# any-agent Tutorial

Welcome to the `any-agent` tutorial! In this first chapter, we're going to start with the very basics: setting up your agent. Think of this like choosing the settings for a new app or game – you decide how it should behave, what features it should use, and how you want to see what it's doing.

In `any-agent`, these settings are managed by special classes called **configuration objects**. The two main ones we'll look at are `AgentConfig` and `TracingConfig`.

## What's the Problem?

Imagine you want to create an AI agent that can answer questions. You might want it to:

1. Use a specific powerful language model (like GPT-4o or Claude 3 Opus).
2. Have a particular personality or set of instructions (e.g., "always be helpful and concise").
3. Maybe even have access to certain tools (like a calculator or a web search).

Different language models and different underlying agent technologies (called "frameworks" in `any-agent`, which we'll cover in Agent Frameworks) have different ways of being set up. How can `any-agent` provide a single, simple way for *you* to specify these things, no matter which model or framework you end up using?

## The Solution: Configuration Objects

This is where `AgentConfig` and `TracingConfig` come in. They act as a standard "settings form" that you fill out. You tell `any-agent` what you want by creating instances of these classes and setting their properties. `any-agent` then reads these settings to figure out how to build and run your agent.

Let's look at the key parts of these configuration objects.

### `AgentConfig`: The Agent's Core Settings

The `AgentConfig` class holds the fundamental settings for your agent. It's like telling the agent: "Here's who you are, what brain to use, and what you know."

Here are some of the most important settings you'll find in `AgentConfig`:

- **`model_id`**: This is crucial! It tells the agent *which* specific language model to use. Examples could be `"gpt-4o"`, `"claude-3-opus"`, `"gemini-1.5-pro"`, or even local models like `"llama3"`.
- **`instructions`**: This is where you give the agent its core directives or personality. You might say something like `"You are a friendly AI assistant that helps users write Python code."`

- `tools`: This is a list of capabilities the agent has. For example, you might give it a tool to search the web or run code. We'll dive deep into Tools in a later chapter.
- `api_base` and `api_key`: Sometimes, you need to provide specific addresses or keys to access the language model you chose in `model_id`. You can set them here.
- `name` and `description`: You can give your agent a name and a brief description for your own organization.

There are other settings too, but these are the ones you'll most likely use right away.

### `TracingConfig`: Seeing What Your Agent Does

When your agent is running, especially when it's using tools or interacting with the language model, it can be very helpful to see what's happening step-by-step. This is called **tracing**.

The `TracingConfig` class lets you control how this tracing works. It's like deciding how detailed you want the logs to be and what colors to use for different types of actions. We'll cover Tracing in more detail later.

Key settings in `TracingConfig`:

- `output_dir`: Where to save the trace information (usually in a file). If you set this to `None`, no trace file is saved.
- `llm`, `tool`, `agent`, `chain`: These settings let you specify colors for different types of events in the trace output, making it easier to read.

## Your First Configuration

Let's go back to our simple use case: creating an agent that uses GPT-4o and acts like a friendly assistant. Here's how you would create the configuration objects for that:

```python
from any_agent import AgentConfig, TracingConfig

# 1. Configure the agent's brain and instructions
agent_config = AgentConfig(
    model_id="gpt-4o",  # Specify the language model
    instructions="You are a friendly AI assistant that helps users write Python code.", # G
    # We won't add tools yet, that's for a later chapter!
)

# 2. Configure how you want to see the agent's actions (tracing)
tracing_config = TracingConfig(
    output_dir="my_agent_traces", # Save traces to a folder named 'my_agent_traces'
    llm="yellow", # Show language model calls in yellow
    tool="blue",  # Show tool calls in blue
)

print("Agent Configuration created:")
print(f"  Model ID: {agent_config.model_id}")
print(f"  Instructions: {agent_config.instructions}")

print("\nTracing Configuration created:")
print(f"  Output Directory: {tracing_config.output_dir}")
print(f"  LLM Trace Color: {tracing_config.llm}")
```

**Explanation:**

- We import `AgentConfig` and `TracingConfig` from the `any_agent` library.
- We create an instance of `AgentConfig`, providing the `model_id` and `instructions` we want.
- We create an instance of `TracingConfig`, telling it where to save traces and what colors to use.
- We then print out some of the settings to show that the objects were created correctly.

This code doesn't *run* the agent yet, but it successfully creates the "settings" that `any-agent` will need later.

## Under the Hood (Simplified)

So, what are these configuration objects really? Looking at the code in `src/any_agent/config.py`, you'll see they are defined using something called `pydantic.BaseModel`.

```
# From src/any_agent/config.py (simplified)
from pydantic import BaseModel, Field
from collections.abc import Sequence # Used for lists/tuples

# ... other classes like AgentFramework, TracingConfig ...

class AgentConfig(BaseModel):
    # model_config = ConfigDict(extra="forbid") # Don't worry about this for now
    model_id: str # This means model_id must be a string
    api_base: str | None = None # This means api_base can be a string or None
    api_key: str | None = None
    description: str = "" # Default value is an empty string
    name: str = "any_agent"
    instructions: str | None = None
    tools: Sequence[Tool] = Field(default_factory=list) # tools is a list, defaults to empt
    # ... other parameters ...

class TracingConfig(BaseModel):
    output_dir: str | None = "traces" # Default value is "traces"
    llm: str | None = "yellow"
    tool: str | None = "blue"
    agent: str | None = None
    chain: str | None = None
```

**Explanation:**

- `BaseModel` from `pydantic` is a library that helps define data structures with type hints (like `str` for text, `Sequence` for lists).
- When you create `AgentConfig(model_id="...", instructions="...")`, Pydantic checks if you provided the right types of values.
- These classes are essentially blueprints for creating objects that just hold data – your settings!

When you eventually create an actual agent using `any-agent`, you will pass these configuration objects to it. The `any-agent` library will then read these settings to understand how you want your agent to be built and behave.

