

# Building Agentic AI Applications

---

Now that you've mastered Retrieval-Augmented Generation (RAG), it's time to take the next leap: Agentic AI.

While RAG enhances an LLM's knowledge, agents enhance its ability to act. An AI agent is not just a chatbot — it's an autonomous system that can perceive, plan, act, and reflect to achieve a goal.

In this chapter, you'll learn how to build agents that go beyond answering questions — they execute tasks, use tools, and make decisions.

We'll extend **TaskFriend** from a passive assistant into an active agent that can help users plan trips, check calendars, and fetch weather — all by orchestrating multiple specialized agents.

Let's begin by understanding what makes an AI system "agentic."

## The story so far...

**TaskFriend** is quickly shooting up the leaderboards in App Stores and Markets all around the globe. But with great userbase comes great responsibility:

Users now want more - instead of just planning their day, they want to plan **trips** to work around their schedules

You notice that you can't seem to use any of the previous optimization techniques to add new features:

- Adding to system prompts and templates seem to make the system more fragile
- The more questions users pose, the less specialized your base LLM becomes
- You can't rely on just your knowledgebases anymore
- You need input from real-time data
- You need to find a way to optimize LLM usage as costs are slowly getting out of hand

## Goals

- Break down complex goals into steps
- Create agents to help make decisions based on user input
- Orchestrate multiple specialized agents
- Use tools and APIs to retrieve real-time data
- Synthesize results into a coherent, user-friendly plan

## Intitalizing the environment

### Setting up the API key

Before we start work on in any notebook, we'll need to load the [API key for Model Studio](#). This ensures that we can call APIs of Qwen models we'll be using throughout this course.

If you're unsure about how to find your **Model Studio** API key, refer to the [00 Setting Up the Environment](#) file.

```
# Load Model Studio API key
import os
from config.load_key import load_key
load_key(
    confirmation=False
)
```

```
# Set global settings
import dashscope
from llama_index.core import Settings, VectorStoreIndex,
SimpleDirectoryReader
from llama_index.embeddings.dashscope import DashScopeEmbedding
from llama_index.llms.openai_like import OpenAILike

# Dashscope uses https://dashscope-intl.aliyuncs.com/api/v1
# instead of https://dashscope-intl.aliyuncs.com/compatible-mode/v1
dashscope.base_http_api_url = "https://dashscope-intl.aliyuncs.com/api/v1"

Settings.llm=OpenAILike(
    model="qwen-plus",
    api_base="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    is_chat_model=True
)

Settings.embed_model = DashScopeEmbedding(
    model_name="text-embedding-v2",
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    encoding_format="float"
)

print("✅ Global parameters set!")
```

```
from openai import OpenAI

client = OpenAI(
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
)

def get_qwen_stream_response(query, print_stream=True):
    completion = client.chat.completions.create(
        model="qwen-plus",
        messages=[{"role": "user", "content": query}],
        stream=True
    )
```

```

)
full_response = ""
for chunk in completion:
    try:
        content = chunk.choices[0].delta.content
        if content:
            if print_stream:
                print(content, end="", flush=True)
            full_response += content
    except Exception as e:
        print(f"[Stream error: {e}]", flush=True)
return full_response

```

## Getting to Know the Agent Framework

---

An AI agent is composed of four core components:



What makes an agent

### 1. Planning

- The agent breaks down goals into steps.
- Uses reasoning (e.g., chain-of-thought) to decide what to do next.
- Example: "User wants to plan a trip → First, check calendar availability."

### 2. Memory

- Short-term: Conversation history (stored in thread).
- Long-term: Vector database of user preferences, past trips, etc.
- Enables personalization and continuity.

### 3. Action (Tool use)

- Agents act via tools (functions) like check\_availability, get\_weather.
- Tools connect to external systems (APIs, databases).
- The agent decides when and how to use them.

### 4. Execution & reflection

- After acting, the agent evaluates the result.
- Can retry, ask for clarification, or adjust the plan.

