Lesson 2 — Tool Calling and Schema Design

Overview

This lesson explores how to create robust, validated tools for LangChain agents using Pydantic schemas and the StructuredTool pattern. You'll learn how to transform Python functions into agent-callable tools with automatic input validation, type safety, and LangGraph compatibility.

By the end of this lesson, you'll understand:

- How Pydantic BaseModel provides runtime type validation
- The StructuredTool pattern for wrapping functions
- LangGraph compatibility requirements (kwargs handling)
- Best practices for tool design in production systems

Learning Outcomes

After completing this lesson, you will be able to:

1. Design Pydantic schemas for tool input validation
2. Create StructuredTool instances from Python functions
3. Handle LangGraph's kwargs-based invocation pattern
4. Validate tool inputs with comprehensive error messages
5. Integrate tools seamlessly with LangChain agents

Prerequisites

- Completion of Lesson 1 (environment setup)
- Understanding of Python type hints
- Familiarity with function decorators
- Basic knowledge of JSON schema concepts

The Problem: Unreliable Tool Inputs

When building agentic systems, LLMs must call external tools (functions, APIs) to perform actions. However, LLMs can:

- Pass incorrect types (string instead of integer)
- Omit required parameters
- Provide malformed data structures

Without validation, these errors propagate into your system, causing:

- Runtime crashes
- Silent failures
- Incorrect computations
- Poor user experience

Solution: Use Pydantic schemas to validate all tool inputs before execution.

Part 1: Understanding Pydantic BaseModel

What is Pydantic?

Pydantic (v2.12+) is Python's most widely-used data validation library. It provides:

- **Runtime validation**: Checks data at execution time, not just type-checking
- **Type coercion**: Automatically converts compatible types (e.g., `"123"` → `123`)
- **Clear error messages**: Detailed validation failures with paths
- **JSON Schema generation**: Automatic schema export for tool definitions
- **Performance**: Core validation written in Rust for speed

Basic Pydantic Example

```
from datetime import datetime
from pydantic import BaseModel, Field, PositiveInt

class User(BaseModel):
    """User model with validation."""
    id: int
    name: str = "John Doe"  # Default value
    signup_ts: datetime | None = None
    tastes: dict[str, PositiveInt]  # Values must be positive integers

# Valid input - Pydantic coerces types where appropriate
user_data = {
    'id': '123',  # String → int
    'signup_ts': '2024-01-01 12:00',  # String → datetime
    'tastes': {
        'coffee': '9',  # String → int
        'tea': 7
    }
}

user = User(**user_data)
print(user.id)  # 123 (integer)
print(user.signup_ts)  # datetime.datetime(2024, 1, 1, 12, 0)
```

Validation Errors

```
# Invalid input
bad_data = {
    'id': 'not_an_int',  # Cannot parse
    'tastes': {}
}

try:
    User(**bad_data)
except ValidationError as e:
    print(e.errors())
    # [
    #     {'type': 'int_parsing', 'loc': ('id',), 'msg': 'Input should be a valid integer'},
    #     {'type': 'missing', 'loc': ('signup_ts',), 'msg': 'Field required'}
    # ]
```

Key Features for Tool Design

1. Field Descriptions (for LLMs)

```
from pydantic import Field

class ToolInput(BaseModel):
    campaign_name: str = Field(
        description="Name of the campaign (e.g., 'Campaign 1')"
    )
    data_types: list[str] = Field(
        description="List of data types requested (e.g., ['impacts_by_sex_age'])"
    )
```

The `description` parameter generates JSON schema that LLMs read to understand tool parameters.

2. Optional vs Required Fields

```
class SearchInput(BaseModel):
    query: str  # Required
    filters: dict[str, str] | None = None  # Optional with default None
    max_results: int = 10  # Optional with default value
```

3. Type Validation

```
from pydantic import PositiveInt, conlist

class FilterInput(BaseModel):
    table_names: conlist(str, min_length=1)  # List of strings, at least 1 item
    max_tables: PositiveInt = 100  # Must be > 0
```

Part 2: Creating Tools with StructuredTool

The StructuredTool Pattern

LangChain provides `StructuredTool` to wrap Python functions as agent tools with:

- Automatic input validation via Pydantic schemas
- JSON schema generation for LLM consumption
- Error handling and type safety
- Integration with LangChain's tool-calling infrastructure