Here's a simple diagram showing this flow:

```
sequenceDiagram
    participant User
    participant AgentConfig as Agent Config Object
    participant TracingConfig as Tracing Config Object
    participant AnyAgent as AnyAgent (Next Chapter!)

    User->>AgentConfig: Create AgentConfig(settings...)
    User->>TracingConfig: Create TracingConfig(settings...)
    User->>AnyAgent: Pass AgentConfig and TracingConfig
    AnyAgent-->>User: Agent is ready!
```

As you can see, the configuration objects are just the first step – preparing the instructions and settings before you actually create and use the agent itself.

## Conclusion

In this chapter, you learned that `any-agent` uses configuration objects, specifically `AgentConfig` and `TracingConfig`, to define how your agent should be set up and how its execution should be monitored. You saw how to create simple instances of these classes to specify the language model, instructions, and tracing preferences for your agent.

These configuration objects are the foundation. In the next chapter, we'll see how to take these settings and use them to create and interact with an actual agent using the main `AnyAgent` class.

Ready to build your agent? Let's go to the next chapter!

Chapter 2: AnyAgent (Unified Interface)

Welcome back! In Chapter 1: Agent Configuration, we learned how to set up the "settings" for our agent using `AgentConfig` and `TracingConfig`. We prepared the instructions, chose a language model, and decided how we want to see the agent's actions.

But how do we actually *create* and *talk* to the agent using those settings? That's where the `AnyAgent` class comes in.

## The Universal Remote Control

Imagine you have several different smart devices at home – a TV, a soundbar, a streaming box – all made by different companies. Each one might come with its own remote control, and they all work differently! It can be confusing to switch between them just to watch a movie.

Wouldn't it be great to have *one* universal remote control that can talk to *all* of them? You just press "Play" or "Volume Up" on the universal remote, and it figures out the right signal to send to the specific device you're using.

That's exactly what the `AnyAgent` class does for AI agent frameworks.

There are many powerful libraries and platforms for building AI agents (like LangChain, Smolagents, LlamaIndex, etc.). We call these "agent frameworks" in `any-agent`. Each framework has its own way of being set up, run, and interacted with.

`AnyAgent` acts as your **universal remote control**. Instead of learning the specific "buttons" and "signals" for each framework, you just learn how to use the `AnyAgent` remote. You tell `AnyAgent` *which* framework you want to use (like choosing which device the universal remote should control), give it the settings you prepared in Chapter 1, and then you can simply tell it to "run" a task. `AnyAgent` handles the complexity of talking to the specific framework behind the scenes.

## Using the `AnyAgent` Remote

The main way you interact with the `AnyAgent` universal remote is through its `create` method and then the `run` method of the object it gives you back.

Let's look at the key steps:

1. **Import `AnyAgent`:** Just like you imported the config classes, you need to import `AnyAgent`.
2. **Choose Your Framework:** Decide which agent framework you want to use for this specific agent. You specify this when creating the `AnyAgent`. We'll dive deeper into Agent Frameworks in the next chapter, but for now, you just need to know its name (like `"smolagents"` or `"langchain"`).
3. **Use Your Configuration:** Pass the `AgentConfig` and optional `TracingConfig` objects you created in Chapter 1 to the `create` method.

4. **Tell the Agent What to Do:** Use the `run` method to give the agent a prompt or a task.

### Creating Your First Agent

Let's take the configuration we made in Chapter 1 (using GPT-4o as a friendly Python assistant) and use it to create and run an agent.

First, let's quickly recreate the configuration objects:

```python
from any_agent import AgentConfig, TracingConfig

# Recreate the configuration from Chapter 1
agent_config = AgentConfig(
    model_id="gpt-4o",
    instructions="You are a friendly AI assistant that helps users write Python code.",
)

tracing_config = TracingConfig(
    output_dir="my_agent_traces",
    llm="yellow",
    tool="blue",
)

print("Configuration ready.")
```

Now, let's use `AnyAgent.create()` to build our agent. We'll choose the `"smolagents"` framework for this example (you could easily switch this string to `"langchain"`, `"openai"`, etc., if you had those installed and configured).

```python
from any_agent import AnyAgent # Import the AnyAgent class

# ... (previous code to create agent_config and tracing_config) ...

# Create the agent using the AnyAgent universal remote
# We tell it:
# 1. Which framework to use ("smolagents")
# 2. The agent's settings (agent_config)
# 3. The tracing settings (tracing_config)
agent = AnyAgent.create(
    "smolagents",      # Use the Smolagents framework
    agent_config,      # Pass our agent settings
    tracing=tracing_config # Pass our tracing settings (optional)
)

print(f"Agent created using the '{agent.framework.value}' framework!")
# The agent.framework property tells us which framework was actually used
```

**Explanation:**

- We import the `AnyAgent` class.
- We call the `AnyAgent.create()` method. This is a special type of method (a “class method”) that belongs to the `AnyAgent` class itself, not a specific agent instance. You use it to *make* an agent instance.
- We pass the framework name (`"smolagents"`), our `agent_config`, and our `tracing_config`.
- `AnyAgent.create()` does the work of setting up the chosen framework with your configuration and returns an `AnyAgent` object (or more accurately, an object that *acts* like an `AnyAgent` object, implementing the universal interface).
- We store this returned object in the `agent` variable. This `agent` variable is now our universal remote, ready to receive commands.

### Running Your Agent

Once you have the `agent` object, you can give it a task using the `run()` method.

```
# ... (previous code to create agent_config, tracing_config, and agent) ...

# Now, tell the agent to perform a task using the run() method
print("\nRunning the agent...")
response = agent.run("Write a simple Python function to add two numbers.")

print("\nAgent's Response:")
print(response)
```

**Explanation:**

- We call the `run()` method on our `agent` object, passing the prompt as a string.
- The `AnyAgent` object (which is internally using the Smolagents framework in this case) takes the prompt, processes it according to its instructions and capabilities (though we haven’t added tools yet!), and generates a response.
- The `run()` method waits for the agent to finish and returns the final response.
- We print the response.

