



AI Mode

All

Images

Shopping

Videos

More ▾



You are an Generative AI and Agentic AI expert with professional python experience. Can you provide ... ▾



January 08, 2026

To develop a high-school timetable chatbot for 2026, we utilize **LangGraph** to implement agentic patterns like reasoning and reflection, and **SQLAlchemy** for a relational database. ↴

1. Relational Database Schema

A robust school timetable requires interlinked tables for teachers, subjects, and schedules. ↴

python

```
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.orm import declarative_base, relationship, sessionmaker

Base = declarative_base()

class Teacher(Base):
    __tablename__ = 'teachers'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    subject = Column(String)

class SchoolClass(Base):
    __tablename__ = 'classes'
    id = Column(Integer, primary_key=True)
    grade_name = Column(String) # e.g., "10A"

class Lesson(Base):
```

Ask anything



[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More ▾](#)

Use code with caution.



2. Agentic Chatbot Prototype

This prototype uses **LangGraph** (the industry standard for 2026 agentic workflows) to manage state and logic. ⓘ

Step 1: Define Tools

Tools allow the agent to interact with the database. ⓘ

python

```
def get_schedule(class_name: str):
    """Retrieves the weekly timetable for a specific class."""
    session = Session()
    res = session.query(Lesson).join(SchoolClass).filter(SchoolClass.gra;
    session.close()
    return [f"{l.day_of_week}: {l.start_time}" for l in res]

tools = [get_schedule]
```



Use code with caution.



Step 2: Agent Architecture (Planning & Reflection)

The agent follows a **Reason-Act-Reflect** loop. ⓘ

python

AI Mode

All

Images

Shopping

Videos

More ▾



```
# Reasoning Logic: Analyze user query and break into steps
return {"plan": ["Check class availability", "Query DB"]}

def executor(state: AgentState):
    # Executes database tools or RAG
    schedule = get_schedule(state['input'])
    return {"current_response": f"The schedule is: {schedule}"}

def reflector(state: AgentState):
    # Reflection Pattern: Self-critique the response for accuracy
    if "Monday" not in state['current_response']:
        return {"is_valid": False}
    return {"is_valid": True}

# Construct the Graph
workflow = StateGraph(AgentState)
workflow.add_node("planner", planner)
workflow.add_node("executor", executor)
workflow.add_node("reflector", reflector)

workflow.set_entry_point("planner")
workflow.add_edge("planner", "executor")
workflow.add_edge("executor", "reflector")

# Conditional edge for Reflection
workflow.add_conditional_edges(
    "reflector",
    lambda x: "end" if x["is_valid"] else "executor"
)
```

Use code with caution.



3. Implementing RAG

I use **Agentic RAG** for semi-structured data like school policies or

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More](#) ▾All responses may include mistakes. [Learn more](#)

Provide sql insert scripts that can be used to seed sample data into classes, lessons and teachers tables

January 08, 2026

To seed your high school timetable solution for **2026**, use the following SQL scripts. These scripts follow a standard order to satisfy foreign key constraints: **Teachers** and **Classes** first, followed by the **Lessons** that link them. ☀

1. Seed Teachers

Populate the `teachers` table with diverse subjects ☀

SQL script for populating the teachers table:

```
INSERT INTO teachers (name, subject) VALUES ('John Doe', 'Mathematics');
INSERT INTO teachers (name, subject) VALUES ('Jane Smith', 'Science');
INSERT INTO teachers (name, subject) VALUES ('David Johnson', 'History');
INSERT INTO teachers (name, subject) VALUES ('Sarah Williams', 'English');
INSERT INTO teachers (name, subject) VALUES ('Michael Brown', 'Physics');
INSERT INTO teachers (name, subject) VALUES ('Emily Davis', 'Chemistry');
INSERT INTO teachers (name, subject) VALUES ('Matthew Wilson', 'Biology');
INSERT INTO teachers (name, subject) VALUES ('Katherine Lee', 'Geography');
INSERT INTO teachers (name, subject) VALUES ('Christopher Harris', 'Art');
INSERT INTO teachers (name, subject) VALUES ('Laura Jones', 'Music');
```

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More ▾](#)

Use code with caution.