Basic Pattern

```
from langchain_core.tools import StructuredTool
from pydantic import BaseModel, Field

# Step 1: Define input schema
class MyToolInput(BaseModel):
    param1: str = Field(description="First parameter")
    param2: int = Field(description="Second parameter", ge=0)

# Step 2: Define the function
def my_function(param1: str, param2: int) -> dict:
    """Execute some operation."""
    return {"result": f"{param1}: {param2}"}

# Step 3: Wrap as StructuredTool
my_tool = StructuredTool.from_function(
    func=my_function,
    name="my_tool",
    description="Performs my operation with validated inputs",
    args_schema=MyToolInput,
)
```

Real Example: list_available_fixtures

Let's examine the actual implementation from `lesson_01_working_code.py` (lines 527-534):

```
from pathlib import Path
from typing import Any
from langchain_core.tools import StructuredTool

def list_available_fixtures(campaign_dir: Path) -> dict[str, Any]:
    """
    List all available fixture files from a campaign directory.

    Args:
        campaign_dir: Path to campaign root directory

    Returns:
        Dictionary with available fixtures and their paths
    """
    # ... implementation details in Part 3 ...

# Create the tool
list_fixtures_tool = StructuredTool.from_function(
```

```
    func=lambda **_: list_available_fixtures(campaign_dir),
    name="list_available_fixtures",
    description="List all available Campaign 1 fixture files.",
)
```

Note the `lambda **_` pattern - this is crucial for LangGraph compatibility.

Part 3: LangGraph Compatibility (kwargs Handling)

The Challenge

LangChain's `create_agent` (built on LangGraph) invokes tools differently than direct function calls:

```
# Direct call
result = list_available_fixtures(campaign_dir)

# LangGraph call
result = tool.invoke(input={}, config={})  # Passes empty dict even if no args needed
```

LangGraph always passes keyword arguments (`**kwargs`), even when a tool expects no inputs or has pre-bound parameters.

The Solution: **_ Pattern

Use `**_` (kwargs unpacking with throwaway variable) to accept and ignore unexpected arguments:

```
list_fixtures_tool = StructuredTool.from_function(
    func=lambda **_: list_available_fixtures(campaign_dir),
    #            ^^^^ Accepts any kwargs, ignores them
    name="list_available_fixtures",
    description="List all available Campaign 1 fixture files.",
)
```

Why this works:

- `**_` captures all keyword arguments into a throwaway dict named `_`
- The actual function call `list_available_fixtures(campaign_dir)` uses the pre-bound `campaign_dir`
- LangGraph can pass `{}` without causing a TypeError

Pattern Variations

No-argument tools (pre-bound parameters)

```
# Campaign directory is pre-bound from agent config
tool = StructuredTool.from_function(
    func=lambda **_: list_available_fixtures(campaign_dir),
```

```
    name="list_fixtures",
    description="Lists available fixtures",
)
```

Tools with LLM-provided arguments

```
class DataRequestInput(BaseModel):
    campaign_name: str
    data_types: list[str]
    target_audience: str | None = None

tool = StructuredTool.from_function(
    func=lambda campaign_name, data_types, target_audience=None, **_: (
        generate_mock_api_call(
            campaign_dir,
            campaign_name=campaign_name,
            data_types=data_types,
            target_audience=target_audience,
        )
    ),
    name="generate_api_call",
    description="Generate a mock API call for downloading campaign data",
    args_schema=DataRequestInput,
)
```

Key points:

1. Extract required parameters explicitly: `campaign_name, data_types, target_audience=None`
2. Add `**_` at the end to catch unexpected kwargs
3. Pass extracted values to the actual function

Part 4: Implementing Tool Schemas (Campaign 1 Examples)

Example 1: DataRequestInput

From `lesson_01_working_code.py` (lines 414-420):

```
from pydantic import BaseModel, Field

class DataRequestInput(BaseModel):
    """Input schema for data request analysis."""

    campaign_name: str = Field(
        description="Name of the campaign (e.g., 'Campaign 1')"
    )
    data_types: list[str] = Field(
        description="List of data types requested (e.g., ['impacts_by_sex_age', 'tv_spot_schedule'])"
    )
    target_audience: str | None = Field(
        default=None,
        description="Target audience specification if mentioned"
```

```
    )
```

Design decisions:

- `campaign_name` : Required string (no default) - LLM must provide
- `data_types` : Required list of strings - can request multiple data types
- `target_audience` : Optional (default=None) - not all requests specify target