The output you see will be the text generated by the language model (GPT-4o) based on your prompt and the instructions you gave it in `agent_config`. It should provide a Python function to add two numbers.

This simple `run()` method is the core of interacting with your agent, regardless of the underlying framework.

## Under the Hood: How the Universal Remote Works

So, what happens when you call `AnyAgent.create()` and then `agent.run()`?

Think back to the universal remote analogy. When you press "Power", the remote doesn't *become* the TV; it just knows how to send the right signal *to* the TV.

Similarly, the `AnyAgent` class is an **abstraction**. It defines a standard way to interact with *any* agent framework (`create`, `run`, etc.). When you call `AnyAgent.create("smolagents", ...)`, `AnyAgent` doesn't *become* Smolagents. Instead, it looks up which specific class handles the `"smolagents"` framework (there's a class like `SmolagentsAgent` internally), creates an instance of *that* specific class, and sets it up using your configuration.

The object returned by `AnyAgent.create()` is actually an instance of that framework-specific class (`SmolagentsAgent` in our example), but it's designed to have the same `run()` method as all other framework-specific classes. So, from your perspective, you're just interacting with an `AnyAgent`.

Here's a simplified sequence:

```
sequenceDiagram
    participant YourCode
    participant AnyAgentClass as AnyAgent.create()
    participant AnyAgentInstance as AnyAgent Object
    participant FrameworkAgent as Specific Framework Agent (e.g., SmolagentsAgent)

    YourCode->>AnyAgentClass: Call AnyAgent.create("smolagents", config...)
    AnyAgentClass->>AnyAgentClass: Look up "smolagents" handler
    AnyAgentClass->>FrameworkAgent: Create new SmolagentsAgent(config...)
    FrameworkAgent-->>AnyAgentClass: Return SmolagentsAgent instance
    AnyAgentClass-->>YourCode: Return SmolagentsAgent instance (as AnyAgent type)
    YourCode->>AnyAgentInstance: Call agent.run("...")
    AnyAgentInstance->>FrameworkAgent: Call framework-specific run method
    FrameworkAgent->>FrameworkAgent: Process prompt, use LLM, tools, etc.
    FrameworkAgent-->>AnyAgentInstance: Return result
    AnyAgentInstance-->>YourCode: Return result
```

Let's peek at the code in `src/any_agent/frameworks/any_agent.py` to see this structure (simplified):

```python
# From src/any_agent/frameworks/any_agent.py (simplified)
import asyncio
from abc import ABC, abstractmethod # ABC means Abstract Base Class
from typing import Any

from any_agent.config import AgentConfig, AgentFramework, TracingConfig
# ... other imports ...
# This is the blueprint for the universal remote interface
class AnyAgent(ABC): # It's an Abstract Base Class - you can't create an instance of THIS c
    """Base abstract class for all agent implementations.
    This provides a unified interface for different agent frameworks.
    """
    def __init__(
        self,
        config: AgentConfig,
        managed_agents: list[AgentConfig] | None = None,
    ):
        self.config = config
        self.managed_agents = managed_agents
        self._agent = None # This will hold the actual framework-specific agent!
        # ... other internal variables ...

    # This is the method you call to CREATE an agent
    @classmethod # This means you call it on the class itself (AnyAgent.create)
    def create(
        cls, # cls refers to the AnyAgent class
        agent_framework: AgentFramework | str, # Which framework?
        agent_config: AgentConfig, # Your settings
        managed_agents: list[AgentConfig] | None = None,
        tracing: TracingConfig | None = None, # Your tracing settings
    ) -> AnyAgent: # It returns an object that acts like an AnyAgent
        # Find the specific class for the chosen framework
        agent_cls = cls._get_agent_type_by_framework(agent_framework)

        # Create an instance of that specific class!
        agent = agent_cls(agent_config, managed_agents=managed_agents)

        # Set up tracing if configured
        if tracing is not None:
             # ... tracing setup logic ...
             pass # Simplified

        # Load the agent (framework-specific setup)
        asyncio.get_event_loop().run_until_complete(agent.load_agent())

        return agent # Return the framework-specific agent instance

    # This is the method you call to RUN the agent
    def run(self, prompt: str) -> Any:
        """Run the agent with the given prompt."""
        # It just calls the asynchronous version internally
        return asyncio.get_event_loop().run_until_complete(self.run_async(prompt))
    # This method MUST be implemented by each framework-specific class
    @abstractmethod # This means subclasses MUST provide their own version
    async def load_agent(self) -> None:
        """Load the agent instance (framework-specific)."""

    # This method MUST be implemented by each framework-specific class
    @abstractmethod # This means subclasses MUST provide their own version
    async def run_async(self, prompt: str) -> Any:
        """Run the agent asynchronously with the given prompt (framework-specific)."""

    # ... other methods and properties ...

    # This property is intentionally hidden to keep the abstraction clean
    @property
    def agent(self) -> Any:
        """The underlying agent implementation from the framework."""
        # This will raise an error if you try to access it directly
        msg = "Cannot access the 'agent' property..."
        raise NotImplementedError(msg)

    # Helper method to find the right class based on framework name
    @staticmethod
    def _get_agent_type_by_framework(
        framework_raw: AgentFramework | str,
    ) -> type[AnyAgent]:
        # ... logic to return SmolagentsAgent, LangchainAgent, etc. ...
        pass # Simplified
```

**Explanation:**

- The main `AnyAgent` class is an `ABC` (Abstract Base Class). It defines the *interface* (`create`, `run`, `load_agent`, `run_async`) but doesn't fully implement the core logic (`load_agent`, `run_async` are `@abstractmethod`).
- The `create` method is a `@classmethod`. It takes the framework name and configurations.
- Inside `create`, it uses `_get_agent_type_by_framework` to find the *actual* Python class responsible for the chosen framework (like `SmolagentsAgent`).
- It then creates an instance of *that specific framework class* (`agent = agent_cls(...)`). This instance is the one that knows how to talk to the real Smolagents library.
- It calls `agent.load_agent()` (which runs the framework-specific setup code).
- Finally, it returns this framework-specific instance. Because `SmolagentsAgent` (and `LangchainAgent`, etc.) are built to inherit from or implement the `AnyAgent` interface, they have the required `run()` method.

