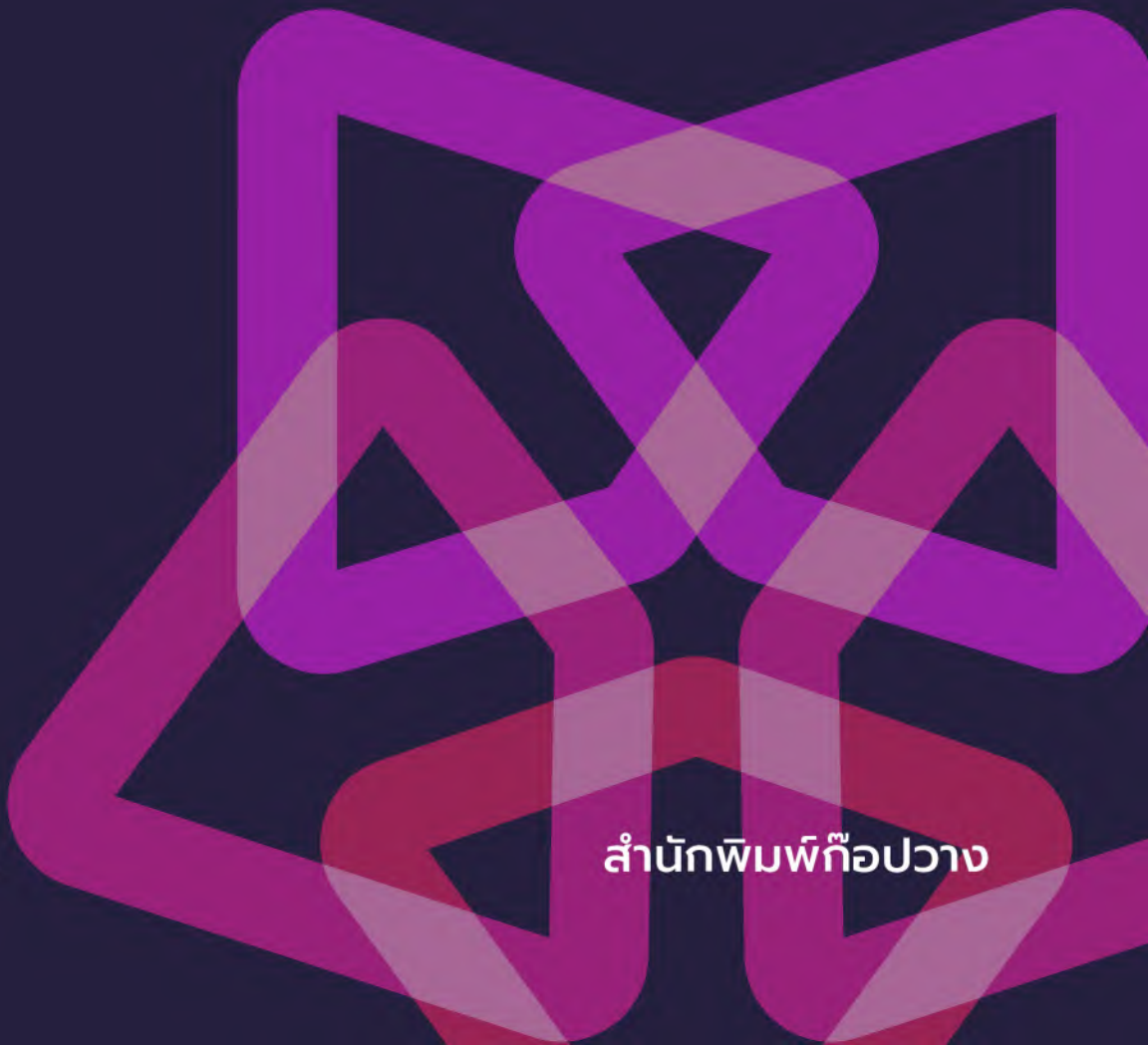
อัปเดตครั้งที่ 3

ปั้น AI Agent ด้วย PydanticAI

เรียนรู้ครบจบในเล่มเดียว สู่การเป็นมืออาชีพ



อนุชิต ชโลธร

สำนักพิมพ์ก๊อปวาง

ปั้น AI Agent ด้วย PydanticAI

เรียนรู้ครบจบในเล่มเดียว สู่การเป็นมืออาชีพ

อนุชิต ชโลธร

(อัปเดตครั้งที่ 3)

คำนำสำนักพิมพ์

ในนามของสำนักพิมพ์ เรามีความยินดีเป็นอย่างยิ่งที่ได้นำเสนอหนังสือ “ปั้น AI Agent ด้วย PydanticAI: เรียนรู้ครบจบในเล่มเดียว สู่การเป็นมืออาชีพ” สู่สายตานักพัฒนาและผู้สนใจในเทคโนโลยีปัญญาประดิษฐ์ทุกท่าน

ปฏิเสธไม่ได้ว่า Generative AI และ Large Language Models (LLMs) คือคลื่นลูกใหม่ที่กำลังเปลี่ยนแปลงโลกเทคโนโลยีในทุกมิติ ความสามารถในการสร้างแอปพลิเคชันที่สามารถ “คิด” และ “สื่อสาร” ได้ใกล้เคียงกับมนุษย์ได้เปิดประตูสู่ความเป็นไปได้ใหม่ๆ อย่างที่ไม่เคยมีมาก่อน อย่างไรก็ตาม การเปลี่ยนศักยภาพอันน่าทึ่งนี้ให้กลายเป็นผลิตภัณฑ์ที่ใช้งานได้จริงและเชื่อถือได้นั้น ยังคงเป็นความท้าทายที่ยิ่งใหญ่