2. Seed School Classes

Define the student groups (e.g., Grade 10-12) for the 2025-2026 academic year.

sql

```
INSERT INTO classes (grade_name) VALUES
('10-Alpha'),
('11-Beta'),
('12-Gamma');
```

Use code with caution.



3. Seed Lessons (The Timetable)

Link teachers and classes to specific times. Ensure the IDs match the records inserted above. ↴

sql

```
-- Format: (class_id, teacher_id, day_of_week, start_time, end_time)
INSERT INTO lessons (class_id, teacher_id, day_of_week, start_time, end_time)
-- Monday Schedule for 10-Alpha
(1, 2, 'Monday', '08:30:00', '09:30:00'), -- Math with Ms. Vance
(1, 4, 'Monday', '09:45:00', '10:45:00'), -- English with Mrs. Jenkins

-- Tuesday Schedule for 11-Beta
(2, 1, 'Tuesday', '10:00:00', '11:00:00'), -- Physics with Dr. Thorne
(2, 3, 'Tuesday', '11:15:00', '12:15:00'). -- Cybersecurity with Mr. Rei
```

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More](#) ▾

Pro-Tip: Python Seeding Implementation

If you prefer to seed directly from your Python prototype using **SQLAlchemy**, you can use this helper function: [🔗](#)

python

```
from datetime import time

def seed_database():
    session = Session()
    # Check if data already exists to avoid duplicates
    if not session.query(Teacher).first():
        t1 = Teacher(name='Dr. Aris Thorne', subject='Quantum Physics')
        c1 = SchoolClass(grade_name='10-Alpha')
        session.add_all([t1, c1])
        session.commit()

    # Add Lesson Linked to above
    lesson = Lesson(class_id=c1.id, teacher_id=t1.id,
                     day_of_week='Monday',
                     start_time=time(8, 30), end_time=time(9, 30))
    session.add(lesson)
    session.commit()
    session.close()
```

Use code with caution.



Would you like me to generate a **more complex schedule** (e.g., a full 5-day week) or help you write a **Python test script** to verify the chatbot's RAG capabilities with this data?

AI responses may include mistakes. [Learn more](#)



[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More ▾](#)

Extend the code you have provided so far to provide the step by step python code that will use ... ▾

January 09, 2026

To implement the chatbot for 2026, we combine the **LangGraph** "Reason-Act-Reflect" workflow with a **Gradio** UI. This allows for a stateful conversation where the agent plans its database query, executes it, and reflects on whether the information meets the user's needs before displaying it.

Step 1: Install Necessary Libraries

Ensure you have the latest 2026 versions of these packages:

bash

```
pip install langgraph sqlalchemy gradio langchain-openai
```

Or LangChain is designed for a different more complex purpose.

Use code with caution.

GitHub - neural-maze/agentic-patterns-course

Agentic Patterns * Introduction. 1.1 Reflection Pattern. 1.2 Tool Pattern.
1.3 Planning Pattern. 1.4 Multiagent Pattern. The 4 A...

