a
detailed ~250 word answer")


llm = ChatGroq(
    model="llama-3.3-70b-versatile",  # good general model
    temperature=0,
)


first_responder_llm = llm.with_structured_output(AnswerQuestion)
revisor_llm = llm.with_structured_output(ReviseAnswer)

# Revisor instructions
```

revise_instructions = """Revise your previous answer using the new information.
- You should use the previous critique to add important information to your answer.
- You MUST include numerical citations in your revised answer to ensure it can be verified.
- Add a "References" section to the bottom of your answer (which does not count towards the word limit). In form of:
- [1] https://example.com
- [2] https://example.com
- You should use the previous critique to remove superfluous information from your answer and make SURE it is not more than 250 words.
"""

first_responder_chain = first_responder_prompt_template | first_responder_llm
revisor_chain = actor_prompt_template.partial(first_instruction=revise_instructions) | revisor_llm
is this support with gro

q

Short answer: **yes**, this code is fine with Groq ✅
But there are a couple of small details you should be aware of.

1️⃣ Your `chains.py` with Groq – is it valid?

You have:

```python
from langchain_groq import ChatGroq
from schema import AnswerQuestion, ReviseAnswer
from dotenv import load_dotenv

load_dotenv()

actor_prompt_template = ChatPromptTemplate.from_messages([
    (
        "system",
        """
You are expert AI researcher.
Current time: {time}

You must follow this process:

1. {first_instruction}
2. Reflect and critique your answer. Be severe to maximize improvement.
3. After the reflection, list 1â€“3 search queries separately for researching improvements.
   - Do NOT include the search queries inside the reflection.
4. Finally, answer the user's question above using the required format.
        """.strip()
    ),
    MessagesPlaceholder(variable_name="messages"),
])
actor_prompt_template = actor_prompt_template.partial(
    time=lambda: datetime.datetime.now().isoformat()
)

first_responder_prompt_template = actor_prompt_template.partial(
    first_instruction="provide a detailed ~250 word answer"
)

llm = ChatGroq(
```

```
    model="llama-3.3-70b-versatile",
    temperature=0,
)

first_responder_llm = llm.with_structured_output(AnswerQuestion)
revisor_llm = llm.with_structured_output(ReviseAnswer)

revise_instructions = """Revise your previous answer using the new information.
    - You should use the previous critique to add important information to your answer.
        - You MUST include numerical citations in your revised answer to ensure it can be
verified.
        - Add a "References" section to the bottom of your answer (which does not count towards
the word limit). In form of:
            - [1] https://example.com
            - [2] https://example.com
    - You should use the previous critique to remove superfluous information from your answer
and make SURE it is not more than 250 words.
"""

first_responder_chain = first_responder_prompt_template | first_responder_llm
revisor_chain = actor_prompt_template.partial(
    first_instruction=revise_instructions
) | revisor_llm
```

✅ This **does work with Groq**, as long as:

1. **`GROQ_API_KEY`** is set in **`.env`**
2. **`AnswerQuestion`** and **`ReviseAnswer`** are **Pydantic BaseModel** classes
3. You use these chains correctly in LangGraph (this is where your last error came from)

Groq supports **`with_structured_output(PydanticModel)`** in **`ChatGroq`**, so that part is fine.

2️⃣ Tiny things to clean up (optional but good)

Weird `1â€“3` in the prompt

That’s just a UTF-8 / editor encoding glitch for the en-dash.
You can replace it with a normal hyphen:

```python
3. After the reflection, list 1-3 search queries separately for researching improvements.
```

Not required, just prettier.

3️⃣ Important: this alone is not enough for LangGraph

What **doesn’t** work is plugging **`first_responder_chain`** directly into **`MessageGraph`**, because it returns a **Pydantic object**, not a message.

So:

```python
graph.add_node("draft", first_responder_chain)  # ❌ causes Unsupported message type:
AnswerQuestion
```

You must **wrap the structured output into an `AIMessage`** in your graph code (in `reflexion_graph.py`), like I explained earlier with:

```python
def draft_node(state: list[BaseMessage]) -> list[BaseMessage]:
    result: AnswerQuestion = first_responder_chain.invoke({"messages": state})
    return [
        AIMessage(
            content=result.answer,
            additional_kwargs={
                "search_queries": result.search_queries,
                "reflection": result.reflection.model_dump(),
            },
        )
    ]
```

and then:

```python
graph.add_node("draft", draft_node)
```

So:

- ✅ Your **Groq + chains code is correct**
- ❌ But **Groq doesn't magically fix the MessageGraph error** — you still need the wrapper.

If you send me your current `reflexion_graph.py`, I can plug this `chains.py` into it and give you a **fully working** version you can just run.

↳

2\. Wrap the chains in node functions Add these functions **before** you build the graph: python Copy code `from chains import revisor_chain, first_responder_chain def draft_node(state: list[BaseMessage]) -> list[BaseMessage]: """ Node for 'draft': - Takes the message history (state) - Calls first_responder_chain, which returns AnswerQuestion (Pydantic) - Wraps it into an AIMessage so LangGraph is happy """ result: AnswerQuestion = first_responder_chain.invoke({"messages": state}) msg = AIMessage(content=result.answer, additional_kwargs={ "search_queries": result.search_queries, "reflection": result.reflection.model_dump() if hasattr(result, "reflection") else None, },) return [msg] def revisor_node(state: list[BaseMessage]) -> list[BaseMessage]: """ Node for 'revisor': - Takes the message history (state) - Calls revisor_chain, which returns ReviseAnswer (Pydantic) - Wraps it into an AIMessage """ result: ReviseAnswer = revisor_chain.invoke({"messages": state}) msg = AIMessage(content=result.answer, additional_kwargs={ "search_queries": result.search_queries, "reflection": result.reflection.model_dump() if hasattr(result, "reflection") else None, "references": result.references, },) return [msg]` We're now **hiding** the Pydantic object inside `AIMessage.additional_kwargs`. The graph only sees `AIMessage`, which is valid.
can you explain why this happend

Yeah, let's unpack *why* we had to do that weird-looking `draft_node` / `revisor_node` wrapping thing 😄

1️⃣ What LangGraph expects vs what your chain returns

Your graph is a **MessageGraph**. That means:

- The **state** is:

```python
List[BaseMessage]  # [HumanMessage, AIMessage, ToolMessage, ...]
```

- Every **node** in the graph must:
  - **take** a `List[BaseMessage]` as input (the conversation so far)
  - **return** one or more `BaseMessage` objects (usually another `AIMessage` or `ToolMessage`)

So from LangGraph's point of view, a node is like:

```python
def node(state: List[BaseMessage]) -> List[BaseMessage]:
    ...
    return [AIMessage(...)]  # or [ToolMessage(...)]
```

2️⃣ What your `first_responder_chain` actually returns

You created this:

```python
first_responder_llm = llm.with_structured_output(AnswerQuestion)
first_responder_chain = first_responder_prompt_template | first_responder_llm
```

So when you call:

```python
result = first_responder_chain.invoke({"messages": state})
```

it returns a **Pydantic object**:

```python
AnswerQuestion(
    answer="...",
    search_queries=["..."],
    reflection=Reflection(...)
)
```

That is **not** a `BaseMessage`. It's just a normal Python object.

When you did this earlier:

```python
graph.add_node("draft", first_responder_chain)
```

LangGraph tried to treat that chain like a node and got back `AnswerQuestion`. Then it tried to merge it into the message state and crashed with:

> `Unsupported message type: <class 'schema.AnswerQuestion'>`

Because it only knows how to handle things like `HumanMessage`, `AIMessage`, etc., **not arbitrary Pydantic models**.

3️⃣ What the wrapper functions are doing

So we wrote:

```python
def draft_node(state: list[BaseMessage]) -> list[BaseMessage]:
    result: AnswerQuestion = first_responder_chain.invoke({"messages": state})

    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "search_queries": result.search_queries,
            "reflection": result.reflection.model_dump() if hasattr(result, "reflection") else
None,
        },
    )
    return [msg]
```

This does 3 important things:

1. **Takes `state` (list of messages)** → this matches what LangGraph passes into a node.
2. **Calls your structured chain** → gets the `AnswerQuestion` Pydantic object.
3. **Wraps it into an `AIMessage`** → which *is* a valid `BaseMessage`.

So now LangGraph sees:

- Before: `AnswerQuestion(...)` ❌
- After: `AIMessage(content="...", additional_kwargs={...})` ✅

Same with `revisor_node` and `ReviseAnswer`.

We're basically saying:

> "Hey LangGraph, don't worry about this Pydantic stuff.
> Here's a normal chat message you understand, and I'll secretly keep the extra fields (`search_queries`, `reflection`, `references`) in `additional_kwargs`."

4️⃣ Why hide things in `additional_kwargs`?

Because later, in **`execute_tools`**, you want to read the **`search_queries`** that the model produced.

Instead of relying on **`tool_calls`** (OpenAI-style), we do:

```python
last_ai_message.additional_kwargs["search_queries"]
```

So the data flow is:

- LLM → structured **`AnswerQuestion`**
- **`draft_node`** → converts to **`AIMessage`** and stores extra fields in **`additional_kwargs`**
- **`execute_tools`** → reads those fields and runs Tavily
- LangGraph → is happy because everything is still just messages

---

5️⃣ So *why* did this happen?

Because you mixed two worlds:

1. **Structured-output LLM world** → returns Pydantic models
2. **MessageGraph world** → wants **`BaseMessage`** objects

LangGraph can't automatically guess how to turn **`AnswerQuestion`** into a message, so you have to be explicit and do the "adapter" step yourself.

Those wrapper functions (**`draft_node`**, **`revisor_node`**) are exactly that adapter layer.

If you want, next I can:

- Draw you a little diagram: "state before draft → after draft → after tools → after revisor"
- Or we can refactor your code so both nodes share one common "pydantic_to_message" helper instead of duplicating logic.