JSON Schema generated (what LLM sees):

```
{
  "type": "object",
  "properties": {
    "campaign_name": {
      "type": "string",
      "description": "Name of the campaign (e.g., 'Campaign 1')"
    },
    "data_types": {
      "type": "array",
      "items": {"type": "string"},
      "description": "List of data types requested..."
    },
    "target_audience": {
      "type": "string",
      "description": "Target audience specification if mentioned",
      "default": null
    }
  },
  "required": ["campaign_name", "data_types"]
}
```

Example 2: TableFilterInput

From `lesson_01_working_code.py` (lines 422-424):

```
class TableFilterInput(BaseModel):
    """Input schema for table filtering based on allowlist."""

    table_names: list[str] = Field(
        description="List of table element keys to filter"
    )
```

Simplicity by design:

- Single required field
- Clear, focused responsibility
- Self-documenting through description

Part 5: Complete Tool Creation Pattern

Step-by-Step Process

1. Design the Function

```
from pathlib import Path
from typing import Any

def list_available_fixtures(campaign_dir: Path) -> dict[str, Any]:
    """
    List all available fixture files from a campaign directory.

    Args:
        campaign_dir: Path to campaign root directory

    Returns:
        Dictionary with available fixtures and their paths
    """
    logger.info(f"Listing fixtures for: {campaign_dir.name}")

    if not campaign_dir.exists():
        raise OSError(f"Campaign directory not found: {campaign_dir}")

    input_dir = campaign_dir / "01_pre_postprocessing" / "input_from_api"

    if not input_dir.exists():
        logger.warning(f"Fixture directory not found: {input_dir}")
        return {
            "campaign": campaign_dir.name,
            "available_fixtures": [],
            "fixture_paths": {},
            "count": 0,
        }

    available_files = {}
    for file_path in sorted(input_dir.glob("*.json")):
        available_files[file_path.stem] = str(file_path)

    return {
        "campaign": campaign_dir.name,
        "available_fixtures": list(available_files.keys()),
        "fixture_paths": available_files,
        "count": len(available_files),
    }
```

2. Create the Tool (No Schema Needed)

For functions with pre-bound parameters (no LLM input):

```
from langchain_core.tools import StructuredTool

list_fixtures_tool = StructuredTool.from_function(
    func=lambda **_: list_available_fixtures(campaign_dir),
    name="list_available_fixtures",
    description="List all available Campaign 1 fixture files.",
)
```

3. Create Tool with Schema (LLM-Provided Inputs)

For functions requiring LLM-provided parameters:

```
from pydantic import BaseModel, Field

class TableFilterInput(BaseModel):
    table_names: list[str] = Field(
        description="List of table element keys to filter"
    )

def filter_tables_by_allowlist(table_names: list[str]) -> dict[str, Any]:
    """Filter table names based on AGENTS.md allowlist rules."""
    # ... implementation ...
    return {"allowed_tables": [...], "rejected_tables": [...]}

filter_tables_tool = StructuredTool.from_function(
    func=lambda table_names, **_: filter_tables_by_allowlist(table_names),
    name="filter_tables_by_allowlist",
    description="Filter table names based on allowlist rules (TRP/TabSummary only)",
    args_schema=TableFilterInput,
)
```

Part 6: Testing and Validation

Manual Tool Testing

```
# Test the underlying function
result = list_available_fixtures(Path("Starter Kit – Agentic Models/Campaign 1"))
assert result["count"] > 0
assert "impacts_in_target" in result["available_fixtures"]

# Test the tool invocation
tool_result = list_fixtures_tool.invoke({})
assert tool_result["count"] > 0
```

Schema Validation Testing

```
from pydantic import ValidationError

# Test valid input
valid_input = {
    "campaign_name": "Campaign 1",
    "data_types": ["impacts_in_target", "tv_spot_schedule"],
    "target_audience": "W25-54"
}
schema = DataRequestInput(**valid_input)
assert schema.campaign_name == "Campaign 1"

# Test invalid input
try:
```

```
    invalid_input = {
        "campaign_name": 123,  # Should be string
        "data_types": "not_a_list"  # Should be list
    }
    DataRequestInput(**invalid_input)
    assert False, "Should have raised ValidationError"
except ValidationError as e:
    assert len(e.errors()) >= 2
    print("Validation correctly rejected invalid input")
```

