Task 2 (Pydantic AI)

What is Pydantic AI?

Pydantic AI is a Python agent framework that brings structure and type safety to LLM applications. Unlike basic LLM integrations that return raw text, Pydantic AI treats AI interactions as structured conversations with validated inputs and outputs, making it easier to build reliable applications.

Key features of Pydantic AI

- **Structured output validation**—Your AI responses automatically conform to Pydantic models, preventing parsing errors and ensuring data consistency
- **Function tools**—LLMs can call your Python functions during conversations, giving them access to real data and computations
- **Type safety**—Full typing support means fewer runtime errors and better IDE assistance when building complex AI workflows
- **System prompts**—Define clear instructions for your AI agents that stay consistent across all interactions
- **Dependency injection**—Share context, database connections, and user preferences across all agent components cleanly
- **Multiple execution modes**—Run agents synchronously, asynchronously, or stream responses in real-time based on your needs
- **Agent reusability—**Create agents once and reuse them throughout your application, similar to FastAPI routers

Difference between Pydantic AI vs LangChain

Feature	Pydantic AI	LangChain
Focus	Structured output + typed agents using Pydantic models	Orchestration of LLMs + chains + memory + tools
Output validation	Native: defines `output_type=` or JSON schema enforcement	Typically manual: you build parsing/validation yourself
Tool/function support	Works well with OpenAI/Anthropic; needs adaptation for free models like GROQ	Full tool support, multi-agent systems, memory layers, workflows
Use-case strength	Building typed, production-ready single agents with guaranteed	Constructing workflows, pipelines, multi-step agents, chains of

Feature	Pydantic AI	LangChain
	structure	prompts
Ideal scenario	You need strict data types, validation, deterministic output	You need flexible pipelines, many tools, multi-stage reasoning

Code examples

creating a basic Settings class to import the api key from env file and create a sales agent from it

This section demonstrates how to centralize and securely manage your API keys using a BaseSettings configuration class.
Environment variables (loaded from .env) are mapped to typed attributes so that your application avoids hard-coding sensitive credentials.
Once the settings are loaded, environment variables are exported so pydantic_ai and other libraries can access the keys.
A Groq model (llama-3.1-8b-instant) is then used to instantiate a simple sales analysis agent, which can answer questions in plain language.

Key idea:
Environment configuration → Agent creation → First query to the model

```python
from pydantic_ai import Agent
from pydantic_settings import BaseSettings
from pydantic import Field
import os
from dotenv import load_dotenv



load_dotenv()
class Settings(BaseSettings):
    groq_api_key: str = Field(alias="GROQ_API_KEY")
    serpapi_key: str = Field(alias="SERPAPI_KEY")
    openai_api_key: str = Field(alias="OPENAI_API_KEY")
    openrouter_api_key: str = Field(alias="OPENROUTER_API_KEY")
    openrouter_api_base: str = Field(alias="OPENROUTER_API_BASE")
    model_config = {"env_file": ".env", "extra": "ignore"}



settings = Settings()
```

```
os.environ["GROQ_API_KEY"] = settings.groq_api_key
os.environ["OPENAI_API_KEY"] = settings.open_api_key
os.environ["SERPAPI_KEY"]= settings.serpapi_key
os.environ["OPENROUTER_API_KEY"]= settings.openrouter_api_key
os.environ["OPENROUTER_API_BASE"]= settings.openrouter_api_base

sales_agent = Agent(
    model="groq:llama-3.1-8b-instant",
    system_prompt="You are a data analyst."
)

result = await sales_agent.run("What metrics should I track?")

print(result.output)
```

creating agents with differnet models and tweaking the model settings

This section shows how you can experiment with different behavior styles by adjusting **model hyperparameters**, such as temperature and maximum response length.
By lowering temperature, responses become more deterministic and professional.
By adjusting `max_tokens`, you control how long the generated output can be.
Even though the model remains the same (`groq:llama-3.1-8b-instant`), modifying these settings allows you to simulate a range of "personalities" or levels of expertise (e.g., intern vs. senior analyst voice).

Key idea:
The same Groq model → different style and tone via `model_settings`.

```
groq_intern_agent=Agent(
    "groq:llama-3.1-8b-instant",
    system_prompt="You are a data science intern at an AI Health startup.
Provide clear, actionable insights based on the data provided",
model_settings={'temperature': 0.3, 'max_tokens':500}
)

question=await groq_intern_agent.run("What are some of the data formats for
health data in India? Answer in 2-3 lines")

print(question.output)
```

using built in search tool with our sales agent

Since **Groq models do not support built-in tool calling** (unlike OpenAI and Anthropic), we define our own function to fetch web data using **SerpAPI**.
This function acts as a **manually invoked external tool** to retrieve real-time information from the internet, while the Groq model performs reasoning and summarization based on the returned data.
The model is instructed to *conceptually "use" the tool*, but we trigger the tool ourselves, rather than relying on automatic model-driven function calls.