```
(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>python reflexion_graph.py
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
Traceback (most recent call last):
  File "D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent\reflexion_graph.py", line 81, in <module>
    response = app.invoke('Write about how small business can leverage AI to grow')
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2683, in invoke
    for chunk in self.stream(
  File "C:\Users\Asus\.conda\envs\agent2\Lib\site-packages\langgraph\pregel\__init__.py", line 2351, in stream
    raise GraphRecursionError(msg)
```

langgraph.errors.GraphRecursionError: Recursion limit of 25 reached without hitting a stop condition. You can increase the limit by setting the `recursion_limit` config key.
For troubleshooting, visit:
https://python.langchain.com/docs/troubleshooting/errors/GRAPH_RECURSION_LIMIT

(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>

Ahhh nice, **new kind of error = new kind of win** 🎯
This one is about your **graph looping forever**, not about keys or models.

Error:

```
GraphRecursionError: Recursion limit of 25 reached without hitting a stop condition.
```

That basically means:

> "Your graph keeps bouncing between nodes again and again, and it never reaches `END`, so I stopped after 25 steps."

Let's unpack **why** that's happening in *your* graph.

1️⃣ How your graph currently flows

From earlier, your graph is something like:

```python
graph.add_node("draft", draft_node)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_node)

graph.add_edge("draft", "execute_tools")
graph.add_edge("execute_tools", "revisor")

graph.add_conditional_edges("revisor", event_loop)

graph.set_entry_point("draft")
```

So the flow is:

1. Start → `draft`
2. `draft` → `execute_tools`
3. `execute_tools` → `revisor`
4. From `revisor` → **decided dynamically by `event_loop`**

Your `event_loop` earlier looked like this:

```python
MAX_ITERATIONS = 2
```

```
def event_loop(state: List[BaseMessage]) -> str:
    count_tool_visit = sum(isinstance(item, ToolMessage) for item in state)
    num_of_iterations = count_tool_visit
    if num_of_iterations > MAX_ITERATIONS:
        return END
    return "execute_tools"
```

So:

- It counts how many `ToolMessage`s are in the state.
- If `> MAX_ITERATIONS` → go to `END`
- Else → go back to `"execute_tools"`

2️⃣ Why you hit the recursion limit

There are **two common reasons** this blows up into an infinite(ish) loop:

🧨 Case A — `execute_tools` never returns a ToolMessage

If your current `execute_tools` function returns `[]` (no `ToolMessage`s), then:

- `state` never contains any `ToolMessage`
- So: `count_tool_visit` is **always 0**
- So: `num_of_iterations` is always 0
- So: `num_of_iterations > MAX_ITERATIONS` is **never true**
- So: `event_loop` **always** returns `"execute_tools"`

That gives you a loop like:

```text
draft → execute_tools → revisor → execute_tools → revisor → execute_tools → ...
```

LangGraph says: "Okay, 25 steps done, still no END, I'm out." → `GraphRecursionError`.

This is very likely what's happening *right now*, especially since we changed how tools are handled.

🧨 Case B — `event_loop` logic never returns END for your state

Even if you *do* create `ToolMessage`s, if `MAX_ITERATIONS` is big or your logic is wrong, you might *never* reach `> MAX_ITERATIONS`.

Example:
If `MAX_ITERATIONS = 2` and for some reason only 1 `ToolMessage` ever gets added to state (like you reset state or it doesn't grow as you think), then again:

- `num_of_iterations` will be ≤ 2 forever
- It will keep returning `"execute_tools"`
- Infinite loop again.

3️⃣ Easiest ways to fix it

You have a few options, depending on what you want.

✅ Option 1 — Don't loop at all (simplest)

If you're okay with:

- 1 draft
- 1 tools call
- 1 revision

then just **remove the loop** and make **`revisor`** go straight to **`END`**.

Replace:

```python
graph.add_conditional_edges("revisor", event_loop)
```

with:

```python
graph.add_edge("revisor", END)
```

Now the flow is:

```text
draft → execute_tools → revisor → END
```

No recursion, no loop, no headache.
Still a valid Reflexion-type agent, just with one refinement.

✅ Option 2 — Keep loop, but count iterations reliably

If you **really want multiple refinement loops**, then we need a safer condition than "count ToolMessages".

The hacky-but-simple way in a **`MessageGraph`** is:

- Count how many times **`revisor`** has run = how many **`AIMessage`**s with some flag exist.

Example:

In **`revisor_node`**, you could tag messages with **`iteration`**:

```python
def revisor_node(state: list[BaseMessage]) -> list[BaseMessage]:
    # count previous revisions
    past_revisions = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
        for m in state
    )

    result: ReviseAnswer = revisor_chain.invoke({"messages": state})

    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "role": "revisor",
            "iteration": past_revisions + 1,
            "search_queries": result.search_queries,
            "reflection": result.reflection.model_dump() if hasattr(result, "reflection") else
None,
            "references": result.references,
        },
    )
    return [msg]
```

Then in `event_loop`:

```python
def event_loop(state: List[BaseMessage]) -> str:
    num_iterations = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
        for m in state
    )
    if num_iterations >= MAX_ITERATIONS:
        return END
    return "execute_tools"
```

Now the loop is controlled by how many times `revisor` has actually run.

But this is **more advanced**. If you're still wiring this up, I'd recommend Option 1 first.

4️⃣ TL;DR — Why this happened to *you*

- LangGraph keeps stepping through the graph until:
  - it hits `END`, **or**
  - it hits `recursion_limit` (default 25)
- Your graph:
  - Never reached `END`
  - Because `event_loop` always returned `"execute_tools"`
  - Because `num_of_iterations` never went over `MAX_ITERATIONS`
  - Most likely since no `ToolMessage` was ever added to state
    (so `count_tool_visit` was always 0)

So LangGraph said:

> "25 hops, still no END → I assume you made an infinite loop → `GraphRecursionError`."

If you show me your **current `execute_tools`** and **current `event_loop`**, I can tell you exactly which of the two cases you hit and rewrite them into something clean and stable.

option 1 worked

Yesss 🎉 nice, that means your whole Groq + LangGraph + Reflexion setup is now actually running end-to-end.

So right now your flow is:

```text
Human question
  → draft (first_responder_chain, structured AnswerQuestion → AIMessage)
  → execute_tools (Tavily or whatever you wired)
  → revisor (ReviseAnswer → AIMessage)
  → END
```

No looping → no recursion limit → clean run ✅

What changed with Option 1

Before:

- `revisor` had **conditional edges**:

  ```python
  graph.add_conditional_edges("revisor", event_loop)
  ```

- `event_loop` often returned `"execute_tools"` again
  → graph bounced: `execute_tools → revisor → execute_tools → ...`
  → never reached `END` → `GraphRecursionError`.

Now:

- You replaced that with:

  ```python
  graph.add_edge("revisor", END)
  ```

- So after one revision, the graph **always stops**.

This is still a perfectly fine “Reflexion-style” setup:
1 draft, 1 round of tools, 1 revision.

Next useful thing: see the *final* answer cleanly

Right now you're doing:

```python
response = app.invoke("Write about how small business can leverage AI to grow")
print(response)
```

`response` is probably a **list of messages** (history). To just see the final revised answer:

```python
final_msg = response[-1]          # last message in the state
print(type(final_msg))
print(final_msg.content)          # the revised answer text
```

If you stored references in **`additional_kwargs`**, you can also inspect:

```python
print(final_msg.additional_kwargs.get("references"))
```

If you want, you can paste the current **`reflexion_graph.py`** and I'll add a tiny "pretty print final answer" section at the end so it prints something like:

```text
=== FINAL ANSWER ===
...

=== REFERENCES ===
[1] ...
[2] ...
```

but i want loop

Nice, that means your graph is healthy now 🙌
Let's bring the loop back *safely* so it doesn't go infinite and hit the recursion limit again.

Why the loop broke before

Your old setup was:

```text
draft → execute_tools → revisor → (event_loop decides) → execute_tools or END
```

But your **event_loop** logic was basically:

```python
def event_loop(state):
    # count ToolMessages
    count_tool_visit = sum(isinstance(m, ToolMessage) for m in state)
    if count_tool_visit > MAX_ITERATIONS:
        return END
    return "execute_tools"
```

If **no `ToolMessage` ever gets added** (e.g. `execute_tools` returns `[]` because there are no `search_queries`), then:

- `count_tool_visit` is always 0
- `0 > MAX_ITERATIONS` is never true
- So it *always* returns `"execute_tools"`
- → infinite loop → `GraphRecursionError`.

So we need a **better condition** for stopping.

Plan: loop a fixed number of times using `revisor` iterations

Instead of counting `ToolMessage`s, we'll count:

- How many times `revisor` has run.

Step 1 – Tag revisor messages with a role & iteration

In your `revisor_node` (the wrapper we added in `reflexion_graph.py`), change it like this:

```python
from langchain_core.messages import AIMessage, BaseMessage
from schema import ReviseAnswer

MAX_ITERATIONS = 2  # or whatever you like

def revisor_node(state: list[BaseMessage]) -> list[BaseMessage]:
    # Count how many previous revisor outputs we already have
    past_revisions = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
        for m in state
    )

    result: ReviseAnswer = revisor_chain.invoke({"messages": state})

    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "role": "revisor",
            "iteration": past_revisions + 1,
            "search_queries": result.search_queries,
            "reflection": (
                result.reflection.model_dump()
                if hasattr(result, "reflection")
                else None
            ),
            "references": result.references,
```

```
        },
    )
    return [msg]
```

So now every time **`revisor`** runs, it adds an **`AIMessage`** like:

```python
AIMessage(
    content="revised answer...",
    additional_kwargs={
        "role": "revisor",
        "iteration": 1 or 2 or ...,
        "search_queries": [...],
        "reflection": {...},
        "references": [...]
    }
)
```

Step 2 – Write a smarter `event_loop`

Now we use those tags to decide whether to continue or stop:

```python
from langgraph.graph import END
from langchain_core.messages import AIMessage, BaseMessage
from typing import List

def event_loop(state: List[BaseMessage]) -> str:
    # How many times has revisor run?
    num_iterations = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
        for m in state
    )

    # 1) Stop if we've hit the max number of revisions
    if num_iterations >= MAX_ITERATIONS:
        return END

    # 2) Also stop if there are no search_queries to run tools with
    last_ai = next((m for m in reversed(state) if isinstance(m, AIMessage)), None)
    if not last_ai:
        return END

    search_queries = last_ai.additional_kwargs.get("search_queries", [])
    if not search_queries:
        # Nothing more to search → no point looping
        return END

    # Otherwise, go to tools again
    return "execute_tools"
```

So the loop ends when:

- You've revised **`MAX_ITERATIONS`** times, **or**
- There are no more **`search_queries`** from the last AI step.

Step 3 – Wire the loop back in the graph

Right now you have:

```python
graph.add_edge("revisor", END)  # <- you added this for Option 1
```

Replace that with:

```python
graph.add_conditional_edges("revisor", event_loop)
```

Full node wiring should look like:

```python
graph.add_node("draft", draft_node)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_node)

graph.add_edge("draft", "execute_tools")
graph.add_edge("execute_tools", "revisor")

graph.add_conditional_edges("revisor", event_loop)

graph.set_entry_point("draft")
```

Now the flow is:

```text
draft
  → execute_tools
  → revisor (iteration 1)
  → event_loop → execute_tools
  → revisor (iteration 2)
  → event_loop → END   (because num_iterations >= MAX_ITERATIONS)
```

Or it may stop earlier if there are no `search_queries`.

Recap

- Option 1 worked because we forced `revisor → END`.
- When you added the loop back, it broke because the stop condition was based on `ToolMessage`s, which didn't behave as you expected.
- We fixed it by:
  - tagging `revisor` outputs with `role="revisor"` and `iteration=n`
  - counting those in `event_loop`
  - stopping after `MAX_ITERATIONS` revisions or if there are no queries.

If you paste your current `revisor_node`, `event_loop`, and graph wiring, I can inline-edit them into a final version you can just drop in.

(agent2) D:\New folder\Agentic-AI\Langgraph-Bootcamp\04_Reflexion Agent>python reflexion_graph.py
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
[HumanMessage(content='Write about how small business can leverage AI to grow', additional_kwargs={}, response_metadata={}, id='8877f86c-e4ea-4cf2-9618-aabe9a0d3e19'), AIMessage(content="Small businesses can leverage AI to grow in various ways. Firstly, AI-powered chatbots can be used to provide 24/7 customer support, helping to improve customer engagement and reduce the workload of human customer support agents. Additionally, AI-driven analytics tools can help small businesses gain valuable insights into their customers' behavior, preferences, and needs, enabling them to make data-driven decisions and create targeted marketing campaigns. AI can also be used to automate routine tasks such as bookkeeping, invoicing, and data entry, freeing up time for more strategic and creative work. Furthermore, AI-powered tools can help small businesses to improve their supply chain management, predict demand, and optimize their pricing strategies. Moreover, AI can be used to enhance the customer experience through personalized recommendations, offers, and content. To get started with AI, small businesses can explore cloud-based AI services, such as Google Cloud AI Platform, Microsoft Azure Machine Learning, or Amazon SageMaker, which provide pre-built AI models and tools that can be easily integrated into their existing systems. By embracing AI, small businesses can gain a competitive edge, increase efficiency, and drive growth.", additional_kwargs={'search_queries': ['small business AI adoption success stories', 'AI for small business marketing', 'challenges of implementing AI in small businesses'], 'reflection': {'missing': 'The answer could benefit from more specific examples of small businesses that have successfully leveraged AI to grow, as well as more detailed information on the potential challenges and limitations of implementing AI in small businesses.', 'superfluous': 'The answer could be more concise and focused on the most important points, avoiding unnecessary details and examples.'}}, response_metadata={}, id='7388d72d-123e-4f10-af8d-48c538c92517'), AIMessage(content='Small businesses can leverage AI to grow in various ways. According to a study, 61% of businesses have seen an increase in productivity after implementing AI [1]. AI-powered chatbots can provide 24/7 customer support, improving customer engagement and reducing workload. AI-driven analytics tools can help small businesses gain insights into customer behavior, enabling data-driven decisions and targeted marketing campaigns. AI can automate routine tasks, freeing up time for strategic work. AI-powered tools can also improve supply chain management, predict demand, and optimize pricing strategies. To get started, small businesses can explore cloud-based AI services like Google Cloud AI Platform or Amazon SageMaker. By embracing AI, small businesses can gain a competitive edge, increase efficiency, and drive growth. References: [1] https://www.example.com/ai-productivity', additional_kwargs={'role': 'revisor', 'iteration': 1, 'search_queries': ['AI for small businesses', 'cloud-based AI services', 'AI-driven analytics tools'], 'reflection': {'missing': 'Previous answer lacked numerical citations and specific examples', 'superfluous': 'Previous answer included unnecessary information about AI-powered tools'}, 'references': ['https://www.example.com/ai-productivity']}, response_metadata={}, id='a5926722-354f-47ab-a962-5b5c93611bbb')]
without loop how output look like

Here is **exactly how your output will look WITHOUT the loop**, step by step, so you understand the difference clearly.

Right now your printed output (WITH loop) looks like this:

```ini
[
  HumanMessage(...),
  AIMessage(draft answer ...),
  AIMessage(revised answer ...)
]
```

That is **3 messages**:

1. Human input
2. First responder draft
3. Revisor improved answer (iteration 1)

✅ What output looks like WITHOUT LOOP

If you remove the loop and connect:

```python
graph.add_edge("revisor", END)
```

Then the flow becomes:

```powershell
draft → execute_tools → revisor → END
```

So the output will ALWAYS be:

Final state list (3 messages):

```arduino
[
  HumanMessage(
      content="Write about how small business can leverage AI to grow"
  ),

  AIMessage(
      content="Draft answer (~250 words)...",
      additional_kwargs={
          "search_queries": [...],
          "reflection": {...}
      }
  ),

  AIMessage(
      content="Revised answer with citations...",
      additional_kwargs={
          "role": "revisor",
          "iteration": 1,
          "search_queries": [...],
          "reflection": {...},
          "references": [...]
      }
```

```
  )
]
```

✔️ **Only ONE revision**

✔️ **No second revision**

✔️ **No repeated tool calls**

✔️ **Graph stops immediately after revisor**

🔍 Let me show you EXACTLY how it would print in your terminal

If you run:

```python
response = app.invoke("Write about how small business can leverage AI to grow")
print(response)
```

You will see something like this:

OUTPUT WITHOUT LOOP

```bash
[
HumanMessage(content='Write about how small business can leverage AI to grow'),

AIMessage(
  content="Small businesses can leverage AI by...",
  additional_kwargs={
    'search_queries': [...],
    'reflection': { 'missing': "...", 'superfluous': "..." }
  }
),

AIMessage(
  content="Revised final answer with citations...",
  additional_kwargs={
    'role': 'revisor',
    'iteration': 1,
    'search_queries': [...],
    'reflection': {...},
    'references': [...]
  }
)
]
```

🟩 Most important difference (NO LOOP)

With loop:

```powershell
draft → execute_tools → revisor → execute_tools → revisor → END
```

Final state has **5 messages**:

- human
- draft
- execute_tools output
- revisor #1
- revisor #2

Without loop:

```powershell
draft → execute_tools → revisor → END
```

Final state has **3 messages**:

- human
- draft
- final revision

🎯 If you want, I can generate a "pretty-print" version like:

```python-repl
=== DRAFT ===
...
=== REVISION ===
...
=== REFERENCES ===
...
```

Just tell me — I can add that at the bottom of your script.

when we change to egde its work no loop but when we change it to conditional then is not work

Yeah, that behaviour is exactly what we'd expect from how **`add_edge`** vs **`add_conditional_edges`** work in LangGraph – you're not doing anything "wrong", it's just that your **stop condition is never actually reached** when you switch back to conditional routing.

Let's break it down in plain language.

---

1. What happens with `graph.add_edge("revisor", END)`

This is the simple case:

```python
graph.add_edge("revisor", END)
```

Flow is:

```text
draft → execute_tools → revisor → END
```

There's **no decision**, no loop, nothing dynamic:

- `revisor` runs once
- Graph immediately goes to `END`
- So it **always terminates** → no recursion error

That's why "it works" when you use `add_edge`.

---

2. What changes with `add_conditional_edges("revisor", event_loop)`

Now you tell LangGraph:

```python
graph.add_conditional_edges("revisor", event_loop)
```

This means:

> "After `revisor`, call `event_loop(state)` and let **that function decide** which node to go to next."

So **your function** controls everything now.

If your `event_loop` is something like:

```python
def event_loop(state: List[BaseMessage]) -> str:
    count_tool_visit = sum(isinstance(m, ToolMessage) for m in state)
    if count_tool_visit > MAX_ITERATIONS:
        return END
    return "execute_tools"
```

Then:

- If `count_tool_visit` never goes above `MAX_ITERATIONS`
  → this function **never returns** `END`
  → it always returns `"execute_tools"`
  → graph loops:

```text
draft → execute_tools → revisor → execute_tools → revisor → execute_tools → ...
```

LangGraph sees 25 hops, still no `END` →
➡️ `GraphRecursionError: Recursion limit of 25 reached`.

So when you say:

> “when we change to edge its work no loop but when we change it to conditional then is not work”

it’s because:

- `add_edge` ignores `event_loop` and just stops ✅
- `add_conditional_edges` uses `event_loop`, and right now `event_loop` **never hits** `END` ❌

---

3. How to make the loop actually stop (and work)

You need **both**:

1. `revisor_node` tagging its outputs
2. `event_loop` reading those tags and deciding when to stop

Here’s a **working pair** you can drop in.

✅ `revisor_node` (in `reflexion_graph.py`)

```python
from langchain_core.messages import AIMessage, BaseMessage
from schema import ReviseAnswer

MAX_ITERATIONS = 2  # how many times you want to revise

def revisor_node(state: list[BaseMessage]) -> list[BaseMessage]:
    # Count how many previous revisor outputs we already have
    past_revisions = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
        for m in state
    )

    result: ReviseAnswer = revisor_chain.invoke({"messages": state})

    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "role": "revisor",
            "iteration": past_revisions + 1,
            "search_queries": result.search_queries,
            "reflection": (
                result.reflection.model_dump()
                if hasattr(result, "reflection")
                else None
            ),
            "references": result.references,
        },
```

```
    )
    return [msg]
```

This makes sure every revision message is marked like:

```python
AIMessage(additional_kwargs={"role": "revisor", "iteration": 1})
```

Then:

✅ `event_loop` (also in `reflexion_graph.py`)

```python
from typing import List
from langgraph.graph import END
from langchain_core.messages import AIMessage, BaseMessage

def event_loop(state: List[BaseMessage]) -> str:
    # How many revisor outputs so far?
    num_iterations = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
        for m in state
    )

    # 1) Stop if we've hit the max number of revisions
    if num_iterations >= MAX_ITERATIONS:
        return END

    # 2) Optional safety: if there are no search_queries, stop
    last_ai = next((m for m in reversed(state) if isinstance(m, AIMessage)), None)
    if not last_ai:
        return END

    search_queries = last_ai.additional_kwargs.get("search_queries", [])
    if not search_queries:
        return END