- When you call `agent.run()`, it's actually calling the `run()` method on the framework-specific object (`SmolagentsAgent` in our example), which in turn calls its own `run_async` method to interact with the underlying framework library.

This design means you only ever need to learn the `AnyAgent` interface (`create`, `run`, etc.). The library handles the translation to the specific framework's API for you.

## Conclusion

In this chapter, you learned that `AnyAgent` is the central class in the `any-agent` library, acting as a unified interface or "universal remote" for different agent frameworks. You saw how to use the `AnyAgent.create()` method to instantiate an agent using your configuration from Chapter 1 and how to use the `agent.run()` method to give the agent a task and get a response.

You also got a peek under the hood to understand that `AnyAgent.create()` selects and sets up a framework-specific agent class behind the scenes, allowing you to interact with it through a consistent interface.

Now that you know how to create and run an agent using the `AnyAgent` interface, the next logical step is to understand more about the different Agent Frameworks that `any-agent` supports and how they fit into the picture.

Chapter 3: Agent Frameworks

Welcome back! In Chapter 1: Agent Configuration, we learned how to set up the "settings" for our agent using `AgentConfig` and `TracingConfig`. In Chapter 2: AnyAgent (Unified Interface), we saw how the `AnyAgent` class acts as a "universal remote control" to create and run an agent using those settings.

But what exactly is `AnyAgent` controlling? What are the different "devices" it can talk to? That's where **Agent Frameworks** come in.

## The Engines Behind the Agent

Think back to our universal remote analogy. The remote itself doesn't *do* the work of showing a picture or playing sound. It sends signals to a specific device – like a Sony TV, a Samsung soundbar, or an LG streaming box – and *that device* performs the action.

In `any-agent`, the **Agent Frameworks** are these underlying "devices" or "engines" that actually power your agent. They are libraries or platforms specifically designed for building and running AI agents. Examples include LangChain, Smolagents, Google's ADK, and others.

Why are there different frameworks? * **Different Philosophies:** Some frameworks might focus on simple, quick agent creation, while others might be designed for complex multi-agent systems or specific types of tasks. * **Different**

**Features:** Each framework might offer unique features, optimizations, or ways of handling things like memory, tool use, or interaction patterns. * **Different Communities:** They are developed by different groups and have different ecosystems of tools and integrations built around them.

If you wanted to use a LangChain agent for one task and a Smolagents agent for another, you would normally have to learn the specific ways each library works – how to initialize their agents, how to pass prompts, how to handle responses, etc. This is like needing a different remote for every device!

`any-agent` solves this by providing the `AnyAgent` universal interface. You choose which framework you want to use, and `AnyAgent` handles the specific setup and communication needed for *that* framework, presenting you with the same simple `run()` method no matter which engine is under the hood.

## Choosing Your Framework

When you used `AnyAgent.create()` in the last chapter, you might have noticed the first parameter: the framework name (like `"smolagents"`). This is where you tell `AnyAgent` which engine you want to use.

```
from any_agent import AnyAgent, AgentConfig, TracingConfig

# Recreate a simple config
agent_config = AgentConfig(
    model_id="gpt-4o",
    instructions="You are a helpful assistant.",
)

tracing_config = TracingConfig(output_dir=None) # No tracing for this example

# --- This is where you choose the framework ---
framework_name = "smolagents" # Or "langchain", "google", "openai", etc.
# ---------------------------------------------

print(f"Attempting to create agent using the '{framework_name}' framework...")

try:
    # Create the agent using the chosen framework
    agent = AnyAgent.create(
        framework_name,
        agent_config,
        tracing=tracing_config
    )

    print(f"Agent successfully created using the '{agent.framework.value}' framework!")
    # You can now run the agent using the universal run() method
    # response = agent.run("What is the capital of France?")
    # print(f"\nAgent Response: {response}")

except ImportError as e:
    print(f"\nError: Could not create agent with framework '{framework_name}'.")
    print(f"Details: {e}")
    print(f"Make sure you have installed the necessary dependencies, e.g., `pip install 'any
except ValueError as e:
     print(f"\nError: Invalid framework name '{framework_name}'.")
     print(f"Details: {e}")
     # You could list valid frameworks here if needed
except Exception as e:
    print(f"\nAn unexpected error occurred: {e}")
```

**Explanation:**

- We import `AnyAgent` and our configuration classes.
- We create simple `AgentConfig` and `TracingConfig` objects.
- We define a variable `framework_name` and set it to the string name of the framework we want (e.g., `"smolagents"`).
- We pass this `framework_name` string as the first argument to `AnyAgent.create()`.
- `AnyAgent.create()` looks up the framework name and sets up the agent using the corresponding framework library.
- We include a `try...except` block because you need to have the specific framework library installed (like `smolagents` or `langchain`) for `any-agent` to be able to use it. `any-agent` uses "extra" dependencies for this, which you install like `pip install 'any-agent[smolagents]'`.