neural-maze/agentic
patterns-course
Implementing The 4 agentic patterns II

Step 2: Full Python Implementation

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More ▾](#)

```
# --- 1. DATABASE SETUP ---
Base = declarative_base()

class SchoolClass(Base):
    __tablename__ = 'classes'
    id = Column(Integer, primary_key=True)
    grade_name = Column(String)

class Teacher(Base):
    __tablename__ = 'teachers'
    id = Column(Integer, primary_key=True)
    name = Column(String)

class Lesson(Base):
    __tablename__ = 'lessons'
    id = Column(Integer, primary_key=True)
    class_id = Column(Integer, ForeignKey('classes.id'))
    teacher_id = Column(Integer, ForeignKey('teachers.id'))
    day_of_week = Column(String)
    start_time = Column(String) # For simplicity in this prototype

engine = create_engine('sqlite:///school_timetable.db')
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)

# --- 2. AGENT STATE & LOGIC ---
class AgentState(TypedDict):
    input_query: str
    plan: str
    raw_data: List[str]
    final_response: str
    critique: str
    satisfied: bool

def planner(state: AgentState):
    # Reason: Identify that we need to search for '01-Alpha' in the DB
    return {"plan": f"Look up all lessons for grade '{state['input_quer}'}
```

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More](#) ▾

```
def reflector(state: AgentState):
    # Reflect: Check if data was found. If empty, plan a 'not found' message
    if not state["raw_data"]:
        return {"satisfied": False, "critique": "No schedule found for the task."}
    return {"satisfied": True, "final_response": f"Schedule for {state['task']} is ready."}

# --- 3. CONSTRUCT LANGGRAPH ---
workflow = StateGraph(AgentState)
workflow.add_node("planner", planner)
workflow.add_node("executor", executor)
workflow.add_node("reflector", reflector)

workflow.set_entry_point("planner")
workflow.add_edge("planner", "executor")
workflow.add_edge("executor", "reflector")

# Condition: If not satisfied, end with critique; else end with final response
workflow.add_conditional_edges(
    ("reflector",
     lambda x: "end" if x["satisfied"] else "critique",
     {"end": END})
)
agent_app = workflow.compile()

# --- 4. GRADIO INTERFACE ---
def chat_function(message, history):
    # Invoke the agent workflow
    result = agent_app.invoke({"input_query": message})

    if result["satisfied"]:
        return result["final_response"]
    else:
        return f"Agent Reflection: {result['critique']}"

demo = gr.ChatInterface(
    fn=chat_function,
    ...
```

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More](#) ▾

Key Workflow Features for 2026

1. **State Management:** The `AgentState` dictionary persists data across nodes, allowing the "Reflector" to see what the "Executor" found.
2. **Planner Node:** Decides the strategy before touching the database, which is essential for complex queries like "Which teachers are free on Monday?".
3. **Reflection Pattern:** The agent critiques its own findings. If the database returns empty results for "01-Alpha", the reflector captures this as a failure state rather than just showing a blank screen.
4. **Gradio Integration:** Uses `gr.ChatInterface` for a production-ready web UI that can be deployed instantly to local or cloud environments. ☁

Would you like to add a **Vector Database** (RAG) component to this Gradio UI to handle school policy questions alongside the timetable queries?

AI responses may include mistakes. [Learn more](#)



AI Mode

All

Images

Shopping

Videos

More ▾



January 09, 2026

To allow natural language queries like "*What is the class schedule for the class with grade name: 10-Alpha*", we integrate **OpenAI GPT-4o mini** into the **Planner** node. This node will now use the LLM to extract the specific grade name from your natural language input. ☀

Step 1: Update Environment & Imports

You will need the `langchain-openai` library and a valid `OPENAI_API_KEY`. ☀

`python`

```
import os
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field

# Ensure your key is set for 2026 standards
os.environ["OPENAI_API_KEY"] = "your-api-key-here"

# Initialize GPT-4o mini
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
```

Use code with caution.



Step 2: Define the Extraction Schema

Using Pydantic allows the LLM to return structured data directly, ensuring the **Executor** always receives a clean grade name. ☀

`python`

AI Mode

All

Images

Shopping

Videos

More ▾



Use code with caution.