Key idea:
Groq = reasoning + summarization
SerpAPI = real-time information source

```python
import os
from pydantic_ai import Agent
from serpapi import GoogleSearch

def web_search_tool(query: str) -> dict:
    params={
        "engine": "google",
        "q" : query,
        "api_key": os.environ["SERPAPI_KEY"]
    }
    result=GoogleSearch(params).get_dict()
    return result.get("organic_results")

market_research_agent=Agent(
    'groq:llama-3.1-8b-instant',
    tools=[web_search_tool],
    system_prompt='You are a market analyst. Use the web search tool to find
the current information.'
)

information_result= await market_research_agent.run("What is the current stock
price of CMS Info Systems Ltd?")

print(os.getenv("SERPAPI_KEY"))
print(information_result.output)
```

using structured outputs (output_type feature available only in anthropic and openai models)

Unlike OpenAI + Anthropic models, Groq does *not* support the automatic `output_type=` structured response feature.
Therefore, we instruct the model to return data in **strict JSON format**, and then parse the result

into a **Pydantic model** ourselves.
This pattern ensures your AI agent returns reliable, typed, validated data that can be safely used in downstream logic.

Key idea:
Model returns JSON text → Pydantic validates → Safe structured output.

```python
import json
from pydantic import BaseModel, ValidationError
from pydantic_ai import Agent

class SalesInsight(BaseModel):
    total_revenue: float
    best_performing_region: str
    worst_performing_region: str
    recommendation: str

# Groq model agent (no output_type)
analysis_agent = Agent(
    "groq:llama-3.1-8b-instant",
    system_prompt="""
You are a sales analyst.
Return your answer *only* in the following JSON format:
{
  "total_revenue": number,
  "best_performing_region": "string",
  "worst_performing_region": "string",
  "recommendation": "string"
}
Do not add explanations, markdown, sentences, or commentary.
Only valid JSON.
"""
)

# User prompt
text_data = """
Q1 2025 sales data:

- North: $385,000 revenue (470 units)

- South: $256,000 revenue (334 units)

- East: $391,500 revenue (381 units)

- West:  $395,500 revenue (457 units)
```

```
Analyze this data and provide insights.

"""
# Run Groq model
response = await analysis_agent.run(text_data)
raw_json = response.output.strip()

# Parse into Pydantic model
try:
    data = SalesInsight(**json.loads(raw_json))
except (json.JSONDecodeError, ValidationError) as e:
    raise ValueError(f"Model returned invalid JSON:\n{raw_json}") from e

# Output

print(f"Total Revenue: ${data.total_revenue:,.0f}")
print(f"Best Region: {data.best_performing_region}")
print(f"Worst Region: {data.worst_performing_region}")
print(f"Recommendation: {data.recommendation}")
```

accessing the message history

Each call to the agent stores the interaction as part of a **conversation history object**.
This history allows you to examine prior exchanges, debug reasoning, or reuse the state in follow-up queries.
The agent can report two useful views:

- **All messages so far** (full conversation memory)
- **Only messages generated during the latest call** (incremental history)

Key idea:
Conversation is stateful. The agent tracks and returns message history.

```
from pydantic_ai import Agent
sales_agent = Agent(
    'groq:llama-3.1-8b-instant',
    system_prompt="You are a sales analyst. Provide clear, concise analysis."
)

# First question
result1 = await sales_agent.run("What are the main KPIs I should track for Q1 2025 sales?")

print(result1.output)
# Conversation history
```

```
all_messages = result1.all_messages()
print(f"Total messages in conversation: {len(all_messages)}")
new_messages = result1.new_messages()
print(f"New messages from this run: {len(new_messages)}")
```

Continuing conversations with message history (to build on previous conversations, pass the message history to your next agent run)

You can extend a conversation by passing previous messages into the next `.run()` call. This allows the model to remember context across turns without requiring you to repeat previous information manually.
It simulates a natural dialogue, enabling layered reasoning such as:

- Learning KPIs → describing calculations → deciding priorities

Key idea:
Conversation context is forwarded manually using `message_history`.

```
#Continuing conversations with message history
from pydantic_ai import Agent

# Create the agent using Groq (FREE)
sales_agent = Agent(
    'groq:llama-3.1-8b-instant',
    system_prompt="You are a sales analyst. Provide clear, concise analysis."
)

# First question
result1 = await sales_agent.run("What are the main KPIs I should track for Q1
2025 sales?")
print(result1.output)  # ✅ use .text

# Continue the conversation
result2 = await sales_agent.run(
    "How should I calculate conversion rates for each of those KPIs?",
    message_history=result1.all_messages()  # ✅ pass conversation history
)
print(result2.output)

print(f"Full conversation length: {len(result2.all_messages())}")

# Follow-up question using total conversation context
result3 = await sales_agent.run(
    "Which of these metrics would be most important for a monthly executive
report?",
    message_history=result2.all_messages()
)
print(result3.output)
```

Storing and loading messages to JSON

Conversations can be **exported as JSON** (using `all_messages_json()`), saved to disk, and later **reloaded** to resume the conversation from the exact point where it left off.
This is essential for:

- Long-term project agents
- Multi-session workflows
- Persisting agent memory between notebook runs

Key idea:
Agent memory can be saved → restored → reused.

```
import json
from pydantic_ai import Agent

# Create Groq Agent
sales_agent = Agent(
    'groq:llama-3.1-8b-instant',
    system_prompt="You are a sales analyst. Provide clear, concise analysis."
)

# Conversation Steps
result1 =await sales_agent.run("What are the main KPIs I should track for Q1
2025 sales?")
result2 = await sales_agent.run(
    "How should I calculate conversion rates for each of those KPIs?",
    message_history=result1.all_messages()
)
result3 =await sales_agent.run(
    "Which of these metrics would be most important for a monthly executive
report?",
    message_history=result2.all_messages()
)

# ✅ Use .text instead of .output for display
print(result3.output)
```

```python
# ✅ Export full conversation history (works identical for Groq)
conversation_json = result3.all_messages_json()

# Save to file
with open('sales_analysis_conversation.json', 'wb') as f:
    f.write(conversation_json)

print("✅ Conversation saved to sales_analysis_conversation.json")

# Load conversation back later
with open('sales_analysis_conversation.json', 'rb') as f:
    loaded_conversation = f.read()

print(f"Loaded conversation size: {len(loaded_conversation)} bytes")
```

ReAct Framework: Simulating Tool Calling in LangChain

What is ReAct?

ReAct stands for **Reasoning + Acting**:

1. **Reasoning**: The model processes the user input and determines if it needs to call an external tool (like a web search).
2. **Acting**: The model generates an "action" (e.g., `Action: search("CMS stock price")`).
3. **Execution**: The framework (like **LangChain**) intercepts the action and calls the actual tool (e.g., `SerpAPI` for web search).
4. **Feedback**: The result from the tool is fed back into the model as an **observation**, and the reasoning continues.

ReAct in LangChain

LangChain uses **ReAct** to allow models to simulate tool use without actually having the model invoke tools. The model outputs an **action** (e.g., `Action: search`), but **LangChain** or another system handles the tool execution and returns the result to the model.

Key Difference with Groq Models

- **LangChain**: The model *appears* to know which tools to call (via ReAct), but in reality, LangChain is **intercepting the model's text** and running the tools externally.
- **Groq**: Does not have built-in tool invocation. You must **manually call external tools** based on the model's reasoning and provide the result for further analysis.

ReAct Workflow Example:

1. **User Query**: "What is the stock price of CMS Info Systems?"
2. **Model Reasoning**: "I need to search for the stock price."
3. **Action**: `Action: search("CMS Info Systems stock price")`
4. **Tool Execution**: LangChain runs the search and returns the result.
5. **Model Feedback**: The model receives the stock price, completes the reasoning, and outputs the answer.

Comparison: LangChain vs Groq

Feature	LangChain (ReAct)	Groq (via `pydantic-ai`)
Reasoning + Acting	✅ Model reasons and calls tools	❌ Groq only reasons, manual tool invocation required
Automatic tool calling	✅ Tools called automatically by LangChain	❌ You manually trigger tools and pass results back
Tool execution	✅ LangChain runs tools for the model	❌ You run tools yourself

Conclusion

- **LangChain + ReAct** simulates tool calling with **automatic action** based on model reasoning.
- **Groq models** can reason but do not natively support automatic tool invocation; instead, you **manually run tools** and provide the results for further reasoning.