Integration Testing with Agent

```
from langchain.agents import create_agent
from langchain_google_genai import ChatGoogleGenerativeAI

# Create agent with tools
llm = ChatGoogleGenerativeAI(model="gemini-2.5-flash", temperature=0.0)
tools = [list_fixtures_tool, filter_tables_tool]
agent = create_agent(llm, tools=tools, system_prompt="You are a helpful assistant")

# Test tool invocation via agent
response = agent.invoke({
    "messages": [{"role": "user", "content": "List available fixtures"}]
})

# Verify tool was called
assert "impacts_in_target" in str(response)
```

Part 7: Best Practices and Common Pitfalls

Best Practices

1. Always Use Field Descriptions

```
# Bad: No description
class BadInput(BaseModel):
    campaign_name: str

# Good: Clear description for LLM
class GoodInput(BaseModel):
    campaign_name: str = Field(
        description="Name of the campaign (e.g., 'Campaign 1')"
    )
```

2. Provide Examples in Descriptions

```
data_types: list[str] = Field(
    description=(
        "List of data types requested. "
        "Examples: ['impacts_by_sex_age', 'tv_spot_schedule', 'target_universe']"
```

```
    )
)
```

3. Use Appropriate Defaults

```
# Optional with sensible default
max_results: int = Field(
    default=100,
    description="Maximum number of results to return",
    ge=1,  # Must be >= 1
    le=1000  # Cannot exceed 1000
)
```

4. Validate Constraints

```
from pydantic import constr, conlist

class StrictInput(BaseModel):
    email: constr(pattern=r'^[\w\.-]+@[\w\.-]+\.\w+$')  # Regex validation
    tags: conlist(str, min_length=1, max_length=10)  # 1-10 items
    age: int = Field(ge=0, le=150)  # 0-150 range
```

Common Pitfalls

Pitfall 1: Forgetting **_ for LangGraph

```
# This will fail with LangGraph
tool = StructuredTool.from_function(
    func=lambda: my_function(),  # Missing **_
    name="my_tool",
    description="..."
)

# LangGraph calls: tool.invoke({})
# Error: lambda() got unexpected keyword arguments
```

Fix:

```
tool = StructuredTool.from_function(
    func=lambda **_: my_function(),  # Accepts any kwargs
    name="my_tool",
    description="..."
)
```

Pitfall 2: Mutable Default Values

```
# Dangerous: mutable default
class BadSchema(BaseModel):
    filters: dict = {}  # Same dict instance shared!
```

```
# Safe: use default_factory or None
class GoodSchema(BaseModel):
    filters: dict | None = None  #  Safe
```

Pitfall 3: Missing Required Fields

```
# Unclear: is campaign_name required?
class UnclearInput(BaseModel):
    campaign_name: str = ""  # Empty string default - looks optional

# Clear: explicitly required
class ClearInput(BaseModel):
    campaign_name: str  # No default = required
```

Pitfall 4: Overly Complex Schemas

```
# Too complex for LLM
class ComplexInput(BaseModel):
    nested: dict[str, dict[str, list[tuple[int, str]]]]  # Confusing

# Simpler, clearer
class SimpleInput(BaseModel):
    tags: list[str]
    metadata: dict[str, str]
```

Part 8: Hands-On Exercise

Exercise: Create a New Tool

Objective: Create a tool that validates Campaign 1 fixtures exist and returns their file sizes.

Requirements:

1. **Schema Design:**
   - Create `FixtureValidationInput` schema
   - Required field: `fixture_names` (list of strings)
   - Optional field: `check_content` (boolean, default False)
2. **Function Implementation:**
   - Function name: `validate_fixtures`
   - Check if each fixture file exists
   - Return file sizes in bytes
   - If `check_content=True`, verify JSON is valid
3. **Tool Creation:**
   - Use StructuredTool.from_function
   - Name: "validate_fixtures"
   - LangGraph-compatible kwargs handling

Starter Code:

```
from pathlib import Path
from typing import Any
import json
from pydantic import BaseModel, Field
from langchain_core.tools import StructuredTool

class FixtureValidationInput(BaseModel):
    """TODO: Define schema"""
    pass

def validate_fixtures(
    campaign_dir: Path,
    fixture_names: list[str],
    check_content: bool = False
) -> dict[str, Any]:
    """
    Validate that fixture files exist and optionally check JSON validity.

    Args:
        campaign_dir: Path to campaign directory
        fixture_names: List of fixture names (without .json extension)
        check_content: Whether to validate JSON content

    Returns:
        Dictionary with validation results per fixture
    """
    # TODO: Implement validation logic
    pass

# TODO: Create the tool
validate_fixtures_tool = StructuredTool.from_function(
    # Your implementation here
)
```