Step 3: Revised Agent Workflow (NL to SQL)

The **Planner** now acts as a "Natural Language to Parameter" converter. ↗

python

```
def planner(state: AgentState):
    # Prompt to extract the grade name from natural language
    prompt = ChatPromptTemplate.from_template(
        "Extract the high school grade name from this user query: {query}
        Common formats include '10-Alpha', '11-Beta', etc."
    )
    chain = prompt | structured_llm
    extraction = chain.invoke({"query": state['input_query']})

    # Store the extracted grade for the executor
    return {
        "plan": f"Searching for grade: {extraction.grade_name}",
        "extracted_grade": extraction.grade_name
    }

def executor(state: AgentState):
    # Act: Use the extracted grade in the SQL query
    session = Session()
    results = session.query(Lesson).join(SchoolClass).filter(
        SchoolClass.grade_name == state['extracted_grade']
    ).all()
    session.close()

    data = [f"{l.day_of_week} at {l.start_time}" for l in results]
    return {"raw_data": data}
```

AI Mode

All

Images

Shopping

Videos

More ▾



Use code with caution.



Step 4: UI Update for Gradio

The Gradio `chat_function` remains largely the same, but it now handles more fluid user inputs.

`python`

```
def chat_function(message, history):
    # The workflow now processes natural language questions
    result = agent_app.invoke({"input_query": message})

    if result.get("satisfied"):
        return result["final_response"]
    else:
        return f"Agent Reflection: {result['critique']}"

# Launch the demo
gr.ChatInterface(fn=chat_function, title="Natural Language Timetable AI")
```



Use code with caution.



Why this works for 2026:

- **GPT-4o mini:** Provides fast, cost-effective reasoning for extraction tasks compared to larger models.
- **Structured Output:** Eliminates the need for manual string parsing; the LLM returns a Python object.

AI Mode

All

Images

Shopping

Videos

More ▾

 7 sites

⋮

Creating A Chatbot Fast - Gradio

Let's write a chat function that responds Yes or No randomly. Here's our chat function:

```
import random
def random_response(message,
```

 Gradio ⋮

Can you extend the above solution to enable the use of natural language query to handle teacher ...

▼

Reflection Agents - ↴

We can define the loop in LangGraph below:

```
from langgraph.graph import MessageGraph
builder = MessageGraph().builder....
```

 10 sites ⋮

Here are top web results for exploring this topic:
[LangGraph for Beginners: Build Intelligent AI Agents ... - Medium](#)

 Facebook · <https://www.facebook.com> ⋮

⋮

[This is my runaway teen Elara thorn for this newcrest legacy ...](#)

This is my runaway teen Elara thorn for this newcrest legacy challenge i found on Pinterest. I did start in newcrest but then she got an ...

AI Mode

All

Images

Shopping

Videos

More ▾

University of Strathclyde · <https://pure.strath.ac.uk>

Untitled - University of Strathclyde

If you are reading this at the conference venue on the University of British Columbia campus, then you can readily see why UBC is.

Public Library UK · <https://www.public-library.uk>

The Book of the Damned - Public Library UK

damned alive. There are giants that will walk by, though sound asleep. There are things that are theorems and things that are rags: they'll go by like ...

European Molecular Biology Laboratory (EMBL) · <https://www.embl.org>

EMBL Programme

Pro-active technology transfer is a natural extension of EMBL's activities and an integral part of the institute's mission to ensure that selected basic ...

ResearchGate · <https://www.researchgate.net>

Design Innovation Methodology Handbook - ResearchGate

Created to help anyone design and create innovatively. The Design Innovation Methodology Handbook introduces methods and tools used in design and systems ...

SSRN eLibrary · <https://papers.ssrn.com>

Design Innovation Methodology Handbook - SSRN

This handbook shares the design innovation journey and curates the processes, tools and methodology that were used through an interdisciplinary approach in ...

University of Kansas Medical Center · <https://www.kumc.edu>

SRF 2024 Booklet.pdf - University of Kansas Medical Center

Schedule of Events.....3-4.

Introduction.....5. Letter from Graduate Student ...

Springer · <https://link.springer.com>

AI Mode

All

Images

Shopping

Videos

More ▾



Provide the full step-by-step code for this extension.