PydanticAI ได้เข้ามามีบทบาทสำคัญในการลดช่องว่างดังกล่าว ด้วยการนำเสนอแนวทางที่สวยงามในการสร้างสะพานเชื่อมระหว่างโลกของ AI ที่ยืดหยุ่นและโลกของซอฟต์แวร์ที่ต้องการความแม่นยำและโครงสร้างที่ชัดเจน

หนังสือเล่มนี้ไม่ได้เป็นเพียงคู่มือการใช้งานไลบรารี แต่เป็นตำราอาหารที่รวบรวม “สูตรสำเร็จ” สำหรับการสร้าง AI Agent ในสถานการณ์ต่างๆ ที่นักพัฒนาต้องเผชิญในโลกแห่งความเป็นจริง ตั้งแต่การดึงข้อมูลพื้นฐาน, การสร้าง Chatbot, การทำงานกับฐานข้อมูลและ API, ไปจนถึงเทคนิคขั้นสูงสำหรับการทดสอบและนำขึ้นระบบโปรดักชัน ด้วยแนวทางการเขียนที่เน้นการลงมือทำและตัวอย่างที่จับต้องได้ เราเชื่อมั่นว่าหนังสือเล่มนี้จะเป็นเครื่องมือสำคัญที่ช่วยให้ท่านสามารถนำ PydanticAI ไปประยุกต์ใช้ได้อย่างรวดเร็วและมั่นใจ

เราหวังว่าหนังสือเล่มนี้จะเป็นส่วนหนึ่งในการขับเคลื่อนวงการพัฒนาซอฟต์แวร์ AI ของไทยให้ก้าวไปข้างหน้า และช่วยให้ท่านสามารถสร้างสรรค์นวัตกรรมที่ยอดเยี่ยมออกมาได้ในที่สุด

ทีมงานสำนักพิมพ์

คำนำ

ในยุคที่ Large Language Models (LLMs) ได้กลายเป็นเทคโนโลยีที่ก้าวล้ำและเข้าถึงได้ง่าย นักพัฒนาทั่วโลกต่างตื่นเต้นกับศักยภาพในการสร้างสรรค์แอปพลิเคชันอัจฉริยะที่สามารถเข้าใจและโต้ตอบกับภาษามนุษย์ได้อย่างเป็นธรรมชาติ แต่ทว่า ท่ามกลางความมหัศจรรย์นี้ เรากลับพบความท้าทายสำคัญประการหนึ่ง นั่นคือ “ความไร้ระเบียบ” ของข้อมูลที่ได้จาก LLMs

LLMs สร้างข้อความที่อ่านลื่นไหลได้อย่างยอดเยี่ยม แต่ผลลัพธ์ที่ได้มักอยู่ในรูปแบบของข้อความดิบ (raw text) ที่ไม่มีโครงสร้างชัดเจน การจะนำข้อมูลเหล่านี้ไปใช้งานต่อในแอปพลิเคชันจริง เช่น การบันทึกลงฐานข้อมูล, การเรียกใช้ API หรือการแสดงผลบน UI จึงกลายเป็นเรื่องที่ซับซ้อนและเต็มไปด้วยโค้ดที่ต้องเขียนขึ้นเพื่อจัดการกับข้อมูลโดยเฉพาะ (boilerplate code)

PydanticAI แก้ปัญหาเรื่องเหล่านี้โดยเฉพาะ โดยใช้ปรัชญาอันเรียบง่ายของ Pydantic ซึ่งเป็นไลบรารีสำหรับการตรวจสอบและจัดการข้อมูลที่ดีที่สุดตัวหนึ่งของ Python เข้ากับ

ความสามารถของ LLMs ผลลัพธ์ที่ได้คือเครื่องมือที่ช่วยให้นักพัฒนาสามารถ “สั่ง” ให้ LLM คืนค่าผลลัพธ์ออกมาเป็น Pydantic Object ที่มีโครงสร้างชัดเจน, ตรวจสอบความถูกต้องได้ และพร้อมใช้งานต่อได้ทันที

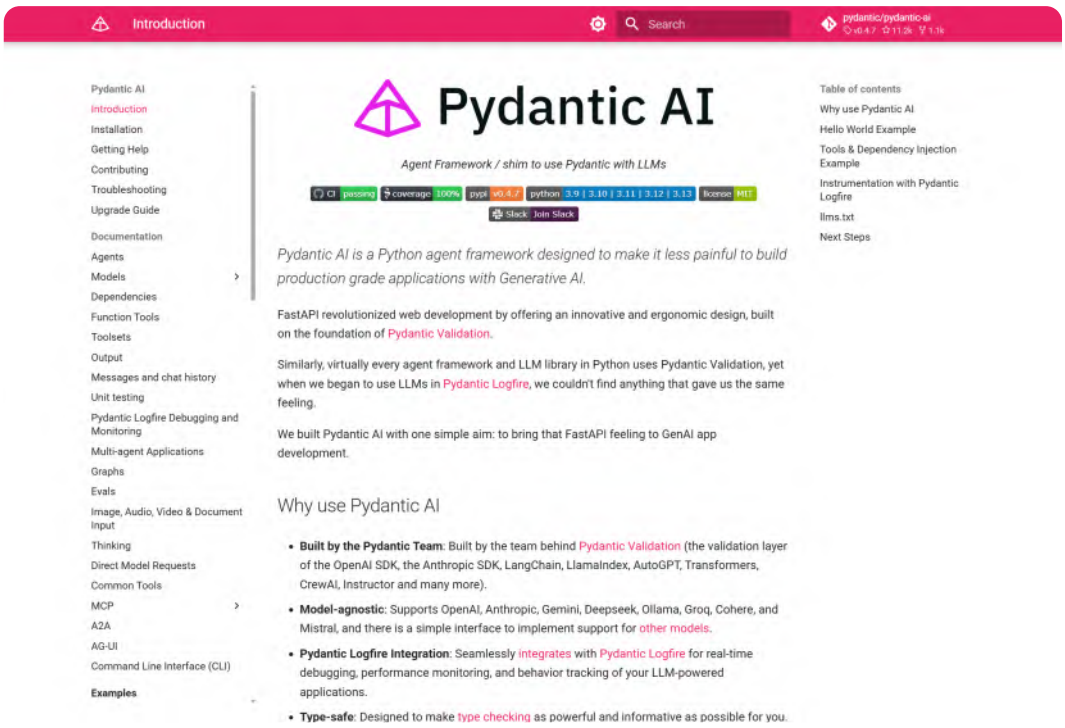
หนังสือ “ปั้น AI Agent ด้วย PydanticAI: เรียนรู้ครบจบในเล่มเดียว สู่การเป็นมืออาชีพ” เล่มนี้ เป็นคู่มือที่จะพาคุณเดินทางตั้งแต่จุดเริ่มต้นของการทำความเข้าใจปัญหา ไปจนถึงการสร้าง AI Agent ที่ซับซ้อนและพร้อมสำหรับนำไปใช้งานจริง (Production-ready) เราจะเปลี่ยน “Prompt” ที่เป็นเพียงไอเดียในหัวของคุณ ให้กลายเป็น “Product” ที่จับต้องได้ ผ่านสูตรสำเร็จที่คัดสรรมาอย่างดี พร้อมตัวอย่างโค้ดที่นำไปใช้ได้จริง และคำอธิบายเบื้องหลังการทำงานที่ชัดเจน