## Building Your First Agent

---

Building an agent with the help of **Model Studio Assistant API** only takes four steps:

```

graph LR
    subgraph agent [Agent Creation Workflow]
        A[Define goal] --> B[Create custom function]
        B --> C[Integrate function]
    end

```

```

    C --> D[Test agent]
end

```

## Step 1: Define your agent's goal

What makes agents so attractive is that each agent is an **expert**. That means each agent is a **specialist** with a singular, clearly defined goal. In our case, we're going to extend the functionality of **TaskFriend** to help with trip planning (who says task management is only limited to work?)

```

graph TD
    subgraph Intent[Intent Classification]
        direction LR
        A["User Query"] --> C{"Knows  
Destination?"}
        C -->|No| E["Travel Advisor Agent"]
    end
    C -->|Yes| D["Planner Agent"]
    E --> D
    subgraph Planner
        direction LR
        D --> H1["Calendar agent"]
        D --> H2["Weather agent"]
        D --> Hx["..."]
        H1 --> I["Summary Agent"]
        H2 --> I
        Hx --> I
        I --> J["Final Trip Plan"]
    end
end

```

But first - let's start small. Before planning a trip, the most important part is to make sure that your schedule is free. So we'll start by creating the **calendar\_agent**!

## Step 2: Create a custom function

Building an agent from scratch is no easy feat. That's why Alibaba Cloud provides the **Assistant API** through **Model Studio** to help simplify the process. However, before we dive into **Assistant API**, we're going to write a custom function that will perform the duties of our **calendar\_agent**.

So what do we need for our **calendar\_agent** to function?

- Needs access to the user's calendar
- Needs to be aware of current and upcoming dates
- It needs to be able to check for the user's availability

In this example, while we are using **tzlocal** to obtain the system time, we're going to simulate the dates on which we are not free.

```

from datetime import datetime, timedelta
from tzlocal import get_localzone

def check_availability(date_range="upcoming weekend"):
    """
    Check user's calendar availability using the machine's local timezone.
    """
    # Get the system's local timezone automatically
    user_tz = get_localzone()
    now = datetime.now(user_tz)

    # Calculate upcoming Saturday and Sunday
    days_ahead = 5 - now.weekday() # Saturday is weekday 5
    if days_ahead <= 0:
        days_ahead += 7
    saturday = now.date() + timedelta(days=days_ahead)
    sunday = saturday + timedelta(days=1)

    # Simulated busy check
    busy_dates = [
        datetime(2025, 7, 4).date(), # Example holiday
    ]

    available_days = []
    busy_events = []

    for day in [saturday, sunday]:
        if day in busy_dates:
            busy_events.append(f"Blocked: Holiday on {day}")
        else:
            available_days.append(day.strftime("%A, %B %d"))

    return {
        "available_days": available_days,
        "busy_events": busy_events,
        "checked_range": {
            "start": saturday.isoformat(),
            "end": sunday.isoformat()
        },
        "timezone": str(user_tz)
    }

```

### Step 3: Integrate function into agent

Now that we've got our function ready to go, it's time to wrap it up in **Assistant API**. To do this, we first have to describe our function to the assistant so that it can call our function properly.

This is done by defining a **tool**, in this case the **calendar\_tool**. It follows a strict schema imposed by **Assistant API**, which we will dive into at the end of the chapter.

```

from dashscope import Assistants, Threads, Messages, Runs
import json

# Define the tool schema
calendar_tool = {
    "type": "function",
    "function": {
        "name": "check_availability",
        "description": "Check if the user is free on upcoming weekend days. Returns available days and any busy events.",
        "parameters": {
            "type": "object",
            "properties": {
                "date_range": {
                    "type": "string",
                    "description": "The date range to check, e.g. 'upcoming weekend', 'next Saturday'"
                }
            },
            "required": ["date_range"]
        }
    }
}