    # Otherwise, go to tools again
    return "execute_tools"
```

And then wire it:

```python
graph.add_node("draft", draft_node)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_node)

graph.add_edge("draft", "execute_tools")
graph.add_edge("execute_tools", "revisor")

graph.add_conditional_edges("revisor", event_loop)

graph.set_entry_point("draft")
```

Now the flow becomes:

```text
draft
  → execute_tools
```

```
  → revisor (iteration 1)
  → event_loop → execute_tools
  → revisor (iteration 2)
  → event_loop → END       (because num_iterations >= MAX_ITERATIONS)
```

So:

- ✅ You **get a loop** (`tools ↔ revisor`)
- ✅ It **stops after N iterations**
- ✅ No recursion error

---

4. TL;DR

- `add_edge("revisor", END)` = always END, no loop → always works.
- `add_conditional_edges("revisor", event_loop)` = **your function** must eventually return `END`.
- Right now, when you switch to conditional, your `event_loop` condition never becomes true → infinite loop → recursion error.
- Fix = tag revisor outputs, then count them in `event_loop` and stop after `MAX_ITERATIONS`.

If you want, you can paste your current `revisor_node` and `event_loop`, and I'll patch them into a final working version using your exact code style.

now its working , i have q quection why my old not worked and this one work and also i want to is it because my old code caompatible with only openAi or what

Nice, glad it's running now 💪
Let's answer both parts:

1️⃣ Why your *old* code didn't work, and this one does

There were actually **two separate problems** in the old version:

Problem A — Returning the *wrong type* into `MessageGraph`

- **What the LangGraph `MessageGraph` expects**:
  Every node must return one or more **messages**:
  - `HumanMessage`, `AIMessage`, `ToolMessage`, etc. (all subclasses of `BaseMessage`)
- **What your old node returned**:
  You used:

```python
first_responder_llm = llm.with_structured_output(AnswerQuestion)
first_responder_chain = first_responder_prompt_template | first_responder_llm
```

So `first_responder_chain.invoke(...)` returned a **Pydantic model**:

```python
AnswerQuestion(
    answer="...",
    search_queries=[...],
    reflection=Reflection(...)
)
```

Then you did:

```python
graph.add_node("draft", first_responder_chain)
```

So the graph node output was `AnswerQuestion`, not `AIMessage`.

- **Result**:
  LangGraph tried to merge that into the message stream and threw:

  ```
  Unsupported message type: <class 'schema.AnswerQuestion'>
  ```

✅ **Fix we did**: we wrapped the structured output into an `AIMessage`:

```python
def draft_node(state: list[BaseMessage]) -> list[BaseMessage]:
    result: AnswerQuestion = first_responder_chain.invoke({"messages": state})

    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "search_queries": result.search_queries,
            "reflection": result.reflection.model_dump(),
        },
    )
    return [msg]
```

Now the node returns a **list of messages**, which `MessageGraph` understands.

Problem B — Loop condition never becoming true → infinite loop

Your old loop logic was something like:

```python
def event_loop(state: List[BaseMessage]) -> str:
    count_tool_visit = sum(isinstance(m, ToolMessage) for m in state)
    if count_tool_visit > MAX_ITERATIONS:
        return END
    return "execute_tools"
```

This assumes:

- `execute_tools` **produces** `ToolMessage`**s each time**,
- so `count_tool_visit` grows (1, 2, 3, ...),
- and eventually `> MAX_ITERATIONS` becomes true → returns `END`.

But in your modified setup:

- `execute_tools` either returned `[]` or not what you expected.
- So **no** `ToolMessage`**s were ever added**.
- So `count_tool_visit` was always `0`.
- So `0 > MAX_ITERATIONS` was **never true**.
- So `event_loop` always returned `"execute_tools"`.

That gave you this infinite-ish cycle:

```text
draft → execute_tools → revisor → execute_tools → revisor → execute_tools → ...
```

LangGraph hit the recursion limit (25 steps) and threw:

`GraphRecursionError: Recursion limit of 25 reached without hitting a stop condition.`

✅ **Fix we did**:

- Tag `revisor` outputs with `role="revisor"` and `iteration=n`.
- In `event_loop`, count how many revisor messages there are:

  ```python
  num_iterations = sum(
      isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
      for m in state
  )
  if num_iterations >= MAX_ITERATIONS:
      return END
  ```

So now the stop condition is based on **your own tags**, not on `ToolMessage` count.

2️⃣ Is this because the old code only worked with OpenAI?

Not *exactly*, but you've hit on something real:

- The **original tutorial code** you followed was likely built for **OpenAI-style tools** using `ChatOpenAI.bind_tools(...)`.
- In that pattern:
  - The LLM returns `AIMessage` with `.tool_calls`.

- `execute_tools` reads `.tool_calls`, runs tools, and emits `ToolMessages`.
- The example loop logic ("count `ToolMessages`") matches that behavior.

When you changed things, you changed the **protocol shape**:

- You switched to **Gemini** / **Groq** and used `with_structured_output(PydanticModel)` instead of `.bind_tools`.
- Now:
  - The LLM returns a **Pydantic object**, not an `AIMessage` with tool calls.
  - We changed `execute_tools` to read `search_queries` from `additional_kwargs`, not from `tool_calls`.
  - That means the **original OpenAI-specific assumptions** ("there will be ToolMessages", "there will be tool_calls") were no longer true.

So:

- ❌ It's **not** that your old code "only works with OpenAI" in a magical way
- ✅ It's that your **graph logic was tightly coupled to OpenAI's tool-calling behavior**:
  - tool_calls
  - ToolMessage count
- When you moved to Groq + structured output, you changed the LLM behavior but *didn't adjust the graph logic* at first → boom, errors.

Once we:

1. Converted structured outputs → `AIMessage`
2. Switched the loop condition from `ToolMessage` count → our own `revisor` tags

...your graph became **provider-agnostic**. It now works the same with:

- Groq (`ChatGroq`)
- OpenAI (`ChatOpenAI`)
- Gemini (`ChatGoogleGenerativeAI`)

as long as you keep the same:

- message types
- additional_kwargs structure
- event loop logic.

If you want, next I can:

- Show you a tiny comparison: **"OpenAI tools version vs Groq structured-output version"** side-by-side, so you can see what changed exactly.

that mean old one work with open ai right

✅ YES — your old code *would have worked* with OpenAI.

And here is the exact reason why:

Your original design assumed **OpenAI-style tool calling**, which looks like this:

- The model returns an **`AIMessage`** containing:

```json
{
  "tool_calls": [
      {
        "name": "AnswerQuestion",
        "args": { ... },
        "id": "call_123"
      }
  ]
}
```

Then:

- **`execute_tools`** reads **`last_ai_message.tool_calls`**
- The tool produces a **`ToolMessage`**
- LangGraph merges the **`ToolMessage`**
- Your **loop stop condition** counted **`ToolMessage`**s

This EXACT pattern is how OpenAI works.

🔍 Why your old code DID NOT work with Groq/Gemini

Groq and Gemini **do not produce `tool_calls`** when using:

```python
llm.with_structured_output(PydanticModel)
```

Instead, they return **only the Pydantic object**, e.g.:

```python
AnswerQuestion(...)
```

So:

🔥 Problem 1 — The graph received a Pydantic model, not an `AIMessage`

LangGraph expects:

```python
AIMessage, HumanMessage, ToolMessage
```

but you gave it:

```python
AnswerQuestion (Pydantic object)
```

OpenAI would have returned an `AIMessage` with `tool_calls`, so the old code would WORK there.

🔥 Problem 2 — Your event loop depended on ToolMessages

Your old loop logic was:

```python
count_tool_visit = sum(isinstance(m, ToolMessage) for m in state)
if count_tool_visit > MAX_ITERATIONS:
    return END
```

With OpenAI:

- The LLM generates `tool_calls`
- `execute_tools` generates `ToolMessage`
- `ToolMessage`s accumulate → loop stops correctly

With Gemini or Groq:

- No `tool_calls`
- `execute_tools` returns nothing
- No `ToolMessage`s added
- `count_tool_visit` remains 0 forever
- `event_loop` never returns END
- → Infinite loop → recursion error

🟢 Summary: Why it works with OpenAI but not Groq/Gemini

Feature	OpenAI	Groq/Gemini
Returns tool_calls	✔️ YES	❌ No
Produces ToolMessage → loop counter works	✔️ YES	❌ No
Structured output returns an AIMessage	✔️ YES	❌ No (returns Pydantic object)
Old loop logic dependent on ToolMessage	✔️ Works	❌ Breaks

So **your OLD design was fundamentally tied to OpenAI's tool-calling mechanism.**

🟦 The NEW design we built = Provider agnostic

What we did:

1️⃣ Converted structured output → `AIMessage`

```python
AIMessage(content=result.answer, additional_kwargs={...})
```

2️⃣ Created our own iteration tracking:

```python
additional_kwargs["role"] = "revisor"
additional_kwargs["iteration"] = n
```

3️⃣ Stopped loop based on our own labels instead of ToolMessages.

✔️ Works with OpenAI

✔️ Works with Groq

✔️ Works with Gemini

✔️ Works without tools

✔️ Always predictable

🟢 Final Answer

✔️ YES — your old code would have worked with OpenAI only

❌ It would not work with Groq or Gemini

✔️ The new code works with ALL LLM providers

If you want, I can show you a **side-by-side comparison of old vs new** so you understand exactly which parts depended on OpenAI.

i need a clear explanation about this never get wrong again this kind of errors

Love this question, because once you really "get" this, a HUGE class of weird LangGraph / LangChain errors will stop happening to you.

I'll break it into **two big ideas**:

1. **What shape of data each layer expects**
2. **Why your old logic was secretly OpenAI-specific**

Then I'll give you a **mental checklist** you can reuse so you don't fall into this again.

---

1. The three layers you were mixing

You had three things stacked:

1. **LLM output pattern**
    - OpenAI tools (`bind_tools`, `tool_calls`)
    - vs structured output (`with_structured_output(PydanticModel)`)
2. **LangChain message types**
    - `HumanMessage`, `AIMessage`, `ToolMessage`, etc.
3. **LangGraph `MessageGraph`**
    - Every node: `List[BaseMessage] -> List[BaseMessage]`
    - Graph loops/termination controlled by your function (like `event_loop`)

If *any one* of these doesn't match what the others expect, you get errors.

---

2. Pattern A – OpenAI tools style (what your old code assumed)

When you use **OpenAI with `.bind_tools(...)`**, the flow is:

Step 1: LLM output

`ChatOpenAI(...).bind_tools([...])` returns an `AIMessage` like:

```python
AIMessage(
    content="",
    tool_calls=[
        {
            "name": "AnswerQuestion",
            "args": {...},
            "id": "call_123",
        }
    ]
)
```

Key thing: **it's already an `AIMessage`**.
LangGraph is happy with this type.

Step 2: `execute_tools` sees `tool_calls`

Your typical tools node code:

```python
last_ai_message = state[-1]

for tool_call in last_ai_message.tool_calls:
    # run tool, get result
    tool_messages.append(
        ToolMessage(
            content=json.dumps(result),
            tool_call_id=tool_call["id"]
        )
    )

return tool_messages
```

So now the state grows like:

```text
HumanMessage
AIMessage (with tool_calls)
ToolMessage (tool result)
```

Step 3: Loop logic based on `ToolMessage`

Your old **`event_loop`**:

```python
def event_loop(state):
    count_tool_visit = sum(isinstance(m, ToolMessage) for m in state)
    if count_tool_visit > MAX_ITERATIONS:
        return END
    return "execute_tools"
```

This works **ONLY IF**:

- Tools are called,
- **`ToolMessage`**s are actually produced,
- The count goes 1, 2, 3, … and eventually **`> MAX_ITERATIONS`**.

This entire story is built around:

- **`AIMessage.tool_calls`**
- **`ToolMessage`** being created by **`execute_tools`**

👉 **This is essentially the “OpenAI tools pattern”.**

---

3. Pattern B – Structured output with Pydantic (what you switched to)

Then you changed your LLM usage to:

```python
llm = ChatGroq(...)
first_responder_llm = llm.with_structured_output(AnswerQuestion)
```

Now, calling:

```python
result = first_responder_llm.invoke(...)
```

returns:

```python
AnswerQuestion(
    answer="...",
    search_queries=[...],
    reflection=Reflection(...)
)
```

Important differences:

1. This is **not an `AIMessage`**. It's a **Pydantic model**.
2. There are **no `tool_calls`**.
3. If you plug this directly into `MessageGraph`, you break it.

You then did:

```python
graph.add_node("draft", first_responder_chain)
```

So the node output type was `AnswerQuestion`, not `AIMessage`.

LangGraph tried to treat it as a message and hit:

> `Unsupported message type: <class 'schema.AnswerQuestion'>`

Because `MessageGraph` **contract is**:

- Node input: `List[BaseMessage]`
- Node output: `BaseMessage` or `List[BaseMessage]`

You gave it something else.

---

4. Why you got the two different errors

Error 1 – `Unsupported message type: <class 'schema.AnswerQuestion'>`

Cause: node output type mismatch.

- **`MessageGraph`** expects messages (like **`AIMessage`**, **`ToolMessage`**).
- You returned **`AnswerQuestion`** (Pydantic).
- This mismatch has **nothing to do with OpenAI vs Groq** directly.
- It's purely: *"Is the thing you returned a LangChain message object or not?"*

✅ Fix: wrap the Pydantic model yourself:

```python
def draft_node(state):
    result: AnswerQuestion = first_responder_chain.invoke({"messages": state})
    return [
        AIMessage(
            content=result.answer,
            additional_kwargs={
                "search_queries": result.search_queries,
                "reflection": result.reflection.model_dump(),
            },
        )
    ]
```

Now the graph sees only messages and it's happy.

Error 2 – `GraphRecursionError: Recursion limit of 25`

Cause: your loop condition never returned `END`.

Your old **`event_loop`** logic:

```python
def event_loop(state):
    count_tool_visit = sum(isinstance(m, ToolMessage) for m in state)
    if count_tool_visit > MAX_ITERATIONS:
        return END
    return "execute_tools"
```

But in the new world:

- You weren't using OpenAI tool_calls.
- You probably didn't produce **`ToolMessage`**s (or not as you thought).
- So **`count_tool_visit`** = 0 always.
- So **`count_tool_visit > MAX_ITERATIONS`** is never true.
- So it **always** returned **`"execute_tools"`**.

That creates a loop:

```text
draft → execute_tools → revisor → execute_tools → revisor → execute_tools → ...
```

LangGraph runs 25 steps, sees no **`END`**, and stops with:

```
GraphRecursionError: Recursion limit of 25 reached without hitting a stop condition.
```

✅ Fix: we changed the **termination condition** to something that always eventually becomes true, by **tagging our own messages**:

```python
# In revisor_node:
msg = AIMessage(
    content=result.answer,
    additional_kwargs={
        "role": "revisor",
        "iteration": past_revisions + 1,
        ...
    },
)
```

then:

```python
def event_loop(state):
    num_iterations = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
        for m in state
    )
    if num_iterations >= MAX_ITERATIONS:
        return END
    return "execute_tools"
```

Now the loop is:

- After 1st `revisor` → iteration = 1
- After 2nd `revisor` → iteration = 2
- `num_iterations >= MAX_ITERATIONS` → returns `END` → stops.

This logic is provider-agnostic. It doesn't care if the LLM is OpenAI, Groq, or Gemini. It only cares about the messages you control.

---

5. So, was it "OpenAI-only"?

Kind of:

- Your **graph's logic** (tools + loop) was **designed for OpenAI's tool calling**:
  - `tool_calls` on `AIMessage`
  - `ToolMessage`s always being produced
  - Stopping based on number of `ToolMessage`s
- When you switched to **Groq + structured output**, you changed:
  - The shape of the LLM output (Pydantic instead of `AIMessage`),
  - How tools are triggered (no `tool_calls`),
  - But **you kept the same loop/termination logic**, which depended on ToolMessages.

So:

- ✅ With OpenAI + `bind_tools`, your original code **would likely work**.
- ❌ With Groq/Gemini + `with_structured_output`, the assumptions break and errors appear.
- ✅ The new version we built is **not tied to OpenAI**. It's based on:
  - You converting model output → `AIMessage`
  - You controlling extra fields via `additional_kwargs`
  - You defining loop conditions based on those fields

---

6. "Never get this wrong again" – Mental checklist

When you wire LLMs + LangGraph next time, run this checklist:

✅ Step 1 – For every node in `MessageGraph`:

- Input type: `List[BaseMessage]`?
- Output type: `BaseMessage` or `List[BaseMessage]`?
- If you're using `with_structured_output`, **wrap it** into an `AIMessage` manually.

  Rule: *No raw Pydantic models or dicts should ever go directly into the graph state.*

✅ Step 2 – If you use tools:

- Are you using **OpenAI-style tools** (`bind_tools`)?
  - Then you will have `AIMessage.tool_calls` → `ToolMessage`.
- If you're using structured output **instead of** tools:
  - Don't write logic that assumes `tool_calls` or `ToolMessage`.

  Rule: *Your control logic (loops, conditions) must match your tool mechanism.*

✅ Step 3 – For loops (`add_conditional_edges`):

Always ask:

1. **What condition will eventually make this function return `END`?**
2. Does that condition depend on something that really changes each iteration?
   (e.g. count of messages with a tag)
3. Can I debug it by printing the state and checking that the counter increases?

   Rule: *A loop that never hits its stop condition will always end in `GraphRecursionError`.*

If you want, I can take your **final working code** and annotate it with comments explaining each design choice (like: "why this is a message", "why this is provider-agnostic", "why this stop condition is safe").

That way you have a “template” to copy from in future projects.

how do i learn return output structure each llm

Great question — THIS is the core skill you need to build ANY agentic workflow reliably.
Once you understand **how each LLM returns output**, you will NEVER again be confused about:

- tool_calls
- structured output
- message formats
- content vs additional fields
- why graphs break
- why chaining fails

So let me teach you a **repeatable method** for learning *any* LLM’s output structure.

✅ THE GOLDEN RULE

> **Never assume what an LLM returns.**
> **Always PRINT the raw output and inspect its structure.**

Every model (OpenAI, Gemini, Groq, Anthropic, etc.) has subtle differences.

🧪 1. The universal method: use `.invoke()` and print the object

Here’s the single most powerful debugging trick:

```python
resp = llm.invoke("hello world")
print(resp)
print(type(resp))
print(resp.__dict__)
```

This will show you:

- whether it's an **`AIMessage`**
- what fields exist (**`content`**, **`additional_kwargs`**, **`tool_calls`**, **`response_metadata`**)
- whether structured output is applied
- any hidden fields

🧪 2. Test each pattern separately

Pattern A — Normal LLM output

```python
resp = llm.invoke("Say hi")
print(resp)
```

OpenAI returns:

```python
AIMessage(
  content="Hi!",
  additional_kwargs={},
)
```

Gemini returns:

```python
AIMessage(
  content="Hi!",
  additional_kwargs={"is_gemini":"yes"},  # depends on library version
)
```

Groq returns:

```python
AIMessage(
  content="Hi!",
  additional_kwargs={}
)
```

These differences are minor — you'll learn to ignore them.

Pattern B — OpenAI tool calling output

```python
llm = ChatOpenAI(...).bind_tools([MyTool])
resp = llm.invoke("run tool")
print(resp)
```

You will see:

```python
AIMessage(
  content="",
  tool_calls=[
    {
      "name": "MyTool",
      "args": { "x": 10 },
```

```
        "id": "call_123"
      }
    ]
  )
```

This ONLY happens with OpenAI **or** providers that copy the OpenAI tools spec.

⚠️ Groq and Gemini DO NOT support this yet.

Pattern C — Structured output (Pydantic)

For Groq, Gemini, or OpenAI:

```python
resp = llm.with_structured_output(MySchema).invoke("give data")
print(resp)
print(type(resp))
```

You will see:

```python
MySchema(
   field1="...",
   field2=123
)
```

→ NOT an AIMessage
→ NOT tool_calls
→ Just a **Pydantic model**

This is where many people get confused.

If you're using LangGraph MessageGraph:

❌ You cannot return this directly
✔️ You must wrap it:

```python
AIMessage(content=str(resp), additional_kwargs=resp.model_dump())
```

🧪 3. Learn the output by viewing JSON format

Every LangChain LLM has:

```python
resp = llm.invoke("test")
print(resp.model_dump())
```

Or:

```python
import json
print(json.dumps(resp.dict(), indent=2))
```

This shows you EXACTLY what the model returned.

🧪 4. Inspect tool calls after structured output

If using structured output:

```python
llm = ChatGroq(...).with_structured_output(MySchema)

resp = llm.invoke("test")
print(resp)
```

You'll see a pure Python object.

If using tool calling:

```python
print(resp.tool_calls)
```

You'll see a list of tool call definitions.

You MUST NOT mix these two patterns.

🧪 5. Visit the model documentation pages

Each provider documents output formats:

OpenAI

https://platform.openai.com/docs ↗

Google Gemini

https://ai.google.dev/docs ↗

Groq

https://console.groq.com/docs ↗

But honestly:
just printing `.invoke()` results is usually clearer than reading docs.

🧪 6. Cheatsheet: What each provider returns

Provider	Normal output	Tool calls	Structured output	Notes
OpenAI	AIMessage	✔️ Full support	✔️ Works	Most mature tool calling
Groq (LLaMA)	AIMessage	❌ No tool_calls	✔️ Pydantic structured	Their LLaMA models don't do tool calling
Gemini	AIMessage	❌ No tool_calls	✔️ Pydantic structured	Needs manual wrapping for LangGraph
Anthropic Claude	AIMessage	❌ No traditional tool_calls	✔️ Function calling like Pydantic	Different function signature

Knowing this lets you avoid incorrect assumptions.

💡 So WHY did your old graph break?

Because your old graph assumed:

✔️ The LLM returns `AIMessage`

✔️ The LLM includes `tool_calls`

✔️ The tools produce `ToolMessages`

✔️ Loop-stopping logic counts `ToolMessages`

These assumptions are **true ONLY for OpenAI tool-calling models**.

When you switched to Groq/Gemini:

- No tool_calls → `execute_tools` never fired
- No ToolMessages → your loop never stopped
- Structured output returned Pydantic → MessageGraph rejected it
- LangGraph recursion blew up

When we fixed your code, we made it:

- provider-agnostic
- message-shaped
- relying on your own tags, not LLM internals

Now it works everywhere.

🧠 Final Mental Model (NEVER FAILS)

Before using any LLM in a graph:

Step 1 — Ask it for output

```python
print(llm.invoke("test"))
```

Step 2 — Ask yourself:

- Does it return an **AIMessage** or a **Pydantic model**?
- Does it contain **tool_calls**?
- Do I need to wrap it manually?

Step 3 — Build your graph around your own structure, not the model's quirks.

If you follow these 3 steps, you will **never again break a graph because of provider differences**.

🎁 If you want...

I can write you a **diagnostic helper script** that prints the output structure for ANY LLM so you can test models instantly.
Just say: **"give me the LLM inspector tool"**.

↳
✏️ 6. Cheatsheet: What each provider returns == | Provider | Normal output | Tool calls | Structured output | Notes | | --- | --- | --- | --- | --- | | **OpenAI** | AIMessage | ✔️ Full support | ✔️ Works | Most mature tool calling | | **Groq (LLaMA)** | AIMessage | ❌ No tool_calls | ✔️ Pydantic structured | Their LLaMA models don't do tool calling | | **Gemini** | AIMessage | ❌ No tool_calls | ✔️ Pydantic structured | Needs manual wrapping for LangGraph | | **Anthropic Claude** | AIMessage | ❌ No traditional tool_calls | ✔️ Function calling like Pydantic | Different function signature |
can you explain this with examples

Perfect — I will explain the table with **very clear, concrete code examples** so you can *see* how each provider behaves differently.

This will make it **impossible for you to misunderstand output formats ever again**.

✅ 1. OPENAI — Supports TOOL CALLING + AIMessage + Structured Output

✔️ OpenAI Normal Output

```python
from langchain_openai import ChatOpenAI
```

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

resp = llm.invoke("Say hi")
print(resp)
```

Output:

```less
AIMessage(
  content="Hi!",
  additional_kwargs={}
)
```

✔️ OpenAI Tool Calling

```python
from langchain_openai import ChatOpenAI
from langchain_core.tools import tool

@tool
def add(x: int, y: int): return x + y

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

resp = llm.invoke("add 5 and 7")
print(resp)
```

Output:

```bash
AIMessage(
  content='',
  tool_calls=[
    {
      "name": "add",
      "args": {"x": 5, "y": 7},
      "id": "call_001"
    }
  ]
)
```

⚠️ **Only OpenAI returns `tool_calls` like this.**

✔️ OpenAI Structured Output

```python
from pydantic import BaseModel

class Person(BaseModel):
    name: str
    age: int

llm = ChatOpenAI(model="gpt-4o-mini").with_structured_output(Person)
```

```
resp = llm.invoke("Name = Alice, Age = 25")
print(resp)
```

Output (Pydantic object):

```scss
Person(name="Alice", age=25)
```

Notice:
❌ Not an AIMessage
✔️ Pydantic model

✅ 2. GROQ (LLaMA) — NO TOOL CALLING, YES STRUCTURED OUTPUT

✔️ Groq Normal Output

```python
from langchain_groq import ChatGroq
llm = ChatGroq(model="llama-3.3-70b-versatile")

resp = llm.invoke("Say hi")
print(resp)
```

Output:

```less
AIMessage(content='Hi!', additional_kwargs={})
```

❌ Groq Tool Calling (NOT SUPPORTED)

If you try:

```python
llm = ChatGroq(model="llama-3.3-70b-versatile").bind_tools([add])
llm.invoke("add 5 and 7")
```

You will get:

```nginx
Does not support tool calling
```

✔️ Groq Structured Output

```python
class Person(BaseModel):
    name: str
    age: int

llm = ChatGroq(model="llama-3.3-70b-versatile").with_structured_output(Person)

resp = llm.invoke("Name Bob, age 31")
print(resp)
```

Output:

```scss
Person(name="Bob", age=31)
```

Again:
❌ Not AIMessage
✔️ Pydantic Model
❌ No tool_calls

✅ 3. GOOGLE GEMINI — NO TOOL CALLING, YES STRUCTURED OUTPUT

✔️ Gemini Normal Output

```python
from langchain_google_genai import ChatGoogleGenerativeAI

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash")

resp = llm.invoke("hello")
print(resp)
```

Output:

```less
AIMessage(content="Hello!", additional_kwargs={...})
```

❌ Gemini Tool Calling

```python
llm = ChatGoogleGenerativeAI(...).bind_tools([add])
```

This DOES NOT create OpenAI-style `tool_calls`.

✔️ Gemini Structured Output

```python
class Person(BaseModel):
    name: str
    age: int

llm = ChatGoogleGenerativeAI(model="gemini-2.0-flash").with_structured_output(Person)

resp = llm.invoke("Name: John, Age: 40")
print(resp)
```

Output:

```scss
Person(name="John", age=40)
```

Same as Groq.

✅ 4. CLAUDE — NO TOOL CALLING, HAS FUNCTION CALLING (DIFFERENT)

Anthropic's function calling looks like this:

```python
from langchain_anthropic import ChatAnthropic
from langchain_core.tools import tool

@tool
def add(x: int, y: int): return x+y

llm = ChatAnthropic(model="claude-3-opus").bind_tools([add])

resp = llm.invoke("add 10 and 20")
print(resp)
```

Output is NOT OpenAI style:

```arduino
AIMessage(
  content="Here is the function call:",
  additional_kwargs={
     "function_call": {
        "name": "add",
        "arguments": {"x": 10, "y": 20}
     }
  }
)
```

Differences from OpenAI:

- field name is `function_call` not `tool_calls`
- format is different
- ID field missing

🎯 NOW THE CHEATSHEET MAKES SENSE

Provider	AIMessage	Tool Calls	Structured Output	Example Meaning
OpenAI	✔️ Yes	✔️ YES (Full)	✔️ Yes	Best for agents
Groq	✔️ Yes	❌ No	✔️ Yes	Only structured output workflow
Gemini	✔️ Yes	❌ No	✔️ Yes	Must wrap output in LangGraph
Claude	✔️ Yes	❌ No (but has function_call)	✔️ Yes	Special JSON format

---

💡 WHY YOUR OLD CODE FAILED

Your old graph expected:

- `tool_calls` field
- `AIMessage` always returned
- messages to be convertible by LangGraph

But when you used:

❌ Groq → no tool_calls

❌ Gemini → returns Pydantic model (not a message)

❌ Your loop logic depended on ToolMessage count

So recursion never stopped.

🏁 If you want, I can also create:

✔️ A diagnostic script

Automatically detects what any LLM returns:

- AIMessage?
- Pydantic?
- tool_calls?
- function_call?
- raw JSON?

Just tell me: **"give me the inspector tool"**.

Would you like a **visual diagram** of how each provider's output flows through LangGraph?

then what is the solution when we use llm without openai

Nice, this is *exactly* the right question to ask at this point 🙌

Short answer:

> ✅ When you use **LLMs without OpenAI**, you **must stop relying on `tool_calls`** and instead:
>
> - use **structured output (Pydantic)** or plain text,
> - **wrap the result into `AIMessage` yourself**, and
> - **run tools + loops based on your own fields**, not on OpenAI's `tool_calls` / `ToolMessage` magic.

Let me turn that into a reusable pattern you can copy for *any* provider (Groq, Gemini, etc.).

---

1. Core idea: stop depending on OpenAI features

With **OpenAI + `bind_tools`** you get:

- `AIMessage.tool_calls`
- automatic `ToolMessage` wiring
- examples on the internet that assume this

With **Groq / Gemini / others**:

- `with_structured_output()` → returns a **Pydantic model**, not `AIMessage`
- No `tool_calls` field
- No automatic `ToolMessage` logic

So your **solution** is:

> **You become the orchestrator**:
>
> - LLM only gives *structured data* (`AnswerQuestion`, `ReviseAnswer`, etc.)
> - You turn it into `AIMessage`
> - You decide when to call tools
> - You decide when to loop/stop

---

2. Provider-agnostic pattern (works with Groq, Gemini, OpenAI, anything)

OpenAI

🔹 Step 1 – Draft node: structured output → AIMessage

You already have something like:

```python
from langchain_groq import ChatGroq
from schema import AnswerQuestion

llm = ChatGroq(model="llama-3.3-70b-versatile")
first_responder_llm = llm.with_structured_output(AnswerQuestion)
```

Now **wrap it** for LangGraph:

```python
from langchain_core.messages import AIMessage, BaseMessage

def draft_node(state: list[BaseMessage]) -> list[BaseMessage]:
    # Ask the LLM, giving it the history
    result: AnswerQuestion = first_responder_chain.invoke({"messages": state})

    # Wrap Pydantic → AIMessage
    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "search_queries": result.search_queries,
            "reflection": (
                result.reflection.model_dump()
                if hasattr(result, "reflection") else None
            ),
            "role": "draft",
        },
    )
    return [msg]
```

👉 This works the same whether `llm` is Groq, Gemini, OpenAI, whatever.