ไม่ว่าคุณจะเป็นนักพัฒนาที่เพิ่งเริ่มต้นกับโลกของ AI หรือเป็นวิศวกรผู้มีประสบการณ์ที่กำลังมองหาเครื่องมือที่จะช่วยให้การทำงานกับ LLMs เป็นเรื่องง่ายและมีประสิทธิภาพมากขึ้น ผมหวังว่าหนังสือเล่มนี้จะเป็นเข็มทิศนำทางให้คุณสามารถปลดปล่อยศักยภาพของ AI และสร้างสรรค์ผลงานที่น่าทึ่งออกมาได้สำเร็จ

สารบัญ

แนะนำ PydanticAI

ถ้า FastAPI คือเฟรมเวิร์กที่ทำให้การสร้าง API เป็นเรื่องสนุก PydanticAI ก็เป็นเครื่องมือสร้าง AI Agent ได้สนุกในแบบเดียวกัน



เว็บไซต์ ai.pydantic.dev

PydanticAI เป็นเฟรมเวิร์กสำหรับสร้าง AI Agent ด้วยภาษา Python ออกแบบมาเพื่อช่วยให้นักพัฒนาสามารถสร้างแอปพลิเคชัน Generative AI ในระดับ Production ได้อย่างรวดเร็ว ถูกต้อง และมีโครงสร้างที่ดี โดยอาศัยจุดแข็งของ

Pydantic ซึ่งเป็นเครื่องมือที่คุ้นเคยกันดีในวงการภาษา Python และถูกใช้อย่างแพร่หลายในไลบรารี AI ชั้นนำ

แม้ว่าทุกวันนี้ไลบรารี LLM ต่างก็ใช้ Pydantic ในการจัดการข้อมูลอยู่เบื้องหลัง แต่กลับยังไม่มีเฟรมเวิร์กตัวไหนที่มอบประสบการณ์การพัฒนาในแบบที่ FastAPI เคยทำได้ นั่นจึงเป็นที่มาของ **PydanticAI**

จุดเด่นของ PydanticAI

- **PydanticAI สร้างโดยทีม Pydantic โดยตรง** PydanticAI พัฒนาขึ้นโดยทีมเดียวกับผู้สร้าง Pydantic ไลบรารีตรวจสอบข้อมูลที่กลายเป็นหัวใจในไลบรารี AI ชั้นนำหลากหลายตัว เช่น OpenAI SDK, LangChain, LlamaIndex, Instructor, CrewAI และอื่นๆ อีกมากมาย
- **รองรับหลายโมเดล** PydanticAI สามารถเชื่อมต่อกับโมเดลยอดนิยมได้หลากหลาย เช่น OpenAI, Anthropic, Gemini, Ollama, Deepseek, Groq, Coher และ Mistral พร้อมระบบอินเตอร์เฟซที่ยืดหยุ่น สามารถขยายให้รองรับโมเดลอื่นๆ ได้ง่าย

- **ทำงานร่วมกับ Pydantic Logfire ได้อย่างไร้รอยต่อ** สามารถเชื่อมต่อกับ Pydantic Logfire เพื่อดูการทำงานของ Agent แบบเรียลไทม์ ตรวจสอบประสิทธิภาพ และพฤติกรรมของ LLM ได้สะดวกสบาย
- **ปลอดภัยด้วยระบบ Type Checking** Agent และ Output ในระบบของ PydanticAI ถูกออกแบบให้รองรับ Type-safe อย่างเต็มรูปแบบ ทำให้โค้ดของคุณสามารถตรวจสอบได้ตั้งแต่ก่อนรันจริง ช่วยลดข้อผิดพลาดในระบบ
- **เขียนได้ง่ายในสไตล์ Python** เฟรมเวิร์กนี้ออกแบบโดยยึดแนวคิด Pythonic ไม่ว่าจะเป็น Control Flow การจัดการ Agent แบบ Composition หรือแม้แต่ Dependency Injection ก็สามารถเขียนในรูปแบบที่คุณคุ้นเคยได้
- **จัดการผลลัพธ์ด้วยโครงสร้างที่แน่นอน** การใช้ Pydantic มาช่วยในการตรวจสอบและจัดรูปแบบผลลัพธ์จากโมเดล LLM ทำให้คุณมั่นใจได้ว่า Response ที่ได้จากโมเดลจะมีความสม่ำเสมอ และสามารถนำไปใช้งานต่อได้ทันที
- **ระบบ Dependency Injection** สามารถส่งข้อมูลหรือบริการต่างๆ ไปยัง Agent ได้ผ่านระบบ Dependency Injection ซึ่งเหมาะอย่างยิ่งสำหรับการทำ Testing, Mocking และการพัฒนาแบบ Iterative