```

Once we've defined the tool, we can use the `Assistants.create` function to integrate our custom code into an agent application.

```

# Create the agent
calendar_agent = Assistants.create(
    model="qwen-plus",
    name="Calendar Checker",
    description="An AI agent that checks user's calendar availability for trip planning.",
    instructions="You are a calendar assistant. Use the check_availability function to determine if the user is free on weekend days.",
    tools=[calendar_tool]
)

print(f"✅ {calendar_agent.name} created with ID: {calendar_agent.id}")

```

And finally, to obtain the response from our **agent** (or *assistant*) through **Assistant API**, we need to define a function that can handle the `thread`, `messages`, and `runs` parameters that they rely on:

```

def get_agent_response(assistant, message=''):
    thread = Threads.create()
    Messages.create(thread.id, content=message)
    run = Runs.create(thread.id, assistant_id=assistant.id)
    run_status = Runs.wait(run.id, thread_id=thread.id)

```

```

    # Handle tool calls
    if run_status.required_action:
        tool_call =
run_status.required_action.submit_tool_outputs.tool_calls[0]
        func_name = tool_call.function.name
        args = json.loads(tool_call.function.arguments)

    # Execute the correct function
    if func_name == "suggest_destinations":
        result = suggest_destinations(args['query'])
    elif func_name == "get_weather_forecast":
        result = get_weather_forecast(args['location'])
    elif func_name == "check_availability":
        result = check_availability(args.get('date_range', 'upcoming
weekend'))
    else:
        result = {"error": "Unknown function"}

    # Submit tool output back to the agent
    Runs.submit_tool_outputs(
        run.id,
        thread_id=thread.id,
        tool_outputs=[{"tool_call_id": tool_call.id, "output":
json.dumps(result)}]
    )
    run_status = Runs.wait(run.id, thread_id=thread.id)

    # Get final response
    msgs = Messages.list(thread.id)
    return msgs['data'][0]['content'][0]['text']['value']

```

## Step 4: Test agent

Finally, it's time to test our agent!

```

response = get_agent_response(
    calendar_agent,
    "I'm planning a road trip for next weekend."
)

print(response)

```

Everything seems to work just fine! Let's move on to the next part.

## Intent Classification: Understanding What the User Wants

So you've built a calendar checker - that works fine when the user wants to check whether they're free to take a day or two off. However, sometimes users have spur of the moments in that they feel like going on a trip, but have not thought about where. Now, we need our agent application is at a crossroads. How can it tell whether to hand off a query to `travel_advisor` (to help decide on where to go) or `trip_planner` (to help plan for the destination)?

To solve this, we can build a simple intent classifier, where the rules are:

- If the user has specified an exact location, send off to `trip_planner`
- If the user has not specified any location, send to `travel_advisor`

We also added some few-shot examples to help the model understand how to handle complex queries like:

What are good weekend trips from Singapore?

## Building an intent router

The intent router does exactly what it's name says:

It routes requests (or in this case, queries) to different agents for processing based on the perceived intent.

```
def classify_trip_intent(question: str) -> str:
    prompt = """
        [Role]
        You are a travel intent classifier. Your task is to determine
        whether the user already knows their travel destination or is still
        exploring options.

        [Task]
        Classify the user's input into one of the following two
        categories:
        1. knows_destination
        2. unsure_destination

        [Rules]
        - Output ONLY the category name. Do not include any explanation or
        formatting.
        - If the user mentions a specific place (e.g., Lake Tahoe,
        Singapore, Napa, Beijing), output: knows_destination
        - If the user expresses uncertainty, curiosity, or exploration
        (e.g., "I want to get away", "somewhere scenic"), output:
        unsure_destination
        - Do not consider conversation history. Classify based solely on
        the current input.

        [Few-shot examples]
        Example 1: User input: "I want to get away this weekend but don't
        know where."
        Classification: unsure_destination

        Example 2: User input: "Plan a trip to Lake Tahoe."
    """
```

```

Classification: knows_destination

Example 3: User input: "Somewhere with mountains and hiking?"
Classification: unsure_destination

Example 4: User input: "Can we go to Yosemite this weekend?"
Classification: knows_destination

Example 5: User input: "What are good weekend trips from
Singapore?"
Classification: unsure_destination

[User input]
Please classify the following input:
""""

full_prompt = prompt + f'"{question}"\nClassification: '
response = get_qwen_stream_response(full_prompt)

# Clean and normalize the response
cleaned = response.strip().strip('\"').strip('\"').strip()

if "knows" in cleaned.lower():
    return "knows_destination"
elif "unsure" in cleaned.lower():
    return "unsure_destination"
else:
    return "unsure_destination"