The `agent.framework.value` property (which we print) confirms which framework was successfully loaded. This property comes from the `AgentFramework` Enum defined in `src/any_agent/config.py`:

```
# From src/any_agent/config.py (simplified)
from enum import Enum, auto

class AgentFramework(str, Enum):
    GOOGLE = auto()
    LANGCHAIN = auto()
    LLAMA_INDEX = auto()
    OPENAI = auto()
    AGNO = auto()
    SMOLAGENTS = auto()

    @classmethod
    def from_string(cls, value: str | Self) -> Self:
        # ... logic to convert string to Enum member ...
        pass
```

This `AgentFramework` Enum lists all the frameworks that `any-agent` currently supports. When you pass a string like `"smolagents"` to `AnyAgent.create()`, `any-agent` uses the `from_string` method (or similar internal logic) to find the corresponding `AgentFramework.SMOLAGENTS` value.

## Under the Hood: Mapping Frameworks to Code

How does `AnyAgent.create()` know which framework corresponds to which code?

When you call `AnyAgent.create("smolagents", ...)`, here's a simplified view of what happens:

1. `AnyAgent.create()` receives the string `"smolagents"`.
2. It looks up this string in an internal mapping (or uses the `AgentFramework` Enum) to find the specific Python class that handles the Smolagents framework. Let's imagine this class is called `SmolagentsAgent`.
3. It creates an instance of this `SmolagentsAgent` class, passing your `AgentConfig` and `TracingConfig`.
4. The `SmolagentsAgent` instance then uses the actual `smolagents` library internally to build the agent according to your configuration.
5. `AnyAgent.create()` returns this `SmolagentsAgent` instance.

Because `SmolagentsAgent` (and `LangchainAgent`, `GoogleAgent`, etc.) are all built to follow the `AnyAgent` interface (they inherit from the base `AnyAgent` class and implement its required methods like `load_agent` and `run_async`), they can all be treated the same way from your perspective.

Here's a simplified sequence diagram:

```
sequenceDiagram
    participant YourCode
    participant AnyAgentClass as AnyAgent.create()
    participant FrameworkLookup as Internal Lookup
    participant SmolagentsAgentClass as SmolagentsAgent Class
    participant SmolagentsAgentInstance as SmolagentsAgent Object

    YourCode->>AnyAgentClass: Call AnyAgent.create("smolagents", config...)
    AnyAgentClass->>FrameworkLookup: Find class for "smolagents"
    FrameworkLookup-->>AnyAgentClass: Return SmolagentsAgent Class
    AnyAgentClass->>SmolagentsAgentClass: Create new instance(config...)
    SmolagentsAgentClass->>SmolagentsAgentInstance: Initialize
    SmolagentsAgentInstance->>SmolagentsAgentInstance: Call load_agent() (uses smolagents li
    SmolagentsAgentInstance-->>AnyAgentClass: Return initialized instance
    AnyAgentClass-->>YourCode: Return SmolagentsAgent instance (as AnyAgent type)
```

Let's look at the structure of one of the framework handler files, like `src/any_agent/frameworks/smolagents.py` (simplified):

```python
# From src/any_agent/frameworks/smolagents.py (simplified)
from typing import Any

from any_agent.config import AgentConfig, AgentFramework
from any_agent.frameworks.any_agent import AnyAgent # Import the base class

try:
    import smolagents # Import the actual framework library
    smolagents_available = True
except ImportError:
    smolagents_available = False

# This class handles the Smolagents framework
class SmolagentsAgent(AnyAgent): # It inherits from the base AnyAgent class
    """Smolagents agent implementation."""

    @property
    def framework(self) -> AgentFramework:
        # This property tells AnyAgent which framework this class handles
        return AgentFramework.SMOLAGENTS

    # This method is called by AnyAgent.create() to set up the framework agent
    async def load_agent(self) -> None:
        if not smolagents_available:
             # Raise error if the library isn't installed
            msg = "You need to `pip install 'any-agent[smolagents]'`..."
            raise ImportError(msg)

        # --- This is where the actual smolagents library is used ---
        # Example: Get the model config specific to smolagents
        smolagents_model = self._get_model(self.config)

        # Example: Load tools (handled by AnyAgent base class)
        tools, _ = await self._load_tools(self.config.tools)

        # Example: Create the actual smolagents agent instance
        main_agent_type = getattr(smolagents, self.config.agent_type or "CodeAgent")
        self._agent = main_agent_type(
            name=self.config.name,
            model=smolagents_model,
            tools=tools,
            # ... other smolagents specific parameters ...
        )
        # ... set instructions etc. ...
        # -----------------------------------------------------------

    # This method is called by agent.run() to execute the task
    async def run_async(self, prompt: str) -> Any:
        # --- This calls the run method of the actual smolagents agent ---
        return self._agent.run(prompt)
        # ---------------------------------------------------------------

    # ... other helper methods specific to Smolagents ...
```

**Explanation:**

- The `SmolagentsAgent` class inherits from the base `AnyAgent` class. This means it promises to implement the required methods.
- The `@property framework` tells the `AnyAgent.create()` method which `AgentFramework` value this class corresponds to.
- The `load_agent()` method contains the specific code needed to initialize an agent using the `smolagents` library, translating the generic `AgentConfig` into parameters the `smolagents` library understands.
- The `run_async()` method contains the specific code needed to run the agent using the `smolagents` library's method (in this case, `self._agent.run(prompt)`).

Similar files exist for `langchain.py`, `google.py`, `openai.py`, etc., each containing the specific code to interface with *that* particular framework library, but all presenting the same `AnyAgent` interface to you.

This is the power of the `any-agent` abstraction: you configure your agent once using `AgentConfig`, and then you can swap out the underlying framework just by changing a string in `AnyAgent.create()`, without needing to rewrite your core logic for running the agent.

## Conclusion

In this chapter, you learned that Agent Frameworks are the different underlying libraries or "engines" that `any-agent` can use to power your agent. You saw how `AnyAgent` acts as a universal interface, allowing you to choose a framework by name when creating an agent with `AnyAgent.create()`. You also got a glimpse into how `any-agent` maps these framework names to specific code handlers that interact with the actual framework libraries behind the scenes, providing a consistent `run()` method regardless of the chosen engine.