- **รองรับการสตรีมผลลัพธ์จากโมเดล** คุณสามารถรับผลลัพธ์จาก LLM แบบสตรีม (Streamed Response) ได้ทันที พร้อมระบบตรวจสอบข้อมูลในระหว่างการส่ง ทำให้เหมาะสำหรับแอปที่ต้องตอบสนองแบบเรียลไทม์
- **เขียนโค้ดแบบกราฟด้วย Pydantic Graph** สำหรับแอปที่มีความซับซ้อน PydanticAI ยังรองรับการสร้างโฟลว์แบบกราฟด้วย Type Hint ช่วยให้การจัดการลำดับขั้นตอนต่างๆ มีความชัดเจน ลดปัญหาโค้ดพันกัน (Spaghetti Code)

PydanticAI คือเฟรมเวิร์กที่ออกแบบมาสำหรับนักพัฒนา Python ที่ต้องการพัฒนา AI Agent อย่างจริงจัง ให้ประสบการณ์แบบเดียวกับการใช้ FastAPI แต่เป็นโลกของ Generative AI

หากคุณกำลังมองหาเครื่องมือที่ช่วยให้การทำงานกับ LLM เป็นระบบมากขึ้น ใช้ซ้ำได้ ตรวจสอบได้ และขยายได้ในระยะยาว *PydanticAI คือคำตอบที่คุณไม่ควรมองข้าม*

ก้าวแรกสู่ AI Agent

ถ้าในโลกของการเขียนโปรแกรมมี “Hello, World!” เป็นบทเรียนเบื้องต้น การสร้าง AI Agent ก็เริ่มต้นจากสิ่งที่สำคัญไม่แพ้กัน นั่นคือ การสกัดข้อมูลแบบมีโครงสร้าง (Structured Data Extraction)

หนึ่งในความท้าทายในการทำงานร่วมกับ Large Language Models (LLMs) คือการรับมือกับผลลัพธ์ที่อยู่ในรูปแบบของ “ข้อความอิสระ” (Unstructured Text) ซึ่งยากต่อการนำไปใช้งานในระบบซอฟต์แวร์แบบดั้งเดิม นี่คือจุดที่ PydanticAI เข้ามามีบทบาทสำคัญ โดยทำหน้าที่เป็นสะพานเชื่อมระหว่างโลกของข้อความที่คลุมเครือของ LLM กับโลกของข้อมูลแบบมีโครงสร้างชัดเจนในแบบของ Pydantic

เราจะเริ่มต้นจากตัวอย่างง่ายๆ เพื่อเรียนรู้การใช้งาน PydanticAI อย่างเป็นรูปธรรม โดยการสร้าง Agent ที่สามารถอ่านข้อความธรรมดา แล้วสกัดข้อมูลชื่อ อายุ และแปลงข้อมูลที่ได้ ให้อยู่ในรูปแบบของ Pydantic Object ที่สามารถใช้งานและตรวจสอบความถูกต้องได้ทันที

เครื่องมือที่ต้องใช้มีดังนี้

- Python 3.9 ขึ้นไป

- ไลบรารี pydantic-ai และ openai
- OpenAI API Key

เริ่มจากการติดตั้งไลบรารีที่จำเป็น ดังนี้

```
pip install "pydantic-ai-slim[openai]"
```

จากนั้นให้ตั้งค่า OpenAI API Key ของคุณเป็น environment variable ดังนี้

สำหรับ Linux หรือ macOS

```
export OPENAI_API_KEY="sk-..."
```

สำหรับ Windows (Command Prompt)

```
set OPENAI_API_KEY="sk-..."
```

สำหรับ Windows (Powershell)

```
$env:OPENAI_API_KEY="sk-..."
```

นิยามโครงสร้างข้อมูลด้วย Pydantic โดยการออกแบบ “พิมพ์เขียว” ให้กับข้อมูลที่เราต้องการให้ LLM ช่วยดึงออกมา ดังนี้

สร้างไฟล์ชื่อ extract_user.py แล้วใส่โค้ดโมเดล User ต่อไปนี้

```
from pydantic import BaseModel, Field


class User(BaseModel):
    """
    A model for storing user data extracted from text
    The text in this docstring will be passed to LLM for understanding
    """
    name: str = Field(description="Full name and surname of the user")
    age: int = Field(description="Age of the user in years")
```

การระบุ docstring และ description ชัดเจนในแต่ละฟิลด์ ช่วยให้ LLM เข้าใจว่าแต่ละฟิลด์หมายถึงอะไร และควรดึงข้อมูลแบบไหน

จากนั้นเราจะสร้าง Agent ที่ทำหน้าที่ดึงข้อมูลจากข้อความให้เป็นไปตามโมเดล User ที่นิยามไว้ เพิ่มโค้ดลงในไฟล์ extract_user.py ดังนี้

```
from pydantic_ai import Agent
from pydantic_ai.models.openai import OpenAIModel

llm = OpenAIModel('gpt-4o')
agent = Agent(llm)

prompt = "My name is John Doe and I will be 30 years
old on my next birthday."

print(f"Input Prompt: '{prompt}'\n")

agent_run_result = agent.run_sync(prompt,
output_type=User)
result: User = agent_run_result.output

print("--- Extracted Data ---")
print(f"Name: {result.name}")
print(f"Age: {result.age}")
print(f"\nType of result: {type(result)}")
print("Object: ", repr(result))
print("----------------------")
```