```

## Testing our intent router

Now that we've build our router, let's test whether it can clearly identify what we need it to do:

- When the user knows their destination, route to `trip_planner`
- When the user is still not sure where to go, route to `travel_advisor`

```

def route_trip_query(query: str):
    intent = classify_trip_intent(query)
    user_tz = get_localzone()
    now = datetime.now(user_tz)
    print(f"\n\n🔍 Identified intent: {intent}\n")

    # Define agent behaviors (simulated here, but can be real agents
    later)
    def travel_advisor():
        print("🧠 [travel_advisor] Thinking of getaway options...\n")
        return get_qwen_stream_response(f'""
            Suggest 3 getaway locations based on {query}.
            If the user does not specify a general location, use the
            user's timezone: {user_tz} \n
            to suggest 3 locations in the same timezone.

```

```

        Be short and concise.
        """
    )

    def trip_planner():
        print("🧠 [planner_agent] Starting full trip planning
workflow...\n")
        # Simulate multi-agent execution
        print("📅 Checking calendar...")
        print("☀️ Checking weather for your destination...")
        print("🚗 Getting travel time...")
        return get_qwen_stream_response(f"""
            Provide a plan to the destination stated in {query}.
            Use the following format:
            * Calendar: Free next weekend.
            * Weather: Return average weather condition at destination
based on \n
            average weather conditions over the years based on {now}.
            * Travel options: Return travel options and estimated
duration \n
            based on where the user is ({user_tz}).
            Be brief and concise.
            """)
    )

    # Route based on intent
    if intent == "unsure_destination":
        print("✉️ Routing to travel_agent...\n")
        return travel_advisor()
    elif intent == "knows_destination":
        print("✉️ Routing to planner_agent...\n")
        return trip_planner()
    else:
        return "I didn't understand your request. Want help planning a
trip?"

```

```

# Test 1: User unsure
user_unsure="I want to get away this weekend but don't know where."
print("\n" + "=" * 50)
print(f"Query: {user_unsure}")
print("=" * 50)
route_trip_query(user_unsure)

# Test 2: User knows destination
user_knows="Plan a trip to Lake Tahoe."
print("\n\n" + "=" * 50)
print(f"Query: {user_knows}")
print("=" * 50)
route_trip_query(user_knows)

# Test 3: User knows a rough destination

```

```
user_kinda_knows="I've never been to France before, what are good spots to see?"
print("\n\n" + "=" * 50)
print(f"Query: {user_kinda_knows}")
print("=" * 50)
route_trip_query(user_kinda_knows)
```

Great, now that we know our intent router works, our application just got a little bit smarter. This is a great example of a key juncture in **multi-agent workflows**, where we are able to control the tools that our application uses to solve the challenges users are facing.

In short, what you've built so far is not just a chatbot - but an application that makes decisions and adapts its behavior to match the type of task at hand. There are several benefits to such workflows:

- You don't need to cover all cases through a single system prompt or prompt template. Each agent can have their own directive.
- You'll be able to create dedicated tools to handle dedicated functions:
  - Dedicated tools are more specialized, leading to better outcomes
  - Dedicated tools may use less resources, cutting down on resource use

Alright, now that we've mastered routing, let's go ahead and build our **multi-agent workflow**!

## Multi-agent Workflows: An Orchestrated Symphony

---

### Planner agent

The planner agent is arguably the most important component of the multi-agent workflow. It evaluates the user's query and selects the best tools for the job.