Now that you understand the role of frameworks, the next crucial piece of building capable agents is giving them abilities beyond just talking. In the next chapter, we'll explore Tools – the functions and services your agent can use to interact with the world.

Chapter 4: Tools

Welcome back! In our journey so far, we've learned how to set up our agent's basic "settings" using Chapter 1: Agent Configuration, how to use the `AnyAgent` class as a "universal remote" to create and run agents via a unified interface in Chapter 2: AnyAgent (Unified Interface), and we explored the different "engines" or Chapter 3: Agent Frameworks that `any-agent` can use.

But an agent that can only *think* and *talk* based on its internal knowledge is limited. What if it needs up-to-date information? What if it needs to perform an action in the real world?

This is where **Tools** come in.

## Giving Your Agent Abilities

Imagine your agent is a brilliant student who knows a lot, but is stuck in a room. They can answer questions based on what they've already learned, but they can't look up new information, use a calculator, or interact with anything outside the room.

To make them truly useful, you need to give them ways to interact with the outside world. These "ways to interact" are **Tools**.

In `any-agent`, **Tools** are the capabilities you give your agent to perform specific actions. They are like the special buttons on your universal remote that do more than just change channels – they might open the TV's menu, switch input sources, or even control a connected soundbar.

Tools allow your agent to: * **Gather Information:** Search the web, read a file, visit a webpage. * **Perform Actions:** Send an email, run code, interact with a service. * **Ask for Help:** Ask the user for clarification or verification.

By giving your agent tools, you empower it to go beyond just generating text and actually *do* things to achieve its goals.

## Using Pre-built Tools

`any-agent` comes with some useful tools already built-in. Let's look at a common one: searching the web.

**Use Case:** You want your agent to be able to answer questions that require current information, like "What is the weather forecast in London tomorrow?"

An agent without tools can't answer this accurately because its internal knowledge is static. It needs a tool to look up the weather online.

`any-agent` provides a `search_web` tool for this exact purpose. It's a simple Python function that uses a search engine (DuckDuckGo by default) and returns the results.

To give your agent this tool, you add it to the `tools` list in your `AgentConfig`:

```
from any_agent import AgentConfig, AnyAgent
from any_agent.tools import search_web # Import the pre-built tool

# 1. Configure the agent, including the tool
agent_config = AgentConfig(
    model_id="gpt-4o", # Or another model that supports tools
    instructions="You are a helpful assistant that can search the web.",
    tools=[search_web], # Add the search_web function to the tools list
)

# 2. Create the agent (using any framework, e.g., smolagents)
# Make sure you have the framework installed: pip install 'any-agent[smolagents]'
try:
    agent = AnyAgent.create("smolagents", agent_config)
    print(f"Agent created with framework: {agent.framework.value}")

    # 3. Run the agent with a query that requires the tool
    print("\nRunning agent with web search query...")
    response = agent.run("What is the current weather in London?")

    print("\nAgent's Response:")
    print(response)

except ImportError as e:
    print(f"\nError: Could not create agent. Make sure you have the framework installed.")
    print(f"Details: {e}")
except Exception as e:
    print(f"\nAn unexpected error occurred: {e}")
```

**Explanation:**

- We import `AgentConfig`, `AnyAgent`, and the specific tool function `search_web`.
- In `AgentConfig`, we add `search_web` to the `tools` list. This tells `any-agent` (and the underlying framework) that this agent *has* the capability represented by the `search_web` function.
- We create the agent using `AnyAgent.create()`, passing the configuration with the tool.
- We call `agent.run()` with a query that the agent can likely only answer by using the `search_web` tool.

When you run this code, the agent (powered by the framework) will receive the prompt. It will analyze the prompt and realize it needs external information. It will then look at the tools it has available (the `search_web` tool) and decide to use it. The framework will call the actual `search_web` function, get the search results, and provide those results back to the language model. The language model will then use the search results to formulate the final answer to your question about the weather.

Other pre-built tools in `any_agent.tools` include `visit_webpage` (to read the content of a URL), and simple user interaction tools like `ask_user_verification` or `send_console_message`.

## Creating Your Own Tools

The real power comes from being able to give your agent *any* capability you can write as a Python function. You can create custom tools for interacting with your database, calling an internal API, controlling hardware, or anything else!

To create your own tool, you just need to write a standard Python function. However, there are a couple of **very important requirements** so that the agent (specifically, the language model) knows how and when to use your tool:

1. **Docstring:** Your function *must* have a clear docstring that explains what the tool does. This description is what the language model sees to decide if the tool is relevant to the current task.
2. **Type Hints:** Your function's arguments and return value *must* have type hints (like `str`, `int`, `bool`, `list[str]`, etc.). This tells the language model what kind of input the tool expects and what kind of output it will produce.

Let's create a simple custom tool: a function that adds two numbers.

```python
# Define your custom tool function
def add_numbers(a: float, b: float) -> float:
    """Adds two numbers together and returns the sum.

    Args:
        a: The first number.
        b: The second number.

    Returns:
        The sum of a and b.
    """
    print(f"\n--- Tool Call: add_numbers({a}, {b}) ---") # Optional: Add logging
    return a + b

# Now, add this custom tool to your AgentConfig
from any_agent import AgentConfig, AnyAgent

agent_config = AgentConfig(
    model_id="gpt-4o", # Or another model that supports tools
    instructions="You are a helpful assistant that can perform basic arithmetic.",
    tools=[add_numbers], # Add your custom function to the tools list
)
```

```python
# Create and run the agent
# Make sure you have the framework installed: pip install 'any-agent[smolagents]'
try:
    agent = AnyAgent.create("smolagents", agent_config)
    print(f"Agent created with framework: {agent.framework.value}")

    print("\nRunning agent with arithmetic query...")
    response = agent.run("What is 123 + 456?")

    print("\nAgent's Response:")
    print(response)

except ImportError as e:
    print(f"\nError: Could not create agent. Make sure you have the framework installed.")
    print(f"Details: {e}")
except Exception as e:
    print(f"\nAn unexpected error occurred: {e}")
```