รันทดสอบจาก Terminal รันไฟล์ด้วยคำสั่ง

```
python extract_user.py
```

คุณจะเห็นผลลัพธ์ในลักษณะนี้

```
Input Prompt: 'My name is John Doe and I will be 30
years old on my next birthday.'

--- Extracted Data ---
Name: John Doe
Age: 29

Type of result: <class '__main__.User'>
Object:  User(name='John Doe', age=29)
----------------------
```

วิเคราะห์ผลลัพธ์

1. **ความถูกต้องของข้อมูล** Agent สามารถสกัดชื่อ John Doe และอายุ 29 ได้อย่างแม่นยำ ในข้อความจะใช้คำว่า “will be 30” ซึ่งหมายถึงในอนาคตจะอายุ 30
2. **ผลลัพธ์เป็นวัตถุ Pydantic** ตัวแปร result ไม่ใช่แค่ string หรือ dict แต่เป็นอินสแตนซ์ของคลาส User ทำให้สามารถ

เรียกใช้ result.name และ result.age ได้โดยตรง พร้อมระบบตรวจสอบข้อมูลและ Type Hint ของ Pydantic

เบื้องหลัง PydanticAI ทำงานดังนี้

- แปลงโมเดล User ให้เป็น Schema ที่ LLM เข้าใจ เหมือน Function Calling
- สร้าง Prompt ที่ประกอบด้วย Schema และคำสั่งเพื่อให้ LLM ตอบกลับข้อมูลในโครงสร้างที่กำหนด
- รับค่าตอบกลับ มักเป็น JSON string และนำมาสร้างเป็นอินสแตนซ์ของ User พร้อมตรวจสอบความถูกต้องให้อัตโนมัติ

PydanticAI ช่วยให้เราสามารถ “ประกาศว่าเราต้องการข้อมูลแบบไหน” และปล่อยให้ Agent จัดการส่วนที่ยุ่งยากในการสื่อสารกับ LLM แทนเรา

ในบทถัดไป เราจะเจาะลึกองค์ประกอบต่างๆ ของ Agent รวมถึงการเชื่อมต่อกับ LLM จากผู้ให้บริการอื่น เพื่อขยายความสามารถของระบบต่อไป

องค์ประกอบของ PydanticAI

เรามาเจาะลึกองค์ประกอบสำคัญของ PydanticAI ซึ่งเป็นรากฐานในการสร้างแอปพลิเคชันที่ขับเคลื่อนด้วยปัญญาประดิษฐ์ ความเข้าใจในองค์ประกอบเหล่านี้จะช่วยให้คุณสามารถนำ PydanticAI ไปปรับใช้ในงานต่างๆ ได้อย่างยืดหยุ่น มีประสิทธิภาพ และสามารถขยายต่อยอดได้ในอนาคต

PydanticAI มีองค์ประกอบหลัก ดังนี้

1. **Agent** ตัวควบคุมหลัก หรือ “สมอง” ของระบบ
2. **LLM Models** ส่วนที่เชื่อมต่อกับผู้ให้บริการ Large Language Models
3. **Messages และ Chat History** กลไกการจัดการบริบทของบทสนทนา

Agent

Agent คือคลาสหลักที่ทำหน้าที่เป็นศูนย์กลางการประมวลผลทั้งหมดใน PydanticAI ทำหน้าที่รับคำสั่งจากผู้ใช้ (prompt), ทำงานประสานกับ LLM, ตัดสินใจว่าจะเรียกใช้เครื่องมือ

(Tools) ใดบ้าง และจัดการส่งผลลัพธ์กลับไปยังผู้ใช้อย่างมีแบบแผน

กระบวนการทำงานของ Agent ประกอบด้วย

- รับข้อมูลจากผู้ใช้
- สร้าง prompt ที่เหมาะสมสำหรับส่งไปยัง LLM
- เรียกใช้งาน LLM เพื่อประมวลผลคำตอบ
- ตรวจสอบผลลัพธ์ที่ได้ และตัดสินใจว่าจะตอบกลับหรือเรียกใช้เครื่องมือเพิ่มเติม
- ส่งผลลัพธ์สุดท้ายกลับในรูปแบบข้อมูลที่โครงสร้างชัดเจนด้วย Pydantic Model

LLM Model

PydanticAI ถูกออกแบบให้สามารถทำงานร่วมกับ LLM จากหลากหลายผู้ให้บริการได้อย่างยืดหยุ่น ผ่านคลาส Model ซึ่งเป็นตัวกลางในการสื่อสารกับ API ของแต่ละราย เช่น OpenAI, Google (Gemini) หรือ Anthropic (Claude) เป็นต้น ทำให้สามารถสลับเปลี่ยนโมเดลได้โดยไม่ต้องแก้ไขโค้ดมาก

OpenAI

ติดตั้ง OpenAI Model ดังนี้

```
pip install "pydantic-ai-slim[openai]"
```

ตั้งค่า API Key ของ OpenAI ใน environment variable ดังนี้

```
export OPENAI_API_KEY='your-api-key'
```

เชื่อมต่อกับโมเดลของ OpenAI อย่างเช่น gpt-5 สามารถทำได้ดังนี้

```
from pydantic_ai import Agent

agent = Agent('openai:gpt-5')
```

หรือเรียกใช้ผ่าน OpenAIModel ดังนี้

```
from pydantic_ai import Agent
from pydantic_ai.models.openai import OpenAIModel
```

```
model = OpenAIModel('gpt-5')
agent = Agent(model)
```

คุณสามารถกำหนด API Key ผ่าน provider ดังนี้

```
from pydantic_ai import Agent
from pydantic_ai.models.openai import OpenAIModel
from pydantic_ai.providers.openai import
OpenAIProvider

# แนะนำให้ตั้งค่า API Key ผ่าน environment variable
provider = OpenAIProvider(api_key='your-api-key')
model = OpenAIModel('gpt-5', provider=provider)
agent = Agent(model)
```