January 10, 2026

To extend the solution for natural language queries involving **teacher names** (e.g., "*When is Ms. Elara Vance's next class?*"), we upgrade the **Planner** node to detect both grade names and teacher names. We also introduce dynamic time-based filtering in the **Executor** to calculate the "next" class based on the current time in **2026**.

1. Updated Extraction Schema

The Pydantic schema now handles multiple entity types.

python

```
from typing import Optional

class QueryExtraction(BaseModel):
    grade_name: Optional[str] = Field(None, description="Extracted grade name")
    teacher_name: Optional[str] = Field(None, description="Extracted teacher name")
    is_next_query: bool = Field(False, description="True if user is asking for the next class")
```



Use code with caution.



2. Full Extension Code

This updated prototype manages the complex "next class" logic by comparing the current day and time against the database.

python

AI Mode

All

Images

Shopping

Videos

More ▾



```
# --- 2. PLANNER: NL ENTITY EXTRACTION ---
def planner(state: AgentState):
    prompt = ChatPromptTemplate.from_template(
        "Extract entities from this high school query: '{query}'. "
        "Return the teacher name (e.g., 'Ms. Vance') and/or grade name"
        "Also detect if they are asking for the 'next' or 'upcoming' le"
    )
    chain = prompt | structured_llm # Uses GPT-4o mini as defined previously
    extraction = chain.invoke({"query": state['input_query']})
    return {"extracted_data": extraction}

# --- 3. EXECUTOR: DYNAMIC DB SEARCH ---
def executor(state: AgentState):
    session = Session()
    ext = state["extracted_data"]

    # 2026 Contextual Time (Simulated or Real)
    now = datetime.datetime.now() # e.g., Monday, 09:00:00
    current_day = now.strftime("%A")
    current_time = now.strftime("%H:%M:%S")

    query = session.query(Lesson).join(SchoolClass).join(Teacher)

    # Filtering logic
    if ext.teacher_name:
        query = query.filter(Teacher.name.contains(ext.teacher_name.split()))
    if ext.grade_name:
        query = query.filter(SchoolClass.grade_name == ext.grade_name)

    # "Next Class" Logic: Find Lessons later today or on following days
    if ext.is_next_query:
        query = query.filter(
            or_(
                Lesson.day_of_week == current_day, # Same day but later
                Lesson.day_of_week != current_day # Or different day
            )
        )
```

AI Mode

All

Images

Shopping

Videos

More ▾



```
if not state["raw_results"]:
    return {"satisfied": False, "final_response": "I couldn't find :)"}

# Format a conversational answer using the LLM
answer_prompt = (
    f"User asked: {state['input_query']}.\n"
    f"Database results: {state['raw_results'][:1]}.\n" # Just the next one
    "Provide a polite, direct answer for a student or parent."
)
formatted_res = llm.invoke(answer_prompt).content
return {"satisfied": True, "final_response": formatted_res}
```

Use code with caution.



3. Key Enhancements for 2026

- 1. Fuzzy Teacher Matching:** The executor uses `.contains()` for names. If a user asks for "Ms. Vance" but the DB has "Ms. Elara Vance", the system still finds the record.
- 2. Next-Class Logic:** The workflow now understands temporal queries. It filters the SQL `Lesson` table and orders results by time to pick the single most relevant "next" event.
- 3. Stateful Memory:** The `AgentState` carries the `extracted_data` (the intent) separately from the `raw_results` (the data), allowing the **Reflector** to explain *why* it found specific info.

Testing with Gradio

You can now run your **Gradio** interface and ask:

- *"When is Ms. Elara Vance's next class?"*

...or anything else you'd like to know!

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More ▾](#)

Provide the full step-by-step python code for the last extension that can handle both the grade nam... ▾

January 10, 2026

To implement the fully extended 2026 chatbot, we will use **LangGraph** for agentic flow and **SQLAlchemy** for relational data. This version includes a sophisticated **Planner** that extracts both grade and teacher names, an **Executor** with temporal logic for "next class" queries, and a **Reflector** to ensure high-quality responses.