🔹 Step 2 – Tools node: read your own field instead of `tool_calls`

No more `tool_calls`. You defined `search_queries` yourself, so use that:

```python
import json
from typing import List, Dict
from langchain_core.messages import AIMessage, BaseMessage, ToolMessage
from langchain_community.tools import TavilySearchResults

tavily = TavilySearchResults(max_results=2)

def execute_tools(state: List[BaseMessage]) -> List[BaseMessage]:
    # get last AI message
    last_ai = next((m for m in reversed(state) if isinstance(m, AIMessage)), None)
    if not last_ai:
        return []

    search_queries = last_ai.additional_kwargs.get("search_queries", [])
    if not search_queries:
        return []

    results: Dict[str, list] = {}
    for q in search_queries:
        results[q] = tavily.invoke(q)
```

```
    # you can create a ToolMessage or just another AIMessage; both are BaseMessage
    tool_msg = ToolMessage(
        content=json.dumps(results),
        tool_call_id="manual-tools-1",
    )
    return [tool_msg]
```

Again: **no OpenAI-specific logic**. Just:

- You store queries in **`additional_kwargs`**.
- You read them.
- You call Tavily.
- You put results back into the message stream.

🔹 Step 3 – Revisor node: structured output again → AIMessage

```python
from schema import ReviseAnswer

revisor_llm = llm.with_structured_output(ReviseAnswer)

def revisor_node(state: list[BaseMessage]) -> list[BaseMessage]:
    # count previous revisions
    from langchain_core.messages import AIMessage
    past_revisions = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
        for m in state
    )

    result: ReviseAnswer = revisor_chain.invoke({"messages": state})

    msg = AIMessage(
        content=result.answer,
        additional_kwargs={
            "role": "revisor",
            "iteration": past_revisions + 1,
            "search_queries": result.search_queries,
            "reflection": (
                result.reflection.model_dump()
                if hasattr(result, "reflection") else None
            ),
            "references": result.references,
        },
    )
    return [msg]
```

🔹 Step 4 – Loop logic based on *your* tags, not `ToolMessage` count

This is the key fix that makes it **not OpenAI-specific**:

```python
from langgraph.graph import END
from typing import List

def event_loop(state: List[BaseMessage]) -> str:
    # How many times has revisor run?
    num_iterations = sum(
        isinstance(m, AIMessage) and m.additional_kwargs.get("role") == "revisor"
```

```
        for m in state
    )

    if num_iterations >= 2:   # MAX_ITERATIONS
        return END

    # Optional: stop if no more queries to search
    last_ai = next((m for m in reversed(state) if isinstance(m, AIMessage)), None)
    if not last_ai:
        return END

    if not last_ai.additional_kwargs.get("search_queries"):
        return END

    return "execute_tools"
```

This will work with:

- Groq
- Gemini
- OpenAI
- Any future provider

Because it only relies on **your own message structure**, not OpenAI's `tool_calls`.

🔹 Step 5 – Wire the graph

```python
from langgraph.graph import MessageGraph

graph = MessageGraph()

graph.add_node("draft", draft_node)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_node)

graph.add_edge("draft", "execute_tools")
graph.add_edge("execute_tools", "revisor")
graph.add_conditional_edges("revisor", event_loop)

graph.set_entry_point("draft")
app = graph.compile()

response = app.invoke("Write about how small business can leverage AI to grow")
print(response[-1].content)  # final revised answer
```

---

3. TL;DR – "Solution when we use LLM without OpenAI"

When you're **not** using OpenAI:

1. **Don't rely on `tool_calls` or `ToolMessage` counting**
2. Use `with_structured_output()` to get clean Pydantic models
3. **Always wrap them into `AIMessage`** when using LangGraph `MessageGraph`
4. Store extra data in `additional_kwargs` (`search_queries`, `reflection`, `role`, `iteration`, etc.)

5. Implement tools as **normal Python functions** that:
    - read those fields from the last message
    - run external APIs / searches
    - push results back as `ToolMessage` or `AIMessage`
6. Implement loops (`add_conditional_edges`) based on **your own fields**, like:
    - how many times `role="revisor"` appeared
    - whether there are still `search_queries` to run

Do this, and your code will **just work** with Groq, Gemini, OpenAI, whatever comes next.

If you want, next I can help you extract this into a **minimal template repo**:
`provider_agnostic_reflexion_agent.py` that you can reuse for future projects.

then can you tell me how we do this in open ai

Nice, now we're doing the reverse trip:
you already know the **provider-agnostic** way (Groq/Gemini style).
Now let's see the **"classic OpenAI way"** that your old tutorial was using.

I'll show:

1. What's *different* when using OpenAI
2. A **minimal Reflexion graph** using `ChatOpenAI` + `bind_tools`
3. How that maps to what you just built with Groq

---

1. What changes when you use OpenAI

With OpenAI you usually use:

```python
from langchain_openai import ChatOpenAI
```

and then:

```python
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
```

The **big difference** is this:

🔹 Groq / Gemini path (what you're doing now)

```python
llm.with_structured_output(AnswerQuestion)
# → returns AnswerQuestion (Pydantic object)
# You must wrap it into AIMessage yourself
```

🔹 OpenAI tools path

```python
llm.bind_tools([AnswerQuestion, ReviseAnswer])
# → returns AIMessage with .tool_calls
# You do NOT get a Pydantic model directly
# LangGraph can consume AIMessage directly — no wrapper needed
```

So in OpenAI version:

- You **don't** use **`with_structured_output`**
- You **do** use **`bind_tools`**
- You let OpenAI produce **`tool_calls`**, and your **`execute_tools`** node reads those.

---

2. Define your Pydantic "tools" (same as before)

```python
from pydantic import BaseModel, Field
from typing import List

class Reflection(BaseModel):
    missing: str = Field(description="Critique of what is missing.")
    superfluous: str = Field(description="Critique of what is superfluous.")

class AnswerQuestion(BaseModel):
    """Answer the question."""

    answer: str = Field(
        description="~250 word detailed answer to the question."
    )
    search_queries: List[str] = Field(
        description="1-3 search queries for researching improvements to address the critique of your current answer."
    )
    reflection: Reflection = Field(
        description="Your reflection on the initial answer."
    )

class ReviseAnswer(AnswerQuestion):
    """Revise your original answer to your question."""

    references: List[str] = Field(
        description="Citations motivating your updated answer."
    )
```

These same models can be used as **OpenAI tools**. LangChain knows how to turn them into a tool schema.

---

3. Build OpenAI LLM + prompt (no structured_output now)

```python
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_openai import ChatOpenAI
import datetime

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

actor_prompt_template = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """You are expert AI researcher.
Current time: {time}

You must follow this process:

1. {first_instruction}
2. Reflect and critique your answer. Be severe to maximize improvement.
3. After the reflection, list 1–3 search queries separately for researching improvements.
   - Do NOT include the search queries inside the reflection.
4. Finally, answer the user's question above using the required format.
""",
        ),
        MessagesPlaceholder(variable_name="messages"),
    ]
).partial(
    time=lambda: datetime.datetime.now().isoformat(),
)

first_responder_prompt = actor_prompt_template.partial(
    first_instruction="Provide a detailed ~250 word answer"
)

revisor_instructions = """Revise your previous answer using the new information.
    - You should use the previous critique to add important information to your answer.
        - You MUST include numerical citations in your revised answer to ensure it can be
verified.
        - Add a "References" section to the bottom of your answer (which does not count towards
the word limit). In form of:
            - [1] https://example.com
            - [2] https://example.com
    - You should use the previous critique to remove superfluous information from your answer
and make SURE it is not more than 250 words.
"""

revisor_prompt = actor_prompt_template.partial(
    first_instruction=revisor_instructions
)
```

---

4. Create OpenAI “tool-calling” chains

Here is the **OpenAI-specific** bit:

```python
# First responder: use AnswerQuestion as a TOOL
first_responder_chain = first_responder_prompt | llm.bind_tools(
    tools=[AnswerQuestion],
    tool_choice="AnswerQuestion",  # force model to call this tool
)

# Revisor: use ReviseAnswer as a TOOL
revisor_chain = revisor_prompt | llm.bind_tools(
    tools=[ReviseAnswer],
    tool_choice="ReviseAnswer",
)
```

Now when you call `first_responder_chain.invoke(...)` you **do NOT** get a Pydantic `AnswerQuestion` object.

You get an **AIMessage like**:

```python
AIMessage(
  content="",
  tool_calls=[
    {
      "name": "AnswerQuestion",
      "args": {
        "answer": "...",
        "search_queries": [...],
        "reflection": {...}
      },
      "id": "call_XYZ"
    }
  ]
)
```

No extra wrapper needed. This AIMessage can go directly into `MessageGraph`.

---

5. execute_tools node: read `.tool_calls` (OpenAI style)

This is how your old tutorial did it:

```python
import json
from typing import List, Dict, Any
from langchain_core.messages import AIMessage, BaseMessage, ToolMessage
from langchain_community.tools import TavilySearchResults

tavily_tool = TavilySearchResults(max_results=2)

def execute_tools(state: List[BaseMessage]) -> List[BaseMessage]:
    last_ai: AIMessage = state[-1]  # must be AIMessage WITH tool_calls

    if not hasattr(last_ai, "tool_calls") or not last_ai.tool_calls:
        return []

    tool_messages: List[ToolMessage] = []

    for tool_call in last_ai.tool_calls:
        name = tool_call["name"]
        args = tool_call["args"]
        call_id = tool_call["id"]

        if name in ["AnswerQuestion", "ReviseAnswer"]:
            search_queries = args.get("search_queries", [])

            query_results: Dict[str, Any] = {}
            for q in search_queries:
                result = tavily_tool.invoke(q)
                query_results[q] = result

            tool_messages.append(
                ToolMessage(
                    content=json.dumps(query_results),
                    tool_call_id=call_id,
                )
            )

    return tool_messages
```

This **relies on OpenAI** because only OpenAI gives you:

```python
AIMessage.tool_calls
```

and then LangChain expects you to respond with **`ToolMessage`** mapped by **`tool_call_id`**.