OpenAI Responses API

Pydantic AI รองรับการใช้งาน **OpenAI Responses API** โดยตรง ซึ่งเป็น API รุ่นใหม่ของ OpenAI ที่ออกแบบมาให้จัดการบริบทการสนทนา เครื่องมือ (tools) และเหตุผลการตอบ (reasoning) ได้อย่างมีประสิทธิภาพมากขึ้น

คุณสามารถใช้งาน OpenAIResponsesModel ได้ทั้งในรูปแบบการอ้างอิงชื่อโมเดลโดยตรง หรือการสร้างอินสแตนซ์ของ

โมเดลด้วยตนเอง

การใช้งานด้วยชื่อโมเดลโดยตรง

```
from pydantic_ai import Agent
from pydantic_ai.models.openai import
OpenAIResponsesModel

model = OpenAIResponsesModel('gpt-5')
agent = Agent(model)
...
```

Built-in Tools ของ Responses API

Responses API มาพร้อม **Built-in Tools** ที่สามารถใช้งานได้ทันที โดยไม่จำเป็นต้องพัฒนาเครื่องมือเอง ได้แก่

- **Web search** — ให้โมเดลค้นหาข้อมูลล่าสุดจากเว็บก่อนสร้างคำตอบ
- **Code interpreter** — ให้โมเดลเขียนและรันโค้ด Python ใน sandbox ก่อนตอบกลับ
- **Image generation** — ให้โมเดลสร้างภาพจากข้อความ
- **File search** — ให้โมเดลค้นหาข้อมูลจากไฟล์ของคุณ

- **Computer use** — ให้โมเดลควบคุมคอมพิวเตอร์เพื่อทำงานแทนผู้ใช้

เครื่องมือ Web search, Code interpreter, Image generation และ File search รองรับแบบ native ผ่านฟีเจอร์ **Built-in tools** ของ Pydantic AI

การเปิดใช้งาน Computer Use

เครื่องมือ **Computer use** สามารถเปิดใช้งานได้โดยกำหนด ComputerToolParam ใน openai_builtin_tools ของ OpenAIResponsesModelSettings

หมายเหตุ ในปัจจุบัน Computer use ยังไม่สร้าง BuiltinToolCallPart หรือ BuiltinToolReturnPart ใน message history และไม่รองรับ streamed events

```
from openai.types.responses import ComputerToolParam

from pydantic_ai import Agent
from pydantic_ai.models.openai import
OpenAIResponsesModel, OpenAIResponsesModelSettings

model_settings = OpenAIResponsesModelSettings(
```

```
    openai_builtin_tools=[
        ComputerToolParam(
            type='computer_use',
        )
    ],
)

model = OpenAIResponsesModel('gpt-5')
agent = Agent(model=model,
model_settings=model_settings)

result = agent.run_sync('Open a new browser tab')
print(result.output)
```

การอ้างอิงคำตอบก่อนหน้า

Responses API รองรับการอ้างอิงคำตอบก่อนหน้าของโมเดลผ่านพารามิเตอร์ previous_response_id เพื่อให้บริบทการสนทนาและ reasoning ก่อนหน้ายังคงอยู่ครบถ้วน

ใน Pydantic AI สามารถใช้งานได้ผ่านฟิลด์ openai_previous_response_id ใน OpenAIResponsesModelSettings

```python
from pydantic_ai import Agent
from pydantic_ai.models.openai import
OpenAIResponsesModel, OpenAIResponsesModelSettings


model = OpenAIResponsesModel('gpt-5')
agent = Agent(model=model)


result = agent.run_sync('The secret is 1234')


model_settings = OpenAIResponsesModelSettings(
    openai_previous_response_id=result.all_messages()
[-1].provider_response_id
)


result = agent.run_sync('What is the secret code?',
model_settings=model_settings)
print(result.output)
#> 1234
```

ด้วยวิธีนี้ โมเดลสามารถต่อยอดจาก reasoning เดิมของตนเองได้ โดยไม่จำเป็นต้องส่ง message history ทั้งหมดซ้ำอีกครั้ง

การอ้างอิงคำตอบก่อนหน้าแบบอัตโนมัติ

หากตั้งค่า openai_previous_response_id เป็น 'auto' Pydantic AI จะเลือก provider_response_id ล่าสุดจาก message history ให้โดยอัตโนมัติ และจะตัดข้อความก่อนหน้านั้นออกจากคำขอ เพื่อให้ OpenAI API ใช้ประโยชน์จาก server-side history ได้อย่างมีประสิทธิภาพมากขึ้น

```python
from pydantic_ai import Agent
from pydantic_ai.models.openai import
OpenAIResponsesModel, OpenAIResponsesModelSettings

model = OpenAIResponsesModel('gpt-5')
agent = Agent(model=model)

result1 = agent.run_sync('Tell me a joke.')
print(result1.output)
#> Did you hear about the toothpaste scandal? They
called it Colgate.

# เมื่อกำหนดเป็น 'auto'
# ระบบจะส่งเฉพาะ provider_response_id ล่าสุด
# และข้อความหลังจากนั้นไปยัง OpenAI
model_settings = OpenAIResponsesModelSettings(
    openai_previous_response_id='auto'
)

result2 = agent.run_sync(
```

```
    'Explain?',
    message_history=result1.new_messages(),
    model_settings=model_settings
)
print(result2.output)
#> This is an excellent joke invented by Samuel
Colvin, it needs no explanation.
```

แนวทางนี้ช่วยลดขนาดของ context ที่ต้องส่งซ้ำ และเพิ่มประสิทธิภาพในการเรียกใช้งาน Responses API โดยยังคงความต่อเนื่องของการสนทนาไว้ครบถ้วน