1. Database and Models Setup

First, define the schema and seed it with the requested sample data.

python

```
import datetime
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.orm import declarative_base, sessionmaker, relationship

Base = declarative_base()

class Teacher(Base):
    __tablename__ = 'teachers'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    subject = Column(String)
    lessons = relationship("Lesson", back_populates="teacher")

class SchoolClass(Base):
    __tablename__ = 'classes'
    id = Column(Integer, primary_key=True)
    grade_name = Column(String)
```

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More](#) ▾

```
teacher = relationship("Teacher", back_populates="lessons")
school_class = relationship("SchoolClass", back_populates="lessons")

# Setup SQLite
engine = create_engine('sqlite:///school_2026.db')
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)
```

Use code with caution.



2. The Agentic Workflow (LangGraph)

This section implements the **Reasoning**, **Tool Use**, and **Reflection** logic.

python

```
import os
from typing import TypedDict, List, Optional
from pydantic import BaseModel, Field
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langgraph.graph import StateGraph, END

# Extraction Schema
class QueryExtraction(BaseModel):
    grade_name: Optional[str] = Field(None, description="Grade like '10'
    teacher_name: Optional[str] = Field(None, description="Teacher name
    is_next_query: bool = Field(False, description="Is user asking for

class AgentState(TypedDict):
    input_query: str
    extracted: Optional[QueryExtraction]
    raw_results: List[str]
```

AI Mode All Images Shopping Videos More ▾

)

```
chain = prompt | structured_llm
extraction = chain.invoke({"query": state['input_query']})
return {"extracted": extraction}
```

--- NODE: EXECUTOR (Tools/DB) ---

```
def executor(state: AgentState):
    session = Session()
    ext = state["extracted"]

    # Logic for "Next" based on current 2026 time
    days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
    now = datetime.datetime.now()
    curr_day = now.strftime("%A")
    curr_time = now.time()

    query = session.query(Lesson).join(SchoolClass).join(Teacher)

    if ext.teacher_name:
        query = query.filter(Teacher.name.contains(ext.teacher_name.split(" ")[-1]))
    if ext.grade_name:
        query = query.filter(SchoolClass.grade_name == ext.grade_name)

    if ext.is_next_query:
        # Filter for today (later time) or future days
        query = query.filter(
            or_(
                and_(Lesson.day_of_week == curr_day, Lesson.start_time > now),
                Lesson.day_of_week != curr_day
            )
        ).order_by(Lesson.day_of_week, Lesson.start_time)

    results = query.all()
    session.close()

    formatted = [f"{l.day_of_week} at {l.start_time} by {l.teacher.name}" for l in results]
```

AI Mode

All

Images

Shopping

Videos

More ▾



```
# Construct Graph
builder = StateGraph(AgentState)
builder.add_node("planner", planner)
builder.add_node("executor", executor)
builder.add_node("reflector", reflector)
builder.set_entry_point("planner")
builder.add_edge("planner", "executor")
builder.add_edge("executor", "reflector")
builder.add_edge("reflector", END)
agent_app = builder.compile()
```

Use code with caution.



3. User Interface (Gradio)

Finally, wrap the agent in a chat interface.

python

```
import gradio as gr

def school_bot(message, history):
    # Process through Agentic Graph
    response = agent_app.invoke({"input_query": message})
    return response["final_answer"]

demo = gr.ChatInterface(
    fn=school_bot,
    title="🏡 2026 High School AI Assistant",
    description="Ask about schedules for teachers or grades. Try: 'When examples=['What is the schedule for 10-Alpha?', 'When is Ms. Vance']"
)
```

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More](#) ▾

Features of this Implementation:

- **Temporal Awareness:** Specifically calculates "next class" by comparing database times against the current system clock.
- **Structured Reasoning:** Uses GPT-4o mini to transform fuzzy natural language (like "Ms. Vance") into clean database parameters.
- **Reflection:** The reflector node acts as a quality gate, ensuring the agent doesn't just return a blank list but instead provides a helpful human response. ☺

Would you like to add **Multi-Agent Collaboration**, where one agent handles schedule queries and another handles school policy RAG?