**Explanation:**

- We define the `add_numbers` function with a docstring explaining its purpose and type hints for its arguments (`a: float`, `b: float`) and return value (`-> float`).
- We add this function object (`add_numbers`) directly to the `tools` list in `AgentConfig`.
- When we run the agent with a query like "What is 123 + 456?", the agent's language model will see the prompt and the description of the `add_numbers` tool. It will understand that this tool is relevant.
- The language model will then decide to call the tool, figuring out the arguments (`a=123`, `b=456`) from the prompt.
- The framework will execute your `add_numbers` function with those arguments.
- The function will return `579`.
- The framework will give `579` back to the language model.
- The language model will use this result to formulate the final answer, likely stating that 123 + 456 is 579.

This ability to add custom functions as tools is incredibly powerful, allowing you to connect your agent to virtually any service or capability.

## Under the Hood: How Tools Work

So, what happens when you give a list of functions to `AgentConfig` and run the agent?

When you call `AnyAgent.create()` with an `AgentConfig` that includes tools:

1. `AnyAgent.create()` selects the appropriate framework handler (like `SmolagentsAgent` for the `"smolagents"` framework) based on your choice (as discussed in Chapter 3: Agent Frameworks).
2. The framework handler's `load_agent()` method is called. This method is responsible for setting up the agent using the specific framework library.
3. Inside `load_agent()`, the handler calls a helper function (like `any_agent.tools.wrappers.wrap_tools`) to process the list of tools you provided.
4. `wrap_tools` does two main things for each tool (if it's a simple function):
    - It **verifies** that the function has a docstring and type hints using `verify_callable`. If not, it raises an error because the agent wouldn't be able to understand how to use it.
    - It **wraps** the function using a framework-specific wrapper (like `wrap_tool_smolagents` for Smolagents). This wrapper converts your standard Python function into an object format that the specific framework library expects for its tools.
5. The framework handler then initializes the actual agent instance from the framework library, passing the list of these *wrapped* tools.
6. When you call `agent.run(prompt)`:
    - The prompt goes to the framework agent.
    - The framework agent sends the prompt and the descriptions of the available tools (extracted from the docstrings and type hints) to the language model.
    - The language model processes this information and decides on the next step. If it determines that a tool is needed to fulfill the prompt, it outputs a special signal indicating which tool to call and with what arguments (based on the type hints).
    - The framework intercepts this signal.
    - The framework calls the corresponding *wrapped* tool function with the arguments provided by the LLM.
    - Your original Python function (e.g., `search_web` or `add_numbers`) is executed.
    - The result of your function is returned to the framework.
    - The framework gives this result back to the language model.
    - The language model uses the tool's result to continue processing the prompt and generate the final response.

Here's a simplified sequence diagram:

```
sequenceDiagram
    participant User
    participant YourCode
    participant AnyAgentInstance as AnyAgent Object
    participant FrameworkAgent as Specific Framework Agent
    participant ToolWrapper as any_agent.tools.wrappers
    participant LLM as Language Model
    participant YourTool as Your Tool Function
    User->>YourCode: Call agent.run(prompt)
    YourCode->>AnyAgentInstance: Call run(prompt)
    AnyAgentInstance->>FrameworkAgent: Call framework-specific run
    FrameworkAgent->>LLM: Send prompt + Tool Descriptions
    LLM->>LLM: Decide to use a tool
    LLM-->>FrameworkAgent: Output: Call ToolX(arg1, arg2)
    FrameworkAgent->>ToolWrapper: Find/Call Wrapped ToolX
    ToolWrapper->>YourTool: Call YourTool(arg1, arg2)
    YourTool-->>ToolWrapper: Return Result
    ToolWrapper-->>FrameworkAgent: Return Result
    FrameworkAgent->>LLM: Send Tool Result
    LLM->>LLM: Process Result, Formulate Response
    LLM-->>FrameworkAgent: Output: Final Response
    FrameworkAgent-->>AnyAgentInstance: Return Final Response
    AnyAgentInstance-->>YourCode: Return Final Response
    YourCode-->>User: Display Response
```

Let's look at a simplified snippet from `src/any_agent/tools/wrappers.py`:

```python
# From src/any_agent/tools/wrappers.py (simplified)
import inspect
from collections.abc import Callable, Sequence
from typing import Any

from any_agent.config import AgentFramework, Tool # Tool type alias includes Callable
# ... other imports and framework-specific wrappers ...

# This function checks if your tool function has docstrings and type hints
def verify_callable(tool: Callable[..., Any]) -> None:
    signature = inspect.signature(tool)
    if not tool.__doc__:
        msg = f"Tool {tool.__name__} needs a docstring..."
        raise ValueError(msg)
    if signature.return_annotation is inspect.Signature.empty:
        msg = f"Tool {tool.__name__} needs a return type..."
        raise ValueError(msg)
    for param in signature.parameters.values():
        if param.annotation is inspect.Signature.empty:
            msg = f"Tool {tool.__name__} needs typed arguments..."
            raise ValueError(msg)

# This function is called by the framework handler to process your tools
async def wrap_tools(
    tools: Sequence[Tool], # The list of tools from AgentConfig
    agent_framework: AgentFramework, # The chosen framework
) -> tuple[list[Tool], list[Any]]: # Returns wrapped tools and MCP servers (more on MCP lat
```

```python
    wrapper = WRAPPERS[agent_framework] # Get the correct framework-specific wrapper

    wrapped_tools = list[Tool]()
    mcp_servers = list[Any]() # Simplified, holds MCP related objects

    for tool in tools:
        # Check if it's a simple callable function
        if callable(tool):
            verify_callable(tool) # Check docstring and type hints
            wrapped_tools.append(wrapper(tool)) # Wrap it for the specific framework
        # ... handle other tool types like MCPParams (covered in next chapter) ...
        else:
             msg = f"Tool {tool} needs to be a function or MCPParams..."
             raise ValueError(msg)

    return wrapped_tools, mcp_servers

# Example of a framework-specific wrapper (simplified)
def wrap_tool_smolagents(tool: Tool) -> Any:
    # This function takes your callable and converts it
    # into the format that the smolagents library expects for a tool.
    # It uses smolagents' internal tool wrapping logic.
    from smolagents import tool as smolagents_tool
    return smolagents_tool(tool) # Calls the smolagents library's wrapper
```

