PydanticAI

Agent Framework / shim to use Pydantic with LLMs

When I first found FastAPI, I got it immediately. I was excited to find something so innovative and ergonomic built on Pydantic.

Virtually every Agent Framework and LLM library in Python uses Pydantic, but when we began to use LLMs in Pydantic Logfire, I couldn't find anything that gave me the same feeling.

PydanticAI is a Python Agent Framework designed to make it less painful to build production grade applications with Generative AI.

Why use PydanticAI

- Built by the team behind Pydantic (the validation layer of the OpenAI SDK, the Anthropic SDK, LangChain, LlamaIndex, AutoGPT, Transformers, CrewAI, Instructor and many more)
- Model-agnostic — currently OpenAI, Gemini, and Groq are supported, Anthropic is coming soon. And there is a simple interface to implement support for other models.
- Type-safe
- Control flow and agent composition is done with vanilla Python, allowing you to make use of the same Python development best practices you'd use in any other (non-AI) project
- Structured response validation with Pydantic
- Streamed responses, including validation of streamed *structured* responses with Pydantic
- Novel, type-safe dependency injection system, useful for testing and eval-driven iterative development
- Logfire integration for debugging and monitoring the performance and general behavior of your LLM-powered application

> **In Beta**
>
> PydanticAI is in early beta, the API is still subject to change and there's a lot more to do. Feedback is very welcome!

Hello World Example

Here's a minimal example of PydanticAI:

hello_world.py

```python
from pydantic_ai import Agent

agent = Agent(  # (1)
    'gemini-1.5-flash',
    system_prompt='Be concise, reply with one sentence.',  # (2)
)

result = agent.run_sync('Where does "hello world" come from?')  # (3)
print(result.data)
"""
The first known use of "hello, world" was in a 1974 textbook about the C programming language.
"""
```

1. Define a very simple agent — here we configure the agent to use Gemini 1.5's Flash model, but you can also set the model when running the agent.
2. Register a static system prompt using a keyword argument to the agent. For more complex dynamically-generated system prompts, see the example below.
3. Run the agent synchronously, conducting a conversation with the LLM. Here the exchange should be very short: PydanticAI will send the system prompt and the user query to the LLM, the model will return a text response.

(This example is complete, it can be run "as is")

Not very interesting yet, but we can easily add "tools", dynamic system prompts, and structured responses to build more powerful agents.

Tools & Dependency Injection Example

Here is a concise example using PydanticAI to build a support agent for a bank:

bank_support.py

```python
from dataclasses import dataclass

from pydantic import BaseModel, Field
from pydantic_ai import Agent, RunContext

from bank_database import DatabaseConn


@dataclass
class SupportDependencies:  # (3)
    customer_id: int
    db: DatabaseConn  # (12)


class SupportResult(BaseModel):  # (13)
    support_advice: str = Field(description='Advice returned to the customer')
    block_card: bool = Field(description="Whether to block the customer's card")
    risk: int = Field(description='Risk level of query', ge=0, le=10)


support_agent = Agent(  # (1)
    'openai:gpt-4o',  # (2)
    deps_type=SupportDependencies,
    result_type=SupportResult,  # (9)
    system_prompt=(  # (4)
        'You are a support agent in our bank, give the '
        'customer support and judge the risk level of their query.'
    ),
)
```

```python
@support_agent.system_prompt  (5)
async def add_customer_name(ctx: RunContext[SupportDependencies]) -> str:
    customer_name = await ctx.deps.db.customer_name(id=ctx.deps.customer_id)
    return f"The customer's name is {customer_name!r}"


@support_agent.tool  (6)
async def customer_balance(
    ctx: RunContext[SupportDependencies], include_pending: bool
) -> float:
    """Returns the customer's current account balance."""  (7)
    return await ctx.deps.db.customer_balance(
        id=ctx.deps.customer_id,
        include_pending=include_pending,
    )


...  (11)


async def main():
    deps = SupportDependencies(customer_id=123, db=DatabaseConn())
    result = await support_agent.run('What is my balance?', deps=deps)  (8)
    print(result.data)  (10)
    """
    support_advice='Hello John, your current account balance, including pending transactions, is $123.45.' block_card=False risk=1
    """

    result = await support_agent.run('I just lost my card!', deps=deps)
    print(result.data)
    """
    support_advice="I'm sorry to hear that, John. We are temporarily blocking your card to prevent unauthorized transactions." block_card=True risk=8
    """
```

1. This agent will act as first-tier support in a bank. Agents are generic in the type of dependencies they accept and the type of result they return. In this case, the support agent has type `Agent[SupportDependencies, SupportResult]`.
2. Here we configure the agent to use OpenAI's GPT-4o model, you can also set the model when running the agent.
3. The `SupportDependencies` dataclass is used to pass data, connections, and logic into the model that will be needed when running system prompt and tool functions. PydanticAI's system of dependency injection provides a type-safe way to customise the behavior of your agents, and can be especially useful when running unit tests and evals.
4. Static system prompts can be registered with the `system_prompt` keyword argument to the agent.
5. Dynamic system prompts can be registered with the `@agent.system_prompt` decorator, and can make use of dependency injection. Dependencies are carried via the `RunContext` argument, which is parameterized with the `deps_type` from above. If the type annotation here is wrong, static type checkers will catch it.
6. `tool` let you register functions which the LLM may call while responding to a user. Again, dependencies are carried via `RunContext`, any other arguments become the tool schema passed to the LLM. Pydantic is used to validate these arguments, and errors are passed back to the LLM so it can retry.
7. The docstring of a tool is also passed to the LLM as the description of the tool. Parameter descriptions are extracted from the docstring and added to the parameter schema sent to the LLM.
8. Run the agent asynchronously, conducting a conversation with the LLM until a final response is reached. Even in this fairly simple case, the agent will exchange multiple messages with the LLM as tools are called to retrieve a result.
9. The response from the agent will, be guaranteed to be a `SupportResult`, if validation fails reflection will mean the agent is prompted to try again.
10. The result will be validated with Pydantic to guarantee it is a `SupportResult`, since the agent is generic, it'll also be typed as a `SupportResult` to aid with static type checking.
11. In a real use case, you'd add more tools and a longer system prompt to the agent to extend the context it's equipped with and support it can provide.
12. This is a simple sketch of a database connection, used to keep the example short and readable. In reality, you'd be connecting to an external database (e.g. PostgreSQL) to get information about customers.
13. This Pydantic model is used to constrain the structured data returned by the agent. From this simple definition, Pydantic builds the JSON Schema that tells the LLM how to return the data, and performs validation to guarantee the data is correct at the end of the run.

Complete `bank_support.py` example

The code included here is incomplete for the sake of brevity (the definition of `DatabaseConn` is missing); you can find the complete `bank_support.py` example here.

Instrumentation with Pydantic Logfire

To understand the flow of the above runs, we can watch the agent in action using Pydantic Logfire.

To do this, we need to set up logfire, and add the following to our code:

bank_support_with_logfire.py

```python
...
from bank_database import DatabaseConn

import logfire
logfire.configure()  (1)
logfire.instrument_asyncpg()  (2)
...
```

1. Configure logfire, this will fail if not project is set up.
2. In our demo, `DatabaseConn` uses `asyncpg` to connect to a PostgreSQL database, so `logfire.instrument_asyncpg()` is used to log the database queries.

That's enough to get the following view of your agent in action:

See Monitoring and Performance to learn more.

Next Steps

To try PydanticAI yourself, follow the instructions in the examples.

Read the docs to learn more about building applications with PydanticAI.

Read the API Reference to understand PydanticAI's interface.