Google

ติดตั้ง Google Model ดังนี้

```
pip install "pydantic-ai-slim[google]"
```

ตั้งค่า API Key ของ Gemini ใน Environment Variable ดังนี้

```
export GOOGLE_API_KEY=your-api-key
```

Google Generative Language API

เชื่อมต่อกับโมเดลของ Google ทำได้ได้ดังนี้

```
from pydantic_ai import Agent


agent = Agent('google-gla:gemini-2.5-pro')
...
```

หรือตั้งค่าผ่าน GoogleModel ดังนี้

```
from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel
from pydantic_ai.providers.google import
GoogleProvider


provider = GoogleProvider(api_key='your-api-key')
model = GoogleModel('gemini-2.5-flash',
provider=provider)
agent = Agent(model)
```

Vertext AI

หากคุณใช้ Vertext AI ในบริการ Google Cloud เรียกใช้งานได้ดังนี้

```
from pydantic_ai import Agent


agent = Agent('google-vertex:gemini-2.5-pro')
...
```

หรือ

```
from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel
from pydantic_ai.providers.google import
GoogleProvider


provider = GoogleProvider(vertexai=True)
model = GoogleModel('gemini-2.5-flash',
provider=provider)
agent = Agent(model)
```

หากคุณต้องการระบุ Service Account ผ่านไฟล์ JSON ทำได้ดังนี้

```
from google.oauth2 import service_account


from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel
from pydantic_ai.providers.google import
GoogleProvider
```

```
credentials =
service_account.Credentials.from_service_account_file
(
    'path/to/service-account.json',
    scopes=['https://www.googleapis.com/auth/cloud-
platform'],
)
provider = GoogleProvider(credentials=credentials)
model = GoogleModel('gemini-2.5-flash',
provider=provider)
agent = Agent(model)
...
```

หากต้องการระบุ location ที่ให้บริการ VertexAI ทำได้ดังนี้

```
from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel
from pydantic_ai.providers.google import
GoogleProvider


provider = GoogleProvider(vertexai=True,
location='asia-east1', project='your-gcp-project-id')
model = GoogleModel('gemini-2.5-flash',
provider=provider)
```

```
agent = Agent(model)
...
```

นอกจากนี้ยังสามารถตั้งค่า Model เพิ่มเติมโดยใช้ GoogleModelSettings ดังนี้

```
from google.genai.types import HarmBlockThreshold,
HarmCategory


from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel,
GoogleModelSettings


settings = GoogleModelSettings(
    temperature=0.2,
    max_tokens=1024,
    google_thinking_config={'thinking_budget': 2048},
    google_safety_settings=[
        {
            'category':
HarmCategory.HARM_CATEGORY_HATE_SPEECH,
            'threshold':
HarmBlockThreshold.BLOCK_LOW_AND_ABOVE,
        }
    ]
)
```

```
model = GoogleModel('gemini-2.5-flash')
agent = Agent(model, model_settings=settings)
```

Model Garden

คุณสามารถเข้าถึงโมเดลจาก **Model Garden** ที่รองรับ generateContent API และเปิดให้ใช้งานภายใต้โปรเจกต์ GCP ของคุณได้ ซึ่งรวมถึงโมเดลตระกูล Gemini และโมเดลอื่น ๆ อีกมากมาย โดยสามารถระบุ model_name ได้ตามรูปแบบต่อไปนี้

- {model_id} สำหรับโมเดล Gemini
- {publisher}/{model_id}
- publishers/{publisher}/models/{model_id}
- projects/{project}/locations/{location}/publishers/{publisher}/models/{model_id}

รูปแบบเหล่านี้ช่วยให้คุณอ้างอิงโมเดลได้ทั้งจากผู้เผยแพร่ (publisher) โดยตรง หรือจากโมเดลที่ถูกผูกไว้กับโปรเจกต์และภูมิภาค (location) เฉพาะใน GCP

ตัวอย่างด้านล่างแสดงการใช้งานโมเดลจาก Model Garden ผ่าน GoogleModel โดยกำหนดค่าโปรเจกต์และภูมิภาคด้วย GoogleProvider

```
from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel
from pydantic_ai.providers.google import
GoogleProvider

provider = GoogleProvider(
    project='your-gcp-project-id',
    location='us-central1',  # region ที่โมเดลพร้อมให้ใช้
งาน
)

model = GoogleModel(
    'meta/llama-3.3-70b-instruct-maas',
    provider=provider
)

agent = Agent(model)
...
```

ในตัวอย่างนี้ เอเจนต์จะใช้โมเดล meta/llama-3.3-70b-instruct-maas จาก Model Garden ซึ่งถูกเรียกใช้งานผ่านโปรเจกต์และภูมิภาคที่กำหนดไว้ ทำให้สามารถผสานโมเดลจากผู้

ให้บริการหลายรายเข้ากับ Pydantic AI ได้อย่างยืดหยุ่นภายใต้โครงสร้างของ GCP

การรับอินพุตแบบ Document, Image, Audio และ Video

GoogleModel รองรับอินพุตแบบ **Multi-modal** อย่างครบถ้วน ไม่ว่าจะเป็นเอกสาร รูปภาพ เสียง หรือวิดีโอ ทำให้สามารถส่งข้อมูลหลากหลายรูปแบบให้โมเดลประมวลผลร่วมกันได้

การใช้งาน YouTube Video URL โดยตรง

คุณสามารถส่ง URL ของวิดีโอ YouTube ให้กับโมเดล Google ได้โดยตรง เช่น

```
from pydantic_ai import Agent, VideoUrl
from pydantic_ai.models.google import GoogleModel

agent = Agent(GoogleModel('gemini-2.5-flash'))
result = agent.run_sync(
    [
        'What is this video about?',
        VideoUrl(url='https://www.youtube.com/watch?v=dQw4w9WgXcQ'),
```

```
    ]
)
print(result.output)
```