So let's begin by creating a simple agent for it, and test it out:

```
planner_agent = Assistants.create(
    model="qwen-plus",
    name="Trip Strategist",
    description="Plans getaways by coordinating multiple agents at its disposal."
)
print(f"✅ Planner Agent created: {planner_agent.name}")

print(get_agent_response(planner_agent, "I want to plan a trip to Disneyland"))
```

Now, we want to give the `planner_agent` instructions on how to handle itself:

```
planner_agent = Assistants.update(
    planner_agent.id,
```

```

        instructions="""
            You are TaskFriend, a personal trip planning assistant.
            You have access to these agents:
            - calendar_agent
            - weather_agent
            - maps_agent
            - travel_agent

            Rules:
            1. If the user doesn't know where to go, use travel_agent
FIRST.
            2. Once a destination is chosen, plan the trip in this order:
               - calendar_agent (check availability)
               - weather_agent (check conditions)
            3. Output ONLY a JSON list of agent names in execution order.
               Example: ["calendar_agent", "weather_agent"]
            4. Do not include any explanations.
        """
    )

    print(f"✅ Updated Planner Agent instructions: {planner_agent.name}")

```

```

test_queries = [
    "I want to get away this weekend but don't know where.",
    "Plan a trip to Lake Tahoe.",
    "I've never been to France before, what are good spots to see?"
]

for query in test_queries:
    print(f"\n🔍 Query: {query}")
    plan = get_agent_response(planner_agent, query)
    print(f"📋 Plan: {plan}")

```

## Tooling agents

Now that our Planner is up and running, all that's left for us is to create the rest of the tools (agents) for the workflow.

### travel\_agent

First up, let's create our `travel_agent` agent. This agent should be able to recommend nearby getaways by:

- Inferring from the location stated in the query.
- Extract location data based on your timezone to recommend nearby locations.

```

def extract_location_with_llm(query: str) -> str:
    """
    Use an LLM to extract the destination or origin from a natural

```

```

language query.
    Returns 'nearby' if no specific location is mentioned.
    """
    prompt = f"""
        You are a location extraction assistant. Your task is to identify
        any travel destination, city, region, or point of interest mentioned in
        the user's query.
        - If a specific location is mentioned (e.g., 'London', 'near
        Paris', 'from Tokyo'), return just the location name.
        - If no specific location is mentioned (e.g., 'near me',
        'somewhere scenic', 'I want to get away'), return "nearby".
        - Do NOT include any explanations or formatting.
        - Output ONLY the location or "nearby".

        User Query: "{query}"

        Location:
    """

    # Use your existing LLM interface
    response = get_qwen_stream_response(prompt) # Uses non-streaming call

    # Clean and normalize the response
    cleaned = response.strip().strip(' ').strip('"').strip()

    # Fallback if the model returns something irrelevant
    if not cleaned or cleaned.lower() in ["none", "unknown", "no
location"]:
        return "nearby"

    return cleaned

def suggest_destinations(query: str):
    """
    Suggest getaway destinations based on the user's query.
    Uses LLM to extract location and generate suggestions.
    """
    # Get user's current timezone
    user_tz = get_localzone()

    # Extract location using LLM (no regex)
    extracted_location = extract_location_with_llm(query)

    # Determine origin context
    if extracted_location.lower() != "nearby":
        origin_context = extracted_location
    else:
        origin_context = f"within the {user_tz} timezone"

    # Generate suggestions using LLM
    prompt = f"""
        Suggest 2 weekend getaway destinations near {origin_context}
        within the same timezone ± 1 hour.
        Suggest 1 additional getaway destination further away (can be in a

