Reading: Structured Outputs in CrewAI

Estimated time: 6 minutes

Objectives

After completing this reading, you will be able to:

- Explain the importance of structured outputs in AI workflows
- Explain the use of Pydantic to enable structured outputs
- List the key features of Pydantic in the context of structured outputs
- Explore how structured outputs are implemented in CrewAI
- List the benefits of using structured outputs in CrewAI

Why structured outputs matter in AI workflows

How outputs are structured plays a critical role in AI applications, especially those with multiple agents or complex data. Free-form text from language models can be difficult to parse, prone to ambiguity, and risky for downstream tasks. On the other hand, structured outputs (like JSON or objects with defined fields) ensure consistency and make data easier to extract and use.

CrewAI helps developers enforce structured outputs by letting them define schemas for task responses, which leads to more reliable and predictable outcomes. In multi-agent workflows, this structure ensures that one agent's output can be cleanly interpreted by the next, thus reducing miscommunication, data loss, and hallucinations. The result is smoother agent collaboration and easier system integration.

Introduction to Pydantic for data modeling

To enable structured outputs, CrewAI primarily leverages a powerful Python library called Pydantic. Pydantic is commonly used for data validation and settings management by defining data models in Python. A Pydantic model is essentially a class that inherits from BaseModel and defines a set of fields with types.

When you create an instance of this model, Pydantic automatically checks that any data you pass in matches the expected types (and even converts types when possible). If the data is missing required fields or has the wrong type, Pydantic will raise a validation error, alerting you to the mismatch.

Key features of Pydantic

Here are some of the key features of Pydantic:

Feature	Description

Data validation	Ensures inputs match expected types. If a field expects an integer but receives a string, Pydantic will raise an error.
	`from pydantic import BaseModel`
	`class Person(BaseModel):`
	`  age: int`
	`Person(age="twenty")  # Raises ValidationError`

Automatic type conversion	Pydantic attempts to coerce inputs into the expected type when possible.
	`from datetime import date`
	`from pydantic import BaseModel`
	`class Event(BaseModel):`
	`  date: date`
	`e = Event(date="2025-07-24")`
	`print(e.date)  # Outputs: 2025-07-24`

Nested models and complex types	Models can include other models, lists, and optional fields. This is ideal for structured data like JSON.
	`from typing import List`
	`from pydantic import BaseModel`
	`class Item(BaseModel):`
	`  name: str`
	`class Cart(BaseModel):`
	`  items: List[Item]`
	`cart = Cart(items=[{"name": "Apple"}])`

Easy serialization	Convert models to dictionaries or JSON strings for storage or API responses.
	`user = Person(age=30)`
	`print(user.dict())  # {'age': 30}`
	`print(user.json())  # '{"age": 30}'`

Implementing structured output in a CrewAI task

Let us see how we can actually use Pydantic with CrewAI to get structured outputs:

Concept	Explanation

Define a Pydantic model	Start by defining what data your task should return. Create a class that inherits from `BaseModel` and specify typed fields:
	`from pydantic import BaseModel`
	`class BlogSummary(BaseModel):`
	`  title: str`
	`  content: str`
	This model enforces a schema with two required fields: `title` and `content`. You can also nest models or use lists and dicts for more complex structures.

Nested Pydantic models	You can define a model that contains other models. This is useful when your output has grouped or hierarchical data.
	`from typing import List`
	`from pydantic import BaseModel`
	`class Ingredient(BaseModel):`
	`  name: str`
	`  quantity: str`
	`class MealPlan(BaseModel):`
	`  meal_name: str`
	`  ingredients: List[Ingredient]`
	This lets you represent nested data, such as a list of ingredients inside a meal plan. CrewAI will validate every field inside each nested model automatically.

Attach model to task	In your CrewAI task, use the `output_pydantic` (or `output_json`) parameter to bind the model. This enforces structured output:
	`blog_task = Task(`
	`  description="Generate a catchy blog title and a short content about a topic.",`
	`  expected_output="A JSON object with 'title' and 'content' fields.",`
	`  agent=blog_agent,`
	`  output_pydantic=BlogSummary`
	`)`
	Use `output_pydantic` if you want the output as a Pydantic object, or `output_json` if you prefer a plain Python dict.

Using YAML with Pydantic	While you can't reference Pydantic classes directly in YAML, you can define task configurations in YAML and attach the model in Python using `CrewBase`:
	`@task`
	`def blog_task(self) -> Task:`
	`  return Task(`
	`  config=self.tasks_config['blog_task'],  # From YAML`
	`  output_json=BlogSummary  # Defined in Python`
	`  )`
	This approach combines YAML's ease of editing with Python's strict schema enforcement.

Run the crew	Execute the crew as usual using `crew.kickoff()`. The returned result includes structured data: • `result.raw`: Raw model output • `result.json_dict`: Dict output (if using `output_json`) • `result.pydantic`: Pydantic object (if using `output_pydantic`) You can also use dictionary-like access with `result["title"]` due to implemented **`getitem`** support.

Use the structured data	With structured output, you can cleanly feed data into downstream systems, pass it to other tasks, or serialize it:
	`title = result.pydantic.title  # if using output_pydantic`
	`title = result.json_dict.get("title")  # if using output_json`
	This makes AI output a reliable part of your program's data pipeline—no parsing or guesswork required.

Why structured output matters in CrewAI

Now that we've seen how to define and apply Pydantic models in CrewAI, it's important to understand why this practice is so valuable in real-world workflows. The following table outlines the practical benefits of using structured outputs—not just in theory, but in how they help CrewAI applications become more robust, maintainable, and production-ready.

Benefit	How it helps in CrewAI workflows
Type safety and validation	CrewAI validates the LLM output against the schema you define. If the model returns incorrect types or misses required fields, you're immediately notified. This prevents subtle bugs from creeping into your pipeline.
Clear data contracts	A Pydantic model acts like a formal agreement: "This task will always return these fields, with these types." This makes the codebase easier to understand, especially in collaborative projects where agents and tasks are reused.
System integration	Whether you're pushing the output to an API, saving it to a database, or displaying it in a UI, structured outputs make integration seamless. There's no need to manually parse or clean the LLM response.
Less post-processing	Without structured outputs, developers often write custom parsing or regex logic to extract information from raw text. With CrewAI and Pydantic, the heavy lifting is handled for you—validated, parsed, and ready to use.
Consistent agent handoffs	In multi-agent workflows, structure ensures smooth transitions. One agent's output becomes the next agent's input, without ambiguity. This avoids hallucinated formats or misinterpretation between agents.
LLM behavior alignment	When LLMs are prompted to follow a structured schema (e.g., "Return a JSON with fields A, B, and C"), they are more likely to stay focused and accurate. It works like soft prompting or function-calling—constraining the model's output space.

Summary

Structured outputs in CrewAI let you combine the flexibility of language models with the reliability of defined data formats. By using Pydantic models, you create a clear structure that CrewAI enforces, making outputs easier to validate, reuse, and integrate. This approach helps you build more reliable workflows, connect multiple agents smoothly, and plug AI results directly into real systems. It's a key step in turning experimental AI into production-ready applications.

Author

Karan Goswami