---

6. Revisor node: just use `revisor_chain` directly

With OpenAI you can add nodes directly with the chains (no custom wrapper):

```python
from langgraph.graph import MessageGraph, END

graph = MessageGraph()

graph.add_node("draft", first_responder_chain)    # returns AIMessage with tool_calls
graph.add_node("execute_tools", execute_tools)    # returns ToolMessage
graph.add_node("revisor", revisor_chain)          # sees tool result + calls ReviseAnswer

graph.add_edge("draft", "execute_tools")
graph.add_edge("execute_tools", "revisor")
```

Now: **conditional looping** (OpenAI version can use **`ToolMessage`** counting like your old code):

```python
MAX_ITERATIONS = 2

from langchain_core.messages import ToolMessage

def event_loop(state: List[BaseMessage]) -> str:
    count_tool_visit = sum(isinstance(m, ToolMessage) for m in state)

    if count_tool_visit >= MAX_ITERATIONS:
        return END
    return "execute_tools"

graph.add_conditional_edges("revisor", event_loop)
graph.set_entry_point("draft")

app = graph.compile()
```

Then:

```python
response = app.invoke("Write about how small business can leverage AI to grow")
for m in response:
    print(type(m), "→", m.content[:80])
```

---

7. Compare: OpenAI version vs Groq/Gemini version

🟦 OpenAI version

- Uses `llm.bind_tools([...])`
- LLM returns `AIMessage` **with** `.tool_calls`
- `execute_tools` reads `.tool_calls`
- You can base loop logic on `ToolMessage` count

🟧 Groq / Gemini version (what you just fixed)

- Uses `llm.with_structured_output(PydanticModel)`
- LLM returns **Pydantic model** (e.g. `AnswerQuestion`)
- You must wrap it into `AIMessage` yourself
- You put `search_queries`, `role`, `iteration` inside `additional_kwargs`
- `execute_tools` uses `additional_kwargs["search_queries"]`
- Loop logic uses your own tags (`role="revisor"`, `iteration`)

If you want, I can now:

- Take your **current working Groq code**,
- And write a **parallel OpenAI version** in one file `reflexion_graph_openai.py` so you see them side-by-side.

caan you give this all explanations without missing any part as ipynb

Nice, I've packed **all the explanations and code patterns** into a Jupyter notebook for you.

You can download it here:

👉 Download the notebook

It includes:

- Clear explanation of **MessageGraph expectations**
- Difference between:
  - **OpenAI tools pattern (`bind_tools` + `tool_calls`)**
  - **Provider-agnostic structured output pattern (`with_structured_output`)**
- Full **Groq/Gemini-friendly Reflexion graph** (Pydantic → `AIMessage`)
- Full **OpenAI-tools Reflexion graph** (using `tool_calls` and `ToolMessage`)
- All the code pieces we discussed, organized by sections

You can open it in VS Code / Jupyter Lab / Notebook and play with it while you tweak your own project.

```
from typing import List

from langchain_core.messages import BaseMessage, ToolMessage
from langgraph.graph import END, MessageGraph

from chains import revisor_chain, first_responder_chain
from execute_tools import execute_tools

graph = MessageGraph()
MAX_ITERATIONS = 2

graph.add_node("draft", first_responder_chain)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_chain)


graph.add_edge("draft", "execute_tools")
graph.add_edge("execute_tools", "revisor")

def event_loop(state: List[BaseMessage]) -> str:
    count_tool_visits = sum(isinstance(item, ToolMessage) for item in state)
    num_iterations = count_tool_visits
    if num_iterations > MAX_ITERATIONS:
        return END
    return "execute_tools"

graph.add_conditional_edges("revisor", event_loop)
graph.set_entry_point("draft")

app = graph.compile()

print(app.get_graph().draw_mermaid())

response = app.invoke(
    "Write about how small business can leverage AI to grow"
)

print(response[-1].tool_calls[0]["args"]["answer"])
print(response, "response")
is this code work with open ai ?
```

Short answer: **not exactly**.
This code is *very close* to the classic OpenAI tools pattern, but:

1. It will only work if your `first_responder_chain` / `revisor_chain` are built in the OpenAI way (with `bind_tools`).
2. Even then, your `event_loop` + final `print(...)` are a bit off and will lead to confusing behavior / errors.

Let's go through it clearly.

---

1. When *can* this work with OpenAI?

Your graph code:

```python
graph.add_node("draft", first_responder_chain)
graph.add_node("execute_tools", execute_tools)
graph.add_node("revisor", revisor_chain)
```

can work **with OpenAI** *if and only if* **`first_responder_chain`** and **`revisor_chain`** are defined like this:

```python
from langchain_openai import ChatOpenAI
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from schema import AnswerQuestion, ReviseAnswer
import datetime

llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)

actor_prompt_template = ChatPromptTemplate.from_messages([
    (
        "system",
        """You are expert AI researcher.
Current time: {time}

1. {first_instruction}
2. Reflect and critique your answer.
3. After the reflection, list 1–3 search queries separately."""
    ),
    MessagesPlaceholder(variable_name="messages"),
]).partial(time=lambda: datetime.datetime.now().isoformat())

first_responder_prompt = actor_prompt_template.partial(
    first_instruction="Provide a detailed ~250 word answer."
)

revisor_prompt = actor_prompt_template.partial(
    first_instruction="Revise your previous answer using the new information."
)

# 🔹 OpenAI tools style: bind_tools (NOT with_structured_output)
first_responder_chain = first_responder_prompt | llm.bind_tools(
    tools=[AnswerQuestion],
    tool_choice="AnswerQuestion",
)

revisor_chain = revisor_prompt | llm.bind_tools(
    tools=[ReviseAnswer],
    tool_choice="ReviseAnswer",
)
```

In this setup:

- **`first_responder_chain`** returns an **`AIMessage` with `tool_calls`** for the **`AnswerQuestion`** tool.
- **`revisor_chain`** returns an **`AIMessage` with `tool_calls`** for the **`ReviseAnswer`** tool.
- Your **`execute_tools`** function is expected to read those **`tool_calls`** and emit **`ToolMessage`**s.

If `chains.py` is **not** doing this (e.g. using `with_structured_output` instead of `bind_tools`), then your graph will **not** work as written.

---

2. Issues in this specific file

(a) `event_loop` logic

```python
def event_loop(state: List[BaseMessage]) -> str:
    count_tool_visits = sum(isinstance(item, ToolMessage) for item in state)
    num_iterations = count_tool_visits
    if num_iterations > MAX_ITERATIONS:
        return END
    return "execute_tools"
```

- This assumes that **every cycle** you will get a new `ToolMessage` (from `execute_tools`).
- That is true for the OpenAI-tools pattern *if* `execute_tools` is implemented correctly.
- But you're using `>` instead of `>=`, so with `MAX_ITERATIONS = 2` you actually allow **3 tool executions**:
  - `num_iterations = 0` → go
  - `1` → go
  - `2` → go
  - `3` → finally stop

Not a bug, but slightly weird. Most people want:

```python
if num_iterations >= MAX_ITERATIONS:
    return END
```

(b) `app.invoke(...)` input type

```python
response = app.invoke(
    "Write about how small business can leverage AI to grow"
)
```

This is actually OK: `MessageGraph` will treat the string as a `HumanMessage` and start the graph with that.

The important part is that your **nodes** (e.g. `first_responder_chain`) must accept `List[BaseMessage]` as input, or be wrapped to do so. If your prompt uses `MessagesPlaceholder("messages")`, then the underlying chain expects a dict like `{"messages": state}`.

So the safest pattern is to wrap the chains in tiny node functions:

```python
from langchain_core.messages import BaseMessage

def draft_node(state: List[BaseMessage]) -> List[BaseMessage]:
    ai = first_responder_chain.invoke({"messages": state})
    return [ai]

def revisor_node(state: List[BaseMessage]) -> List[BaseMessage]:
    ai = revisor_chain.invoke({"messages": state})
    return [ai]

graph.add_node("draft", draft_node)
graph.add_node("revisor", revisor_node)
```

Then you are 100% sure the types line up.

(c) The final `print(...)` is wrong

```python
print(response[-1].tool_calls[0]["args"]["answer"])
print(response, "response")
```

At the end of the graph, `response` is a **list of messages** (state), something like:

```python
[
  HumanMessage(...),
  AIMessage(tool_calls=[AnswerQuestion...]),
  ToolMessage(...),
  AIMessage(tool_calls=[ReviseAnswer...]),
  ToolMessage(...),  # from last execute_tools
]
```

So:

- `response[-1]` is likely a `ToolMessage`, not an `AIMessage`.
- `ToolMessage` has no `.tool_calls` attribute → this will error.

If you want to get the **final answer text** from the `ReviseAnswer` tool call, you should:

1. Find the **last `AIMessage` that has `tool_calls`**.
2. Grab the `"answer"` field from its args.

Example:

```python
from langchain_core.messages import AIMessage

# find last AIMessage with tool_calls
ai_with_tool = None
for msg in reversed(response):
    if isinstance(msg, AIMessage) and getattr(msg, "tool_calls", None):
        ai_with_tool = msg
        break
```

```
if ai_with_tool is None:
    print("No tool_calls found in final state.")
else:
    args = ai_with_tool.tool_calls[0]["args"]
    print("Final answer:", args["answer"])
```

That is the safe OpenAI-tools way to get the structured answer.

---

3. So… does your exact file "work with OpenAI"?

As-is:

- ✅ The **graph structure** is valid for an OpenAI-tool workflow.
- ❌ It will **only run correctly** if:
  - `first_responder_chain` and `revisor_chain` are built with `ChatOpenAI(...).bind_tools(...)`.
  - `execute_tools` reads `tool_calls` and returns `ToolMessage`s.
- ⚠️ Your `event_loop` condition is slightly off (`>` vs `>=`), and
- ❌ Your final `print(response[-1].tool_calls[...])` is wrong and will break.

If you want, I can give you a **full minimal OpenAI version** of:

- `schema.py`
- `chains.py`
- `execute_tools.py`
- `reflexion_graph.py`

all wired together and ready to run with `OPENAI_API_KEY`, so you have a clean reference implementation.