```

```

different country or region).
    For each destination, include:
    - Name (e.g., a well-known natural or cultural site)
    - Travel time (e.g., "3h 15m drive" or "2h flight")
    - Why it's a good fit for a weekend trip

Respond in JSON format:
{{
  "origin": "{origin_context}",
  "recommended_destinations": [
    {{
      "name": "...",
      "travel_time": "...",
      "reason": "..."
    }}
  ]
}}
"""

response = get_qwen_stream_response(prompt)
try:
    # Parse the response as JSON
    import json
    result = json.loads(response)
    return result
except json.JSONDecodeError as e:
    # If JSON parsing fails, return a clean error structure
    return {
        "error": f"Invalid JSON response: {str(e)}",
        "origin": origin_context,
        "recommended_destinations": []
    }
except Exception as e:
    return {
        "error": f"Could not parse LLM response: {str(e)}",
        "origin": origin_context,
        "recommended_destinations": []
    }

```

```

# Define the tool
travel_tool = {
    "type": "function",
    "function": {
        "name": "suggest_destinations",
        "description": "Suggest weekend getaway destinations based on user query. Uses user's timezone if no location is specified.",
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",

```

```

        "description": "The user's natural language request,
e.g., 'I want to get away this weekend' or 'Plan a trip near London'"
    },
    "required": ["query"]
}
}
}

```

```

travel_agent = Assistants.create(
    model="qwen-plus",
    name="Travel Advisor",
    description="An AI agent that suggests weekend getaway destinations
based on user intent and location.",
    instructions="""
        You are TaskFriend, a friendly and proactive travel advisor.
        Your job is to help users discover weekend getaway options when
they're unsure of where to go.

        You have access to a tool called `suggest_destinations` that can:
        - Extract the location from the user's query (e.g., "near London")
        - If no location is mentioned, use the user's current timezone as
the origin
        - Suggest 2 nearby destinations (within ±1 hour timezone)
        - Suggest 1 further destination
        - Return structured JSON with name, travel time, and reason

        Rules:
        1. Always use the `suggest_destinations` tool when the user is
exploring or unsure.
        2. Do not make up destinations – rely on the tool.
        3. Present the results clearly and conversationally.
        4. End with: "Which one sounds best, or should I suggest more?"
    """,
    tools=[travel_tool]
)

print(f"✅ {travel_agent.name} created with ID: {travel_agent.id}")

```

### weather\_agent: With real-time weather API integration

For our `weather_agent`, we'll be using the open-sourced **open meteo** API. This API is free for non-commercial use, and does not require an API key. This works perfectly for our use case!

```

import requests
import json

def get_weather_forecast(location: str):
    # 1. Geocode using Open-Meteo's geocoding API (no key needed)

```

```

geo_url = "https://geocoding-api.open-meteo.com/v1/search"
geo_params = {"name": location, "count": 1}
geo_response = requests.get(geo_url, params=geo_params).json()

if not geo_response.get("results"):
    return {"error": "Location not found"}

result = geo_response["results"][0]
lat, lon = result["latitude"], result["longitude"]

# 2. Get weather (no API key needed)
weather_url = "https://api.open-meteo.com/v1/forecast"
weather_params = {
    "latitude": lat,
    "longitude": lon,
    "current_weather": True,
    "daily":
"temperature_2m_max,temperature_2m_min,precipitation_sum,weathercode",
    "forecast_days": 3,
    "timezone": "auto"
}

weather_response = requests.get(weather_url, params=weather_params)
if weather_response.status_code != 200:
    return {"error": "Failed to fetch weather data"}

data = weather_response.json()
return {
    "location": f"{result['name']}, {result.get('country', '')}",
    "current": data["current_weather"],
    "daily": data["daily"]
}

```

```

weather_tool = {
    "type": "function",
    "function": {
        "name": "get_weather_forecast",
        "description": "Get weather forecast for a destination. Always use
this when weather is relevant.",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {
                    "type": "string",
                    "description": "The destination city or region (e.g.,
'Lake Tahoe', 'London')
                }
            },
            "required": ["location"]
        }
    }
}