Solution:

```
class FixtureValidationInput(BaseModel):
    """Input schema for fixture validation."""
    fixture_names: list[str] = Field(
        description="List of fixture names to validate (e.g., ['impacts_in_target', 'tv_spot_schedule
    )
    check_content: bool = Field(
        default=False,
        description="Whether to validate JSON content (default: False)"
    )

def validate_fixtures(
    campaign_dir: Path,
    fixture_names: list[str],
    check_content: bool = False
) -> dict[str, Any]:
    """Validate fixture files."""
    fixtures_dir = campaign_dir / "01_pre_postprocessing" / "input_from_api"
    results = {}

    for fixture_name in fixture_names:
```

```
            fixture_path = fixtures_dir / f"{fixture_name}.json"

            if not fixture_path.exists():
                results[fixture_name] = {
                    "exists": False,
                    "error": f"File not found: {fixture_path}"
                }
                continue

            file_size = fixture_path.stat().st_size
            result = {
                "exists": True,
                "size_bytes": file_size,
                "path": str(fixture_path)
            }

            if check_content:
                try:
                    with open(fixture_path) as f:
                        json.load(f)
                    result["valid_json"] = True
                except json.JSONDecodeError as e:
                    result["valid_json"] = False
                    result["error"] = str(e)

            results[fixture_name] = result

        return {
            "total_checked": len(fixture_names),
            "results": results
        }

    validate_fixtures_tool = StructuredTool.from_function(
        func=lambda fixture_names, check_content=False, **_: validate_fixtures(
            campaign_dir,
            fixture_names=fixture_names,
            check_content=check_content
        ),
        name="validate_fixtures",
        description="Validate that fixture files exist and optionally check JSON validity",
        args_schema=FixtureValidationInput,
    )
```

Knowledge Check

Question 1: Schema Design

Which Pydantic pattern is correct for an optional field with a default value?

A) `field: str = None`

B) `field: str | None = Field(default=None)`

C) `field: Optional[str]`

D) `field: str = Field(required=False)`

▶ Answer

Question 2: LangGraph Compatibility

Why do we use `lambda **_:` when creating tools for LangGraph agents?

A) To make the code faster
B) To accept and ignore unexpected keyword arguments
C) To create anonymous functions
D) To pass multiple arguments

▶ Answer

Question 3: Validation

What happens when invalid input is passed to a Pydantic BaseModel?

A) Silent failure with None values
B) Automatic type coercion for all types
C) ValidationError exception with detailed error messages
D) Warning logged but execution continues

▶ Answer

Question 4: Tool Design

Which description is better for an LLM-facing tool parameter?

A) `"campaign name"`
B) `"Name of the campaign (e.g., 'Campaign 1')"`
C) `"str"`
D) `"campaign_name field"`

▶ Answer

Question 5: Implementation

What's wrong with this tool creation?

```
def my_function(param: str) -> str:
    return param.upper()

tool = StructuredTool.from_function(
    func=my_function,
    name="my_tool",
    description="Uppercase a string"
)
```

A) Missing args_schema
B) Missing **_ for LangGraph
C) Function signature is wrong
D) Nothing is wrong

▶ Answer

Summary

In this lesson, you learned:

1. **Pydantic BaseModel** provides runtime type validation with:
   - Automatic type coercion
   - Clear error messages
   - JSON Schema generation
   - Field descriptions for LLMs
2. **StructuredTool pattern** wraps Python functions as agent tools with:
   - Input validation via Pydantic schemas
   - Integration with LangChain's tool-calling infrastructure
   - Type safety and error handling
3. **LangGraph compatibility** requires:
   - `**_` pattern to accept unexpected kwargs
   - Explicit parameter extraction for LLM-provided inputs
   - Pre-bound parameters for contextual data (campaign_dir)
4. **Best practices** include:
   - Always provide Field descriptions
   - Use appropriate defaults for optional fields
   - Test schemas independently before tool integration
   - Keep schemas simple and focused

Next Steps

In **Lesson 3**, you'll learn:

- Implementing allowlist rules for table filtering
- Inclusion and exclusion logic patterns
- Edge case handling (notation variations)
- Testing filtering behavior

References

- Pydantic Documentation (v2.12+)
- LangChain StructuredTool API
- LangGraph Tool Patterns

- Python Type Hints (PEP 484)
- JSON Schema Specification

Code Reference

Complete working implementation: lesson_01_working_code.py (lines 414-424, 527-549)