ในตัวอย่างนี้ เอเจนต์จะรับทั้งคำถามและ URL ของวิดีโอ เพื่อให้โมเดลวิเคราะห์เนื้อหาของวิดีโอและตอบกลับมาโดยตรง

การอัปโหลดไฟล์ผ่าน Files API

นอกจาก URL ภายนอกแล้ว คุณยังสามารถอัปโหลดไฟล์ผ่าน **Files API** และส่งไฟล์นั้นให้โมเดลในรูปแบบ URL ได้เช่นกัน

```
from pydantic_ai import Agent, DocumentUrl
from pydantic_ai.models.google import GoogleModel
from pydantic_ai.providers.google import
GoogleProvider


provider = GoogleProvider()
file = provider.client.files.upload(file='pydantic-
ai-logo.png')
assert file.uri is not None


agent = Agent(GoogleModel('gemini-2.5-flash',
provider=provider))
```

```
result = agent.run_sync(
    [
        'What company is this logo from?',
        DocumentUrl(url=file.uri,
media_type=file.mime_type),
    ]
)
print(result.output)
```

ในกรณีนี้ ไฟล์รูปภาพจะถูกอัปโหลดก่อน จากนั้นนำ uri ของไฟล์มาใช้เป็นอินพุตให้กับเอเจนต์ พร้อมระบุชนิดสื่อ (media_type) ให้ถูกต้อง

การตั้งค่าโมเดล

คุณสามารถปรับพฤติกรรมของโมเดลได้ผ่าน GoogleModelSettings เพื่อควบคุมรูปแบบการตอบ ความยาวของผลลัพธ์ และการตั้งค่าด้านความปลอดภัย

```
from google.genai.types import HarmBlockThreshold,
HarmCategory


from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel,
GoogleModelSettings
```

```
settings = GoogleModelSettings(
    temperature=0.2,
    max_tokens=1024,
    google_thinking_config={'thinking_level': 'low'},
    google_safety_settings=[
        {
            'category':
HarmCategory.HARM_CATEGORY_HATE_SPEECH,
            'threshold':
HarmBlockThreshold.BLOCK_LOW_AND_ABOVE,
        }
    ]
)

model = GoogleModel('gemini-2.5-pro')
agent = Agent(model, model_settings=settings)
...
```

การตั้งค่าเหล่านี้ช่วยให้คุณควบคุมระดับความสุ่มของคำตอบ (temperature) จำนวนโทเคนสูงสุด และระดับการ “คิด” ของโมเดล รวมถึงกำหนดกฎด้านความปลอดภัยได้อย่างละเอียด

การปิด Thinking

สำหรับโมเดลที่เก่ากว่า **Gemini 2.5 Pro** คุณสามารถปิดการทำงานด้าน thinking ได้ โดยตั้งค่า thinking_budget เป็น 0

```
from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel,
GoogleModelSettings

model_settings = GoogleModelSettings(
    google_thinking_config={'thinking_budget': 0}
)

model = GoogleModel('gemini-2.5-flash')
agent = Agent(model, model_settings=model_settings)
...
```

การตั้งค่าความปลอดภัย

คุณสามารถกำหนดนโยบายด้านความปลอดภัยของโมเดลได้ผ่านฟิลด์ google_safety_settings เช่น การบล็อกเนื้อหาที่เข้าข่าย Hate Speech

```
from google.genai.types import HarmBlockThreshold,
HarmCategory


from pydantic_ai import Agent
from pydantic_ai.models.google import GoogleModel,
GoogleModelSettings


model_settings = GoogleModelSettings(
    google_safety_settings=[
        {
            'category':
HarmCategory.HARM_CATEGORY_HATE_SPEECH,
            'threshold':
HarmBlockThreshold.BLOCK_LOW_AND_ABOVE,
        }
    ]
)


model = GoogleModel('gemini-2.5-flash')
agent = Agent(model, model_settings=model_settings)
...
```

การตั้งค่านี้ช่วยให้คุณควบคุมระดับการกรองเนื้อหาได้ตามข้อกำหนดด้านจริยธรรมและความปลอดภัยของแอปพลิเคชันที่คุณพัฒนา

Anthropic (Claude)

ติดตั้ง Anthropic Model ดังนี้

```
pip install "pydantic-ai-slim[anthropic]"
```

ตั้งค่า API Key ของ Anthropic ใน Environment Variable ดังนี้

```
export ANTHROPIC_API_KEY='your-api-key'
```

เรียกใช้โมเดล Claude ของ Anthropic ดังนี้

```
from pydantic_ai import Agent

agent = Agent('anthropic:claude-4-5-sonnet-latest')
```

หรือเรียกใช้ผ่าน AnthropicModel ดังนี้

```
from pydantic_ai import Agent
from pydantic_ai.models.anthropic import
AnthropicModel

model = AnthropicModel('claude-4-5-sonnet-latest')
agent = Agent(model)
```

กำหนด API Key ผ่าน provider ดังนี้

```
from pydantic_ai import Agent
from pydantic_ai.models.anthropic import
AnthropicModel
from pydantic_ai.providers.anthropic import
AnthropicProvider


provider = AnthropicProvider(api_key='your-api-key')
model = AnthropicModel('claude-4-5-sonnet-latest',
provider=provider)
agent = Agent(model)
```

การใช้งานร่วมกับ Cloud Platform

คุณสามารถใช้งานโมเดลของ Anthropic ผ่านแพลตฟอร์มคลาวด์ต่าง ๆ ได้ โดยการส่ง **custom client** เข้าไปให้กับ AnthropicProvider

AWS Bedrock

หากต้องการใช้งานโมเดล Claude ผ่าน AWS Bedrock ให้ทำตามเอกสารของ Anthropic เพื่อสร้าง