```

```
}
}
```

```
weather_agent = Assistants.create(
    model="qwen-plus",
    name="Weather Checker",
    description="Fetches weather forecasts for trip destinations using
Open-Meteo API.",
    instructions="""
        You are a weather assistant. Your job is to provide accurate, up-
to-date weather forecasts for travel destinations.
        You have access to the `get_weather_forecast` tool.

        Rules:
        1. ALWAYS use the `get_weather_forecast` tool when the user asks
about weather or when planning a trip.
        2. The input to the tool must be the **destination** (e.g., 'Lake
Tahoe', 'London').
        3. NEVER make up weather data – always use the tool.
        4. Present the forecast clearly: include high/low temps,
conditions, and wind.
        5. Be concise and helpful.
    """,
    tools=[weather_tool]
)

print(f"✅ {weather_agent.name} created with ID: {weather_agent.id}")

response = get_agent_response(
    weather_agent,
    "I want to go to London this weekend, August 23–24"
)

print(response)
```

## Putting it all together: The summary agent

```
summary_agent = Assistants.create(
    model="qwen-plus",
    name="Trip Concierge",
    description="Summarizes information from other agents into a final,
polished trip plan for the user.",
    instructions="""
        You are TaskFriend, a friendly and helpful AI trip concierge.
        Your job is to take the information gathered by other agents and
create a warm, clear, and actionable trip plan for the user.

        Input:
        – The original user query
    """
)
```

- The responses from: `travel_agent`, `calendar_agent`, `weather_agent`, `maps_agent`, etc.

Rules:

1. NEVER include raw JSON or technical details.
2. Present the information in a structured, conversational format.
3. Be encouraging and positive.
4. Include key details: destination, availability, weather, travel time, and any recommendations.
5. End with a helpful closing note.

"""

)

```
print(f"✅ {summary_agent.name} created with ID: {summary_agent.id}")
```

```
import ast
import re

def get_trip_plan_response(query: str):
    """
    End-to-end function to plan a trip.
    1. Uses planner_agent to decide the agent sequence.
    2. Executes each agent in order.
    3. Uses summary_agent to generate the final plan.
    """
    # 1. Get the plan from the planner_agent
    raw_plan = get_agent_response(planner_agent, query)
    print(f"🔍 Raw Planner Output: '{raw_plan}'") # Debug: See what the
    planner actually returns

    # Robustly extract the list from the response
    try:
        # Remove any leading/trailing whitespace and newlines
        cleaned = raw_plan.strip()

        # Use regex to find the first list-like structure
        match = re.search(r'\[.*\]', cleaned)
        if not match:
            raise ValueError("No list found in planner response")

        list_str = match.group(0)
        agent_order = ast.literal_eval(list_str)

        print(f"🗺️ Planner Agent decided on this workflow: {agent_order}")

    except Exception as e:
        print(f"❌ Failed to parse plan: {e}")
        # Fallback: use travel_agent if planning fails
        return get_agent_response(travel_agent, query)

    # 2. Map agent names to objects
```

```
agent_mapper = {
    "travel_agent": travel_agent,
    "calendar_agent": calendar_agent,
    "weather_agent": weather_agent,
}

# 3. Execute each agent and collect their responses
context = f"Original User Query: {query}\n\n"
for agent_name in agent_order:
    agent = agent_mapper[agent_name]
    print(f"✂ Executing: {agent_name}")
    response = get_agent_response(agent, query)
    context += f"{agent_name} Response:\n{response}\n\n"

# 4. Generate the final summary
final_prompt = f"""
Please create a final trip plan based on the following information:

{context}

Remember to be concise, friendly, and focus on the most important
details.
"""
final_response = get_agent_response(summary_agent, final_prompt)

return final_response
```

```
result = get_trip_plan_response("I want go to London this weekend.")

print(result)
```

## Agent Development Platforms

---

Aside from writing code and designing your own agents, there are multiple alternatives to create agentic workflows. These platforms provide visual interfaces, pre-built templates, and managed infrastructure to accelerate the development and deployment of AI agents.