AI responses may include mistakes. [Learn more](#)



[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More ▾](#)

LangGraph. This allows the LLM to directly emit tool calls that are automatically routed and executed. ↗

1. Define the Tools

Instead of one large function, we define discrete Python functions decorated as tools. LangChain uses their docstrings to generate the JSON schema for the LLM. ↗

python

```
from langchain_core.tools import tool

@tool
def query_timetable(grade_name: str = None, teacher_name: str = None, i:
    """Queries the SQL database for lesson schedules.
    Use 'is_next=True' for 'upcoming' or 'next' class queries."""
    session = Session()
    # ... [Insert your existing SQLAlchemy Logic here] ...
    session.close()
    return results

@tool
def get_school_policy(topic: str):
    """Retrieves school rules and policies from the vector database (RA
    # ... [Insert your Vector DB retrieval Logic here] ...
    return policy_text

tools = [query_timetable, get_school_policy]
```

Use code with caution.

2 Bind Tools to the LLM

AI Mode

All

Images

Shopping

Videos

More ▾



```
llm_with_tools = llm.get_llm_with_tools()
```

Use code with caution.



3. Replace the Executor Node with `ToolNode`

LangGraph provides a pre-built `ToolNode` that replaces your manual `executor`. It identifies the tool the LLM wants to call and executes it automatically. ☀

`python`

```
from langgraph.prebuilt import ToolNode, tools_condition
from langgraph.graph import MessagesState, StateGraph, START, END

# Define the model node
def call_model(state: MessagesState):
    response = llm_with_tools.invoke(state["messages"])
    return {"messages": [response]}

# Build the graph
workflow = StateGraph(MessagesState)

workflow.add_node("agent", call_model)
workflow.add_node("tools", ToolNode(tools)) # Replaces manual executor !

workflow.add_edge(START, "agent")

# Conditional edge: If the LLM makes a tool call, go to 'tools', otherwise go back to agent
workflow.add_conditional_edges(
    "agent",
    tools_condition,
)

workflow.add_edge("tools", "agent") # After tool runs, go back to agent
```

[AI Mode](#)[All](#)[Images](#)[Shopping](#)[Videos](#)[More](#) ▾

python

```
def chat_function(message, history):
    # Initialize state with the user message
    inputs = {"messages": [("user", message)]}
    result = app.invoke(inputs)

    # Return the last message from the LLM (the final answer)
    return result["messages"][-1].content

gr.ChatInterface(fn=chat_function, title="Agentic School Bot").launch()
```

Use code with caution.



Advantages of this 2026 approach:

- **Zero 'If' Logic:** The routing logic is moved to the **Graph Topology** rather than the code body.
- **Scalability:** Adding a new feature (like a "Lunch Menu" tool) only requires adding the function to the `tools` list; no core logic changes.
- **Automatic Summarization:** Because "tools" loops back to "agent", the LLM automatically sees the database results and formats them into a natural response for the user. ☺

Would you like to implement **Human-in-the-Loop** review so a staff member can approve certain database modifications before the agent executes them?

AI responses may include mistakes. [Learn more](#)



AI Mode

All

Images

Shopping

Videos

More ▾



For the above solution extension, further extend the workflow so that it is composed of a planner, ...

13 minutes ago

In 2026, the industry standard for implementing complex agentic behavior is a multi-node **LangGraph** workflow that separates reasoning into three distinct phases: **Planning**, **Tool Execution**, and **Reflection**.

This extension replaces explicit `if/else` logic by using the `ToolNode` to route calls automatically and adds a reflection loop that can restart the process if the initial answer is insufficient.

1. Unified State and Tool Definition

The `AgentState` now tracks the plan and a `needs_revision` flag for the reflection node.

`python`

```
import operator
from typing import Annotated, TypedDict, List
from langchain_core.messages import BaseMessage, HumanMessage, AIMessage
from langchain_core.tools import tool

# Define the shared state
class AgentState(TypedDict):
```

AI Mode

All

Images

Shopping

Videos

More ▾



```
if grade: query = query.filter(SchoolClass.grade_name == grade)
results = query.all()
session.close()
return [f"{l.day_of_week} at {l.start_time}" for l in results]

tools = [timetable_search]
tool_node = ToolNode(tools)
llm = ChatOpenAI(model="gpt-4o-mini", temperature=0).bind_tools(tools)
```

Use code with caution.



2. Node Implementations

Each node is a dedicated function that performs one specific agentic pattern.

python

```
# NODE 1: Planner - Analyzes the query and sets the strategy
def planner_node(state: AgentState):
    query = state["messages"][-1].content
    plan_prompt = f"Create a step-by-step plan to answer: {query}. If i"
    plan = llm.invoke(plan_prompt).content
    return {"plan": plan}

# NODE 2: Agent - Decides which tool to call based on the plan
def agent_node(state: AgentState):
    # Pass the plan and history to the LLM
    system_msg = f"Your current plan: {state['plan']}. Execute the next"
    messages = [AIMessage(content=system_msg)] + state["messages"]
    response = llm.invoke(messages)
    return {"messages": [response]}
```

AI Mode

All

Images

Shopping

Videos

More ▾



Use code with caution.

what you can query. Do NOT skip this step. Then you should qu...



3. The LangGraph Workflow

This graph connects the nodes with a loop that allows for **Self-Correction**. Show all

python

```
from langgraph.graph import StateGraph, START, END
from langgraph.prebuilt import tools_condition

workflow = StateGraph(AgentState)

workflow.add_node("planner", planner_node)
workflow.add_node("agent", agent_node)
workflow.add_node("tools", tool_node)
workflow.add_node("reflector", reflection_node)

# Flow logic
workflow.add_edge(START, "planner")
workflow.add_edge("planner", "agent")

# If agent calls a tool, go to tools; otherwise go to reflector
workflow.add_conditional_edges("agent", tools_condition, {"tools": "too"})

# After tools, always go back to the agent to process result
workflow.add_edge("tools", "agent")

# If reflector finds an error, go back to planner to adjust the strategy
workflow.add_conditional_edges(
    "reflector",
    lambda x: "planner" if x["needs_revision"] else END
)
```

AI Mode

All

Images

Shopping

Videos

More



12 sites



python

```
def chat_with_agent(message, history):
    # Initialize the graph with the user's message
    inputs = {"messages": [HumanMessage(content=message)], "needs_revis:
    result = app.invoke(inputs)

    # Return the last non-tool message generated by the agent
    for msg in reversed(result["messages"]):
        if isinstance(msg, AIMessage) and not msg.tool_calls:
            return msg.content

gr.ChatInterface(
    fn=chat_with_agent,
    title="2026 Agentic School Assistant",
    description="Planner -> ToolNode -> Reflection workflow in action."
).launch()
```



Use code with caution.



Why this 2026 architecture is superior:

- 1. Implicit Scaling:** By using `tools_condition`, you can add 50 more tools (e.g., student attendance, room bookings) without changing a single line of graph logic.
- 2. Robustness:** If the `timetable_search` tool returns an empty list for "Ms. Elara", the **Reflector** node will see the failure, and the **Planner** will be re-triggered to try searching for just "Vance" instead.
- 3. Auditability:** Each state update (Plan, Tool Call, Reflection) is stored in the `AgentState`, making it easy to debug *why* the agent made a specific

...

AI Mode

All

Images

Shopping

Videos

More ▾