### Dify

[Dify](#) is an open-source platform that allows you to build AI agents through a visual workflow interface. It supports connecting various data sources, defining agent behaviors, and deploying agents to multiple channels. Dify's strength lies in its flexibility and extensibility, making it suitable for developers who want to customize every aspect of their agents.



Dify's visual workflow interface

Source: [dify.ai](https://dify.ai)

**Key features:**

- Visual workflow builder
- Support for multiple LLM providers
- Plugin system for extending functionality
- API for integration with external systems
- Open-source with self-hosting options

Dify Community Edition is also available on Alibaba Cloud Compute Nest in two flavors:

- [Community Edition](#)
- [Enterprise Edition](#)

## Hugging Face Agents

Hugging Face provides a suite of tools for building and deploying AI agents. Their platform integrates with the extensive model hub, allowing you to leverage state-of-the-art models for your agents. Hugging Face Agents are particularly strong in natural language processing tasks and can be easily integrated into existing applications.

**Key features:**

- Access to thousands of pre-trained models
- Easy deployment options
- Integration with popular frameworks
- Strong community support
- Model evaluation tools

## LangChain

LangChain is a framework for developing applications powered by language models. While not a platform per se, it provides the building blocks for creating complex agent systems. LangChain excels at connecting LLMs with external data sources and tools, making it ideal for building agents that need to interact with the real world.

**Key features:**

- Modular components for agent development
- Integration with various data sources and APIs
- Support for multiple LLM providers
- Tools for prompt management and optimization
- Active community and extensive documentation

## Choosing the Right Platform

When selecting a platform for agent development, consider the following factors:

- **Complexity of your use case:** Simple workflows might benefit from visual platforms like Dify, while complex, multi-step processes might require the flexibility of LangChain.

- **Development resources:** If you have a strong development team, open-source frameworks provide more control. For teams with limited technical resources, managed platforms with visual interfaces might be more appropriate.
- **Integration requirements:** Consider how the agent needs to interact with your existing systems and data sources.
- **Scalability needs:** Evaluate the platform's ability to handle your expected load and growth.
- **Budget constraints:** Some platforms offer free tiers or open-source options, while others require subscription fees.

The choice of platform should align with your technical capabilities, project requirements, and long-term goals. Many organizations start with a simpler platform for prototyping and then migrate to a more robust framework as their needs evolve.

## What's Next?

---

---

### Quiz yourself!

► **1. What is the primary role of a planner agent in a multi-agent system?**

- A) To generate the final natural language response
- B) To decide which tools or sub-agents to invoke and in what order
- C) To store long-term memory of user preferences
- D) To perform vector similarity search on documents

**View answer →**

✓ **Correct answer:** B) To decide which tools or sub-agents to invoke and in what order

 **Explanation :**

- The planner agent is responsible for goal decomposition and orchestration — it determines the sequence of actions (e.g., check calendar → check weather → suggest destinations) needed to fulfill the user's request.

### Takeaways

- **Why build agentic systems?**
  - **RAG answers questions — agents solve problems.**
  - **Real-world tasks require multiple steps, tools, and decisions** — agents orchestrate them.
  - **Agents decompose complexity** into manageable subtasks (e.g., check calendar, fetch weather).
  - **They enable autonomous, goal-driven behavior** — the next level of AI assistance.
- **Agent orchestration**

- **Use a planner agent** to coordinate sub-agents (calendar, weather, etc.).
  - **Implement intent classification** to route queries to the right agent.
  - **Chain agents together**: one agent's output becomes another's input.
  - **Design for modularity** — each agent should do one thing well.
  - **Orchestration ensures coherence** across multi-step workflows.
- 
- **Tool integration**
    - **Agents use tools to interact with external systems** (APIs, calendars, databases).
    - **Wrap functions as tools** (e.g., `function_tool`) so agents can call them.
    - **Always include error handling** — agents should fail gracefully.
    - **Tools extend agent capabilities** beyond what's in context.
    - **Good tool design is key** — clear inputs, structured outputs.