**Explanation:**

- `wrap_tools` iterates through the list of tools from your `AgentConfig`.
- For each tool that is a Python function (`callable(tool)`), it calls `verify_callable` to ensure it meets the requirements.
- It then uses the correct framework-specific wrapper function (like `wrap_tool_smolagents`) from the `WRAPPERS` dictionary.
- These wrapper functions (like `wrap_tool_smolagents`) call the underlying framework library's own tool wrapping logic to create an object that the framework can understand and execute.
- The list of these wrapped tools is returned and used by the framework handler to initialize the agent.

This process ensures that your simple Python functions are correctly presented to the chosen agent framework, allowing the language model to understand their capabilities and use them effectively.

## Conclusion

In this chapter, you learned that Tools are essential capabilities that allow your agent to interact with the outside world, going beyond static knowledge. You saw how to add both pre-built tools like `search_web` and your own custom Python functions to your agent's configuration using the `tools` list in `AgentConfig`. You also learned the crucial requirements for custom tool functions: a clear docstring and type hints, which the agent uses to understand the tool. Finally, you got a peek under the hood at how `any-agent` and the framework wrappers process and prepare your tools for the language model to use.

Tools are fundamental to building capable agents. In the next chapter, we'll dive into a specific, powerful type of tool called MCP Tools, which allow agents to interact with external processes and services in a standardized way.

Chapter 5: MCP Tools

Welcome back! In Chapter 4: Tools, we learned how to give our agent capabilities by providing it with Python functions. We saw how to add pre-built tools like web search and how to create our own custom functions for tasks like adding numbers.

These simple function-based tools are great, but they have a limitation: they run *within* the same Python process as your agent. What if your tool is a complex service that needs to run separately? What if it's a long-running process? What if you want multiple agents or other applications to share the same set of tools provided by a central service?

Running everything in one process isn't always ideal. This is where **MCP Tools** come in.

## Connecting to External Tool Services

Imagine you have a powerful, dedicated service running somewhere else – maybe it's a service that manages your company's database, or a service that can control smart devices in your home, or a service that provides a suite of specialized data analysis functions. You want your agent to be able to use the capabilities offered by this external service.

You could try to write a simple Python function that connects to this service, but managing the connection, handling errors, and ensuring the agent knows *which* specific tools are available from that service can get complicated quickly.

This is the problem that **MCP (Model Context Protocol)** and `any-agent`'s **MCP Tools** solve.

MCP is a standard way for an AI agent (or any application that needs tools) to communicate with a separate process or service that *provides* tools. Think of it like a standardized "tool server" protocol. The external service acts as an MCP server, listing the tools it offers and waiting for requests. Your agent, using `any-agent`'s MCP tool capabilities, acts as an MCP client, connecting to the server and asking it to execute tools.

`any-agent`'s **MCP Tools** are the configuration objects you use to tell your agent *how to connect* to one of these external MCP servers.

## Configuring an MCP Tool Connection

Instead of adding a Python function directly to your `AgentConfig`'s `tools` list, you add a special configuration object that describes the MCP connection. `any-agent` provides two main types of these objects, based on how the external MCP server is running:

1. **`MCPStdioParams`**: Use this if the external tool service is a command-line program that communicates over standard input and output (stdio).
2. **`MCPSseParams`**: Use this if the external tool service is a web server that communicates using Server-Sent Events (SSE) over HTTP.

Let's look at how you would configure a connection to an external MCP server running as a command-line program using `MCPStdioParams`.

```python
from any_agent import AgentConfig, AnyAgent
from any_agent.config import MCPStdioParams # Import the config object

# Imagine you have an external program called 'my-mcp-server'
# that provides tools via stdio.

# 1. Configure the MCP tool connection
# This object tells any-agent how to start and talk to the external server
mcp_stdio_tool_config = MCPStdioParams(
    command="my-mcp-server", # The command to run the external server
    args=["--port", "12345"], # Any command-line arguments for the server
    tools=["tool_a", "tool_b"] # (Optional) List specific tools to expose from this server
)

# 2. Add this MCP configuration object to your agent's tools list
agent_config = AgentConfig(
    model_id="gpt-4o", # Or another model that supports tools
    instructions="You can use tools provided by an external service.",
    tools=[
        # You can mix regular function tools and MCP tools!
        # search_web, # Add a regular tool if needed
        mcp_stdio_tool_config # Add the MCP connection config
    ],
)

# 3. Create the agent (using any framework that supports MCP, e.g., smolagents)
# Make sure you have the framework and mcp dependencies installed:
# pip install 'any-agent[smolagents,mcp]'
try:
    print("Configuring agent with MCP tool connection...")
    agent = AnyAgent.create("smolagents", agent_config)
    print(f"Agent created with framework: {agent.framework.value}")

    # 4. Now, if the external 'my-mcp-server' was running and providing tools
    # named 'tool_a' and 'tool_b', the agent could potentially use them
    # via the standard agent.run() method.

    # Example (conceptual - this won't work without the actual server running):
    # print("\nRunning agent with a query that might use an MCP tool...")
    # response = agent.run("Use tool_a to process this data: ...")
    # print("\nAgent's Response:")
    # print(response)

    print("\nAgent configured with MCP tool. Ready to interact if server is running.")

except ImportError as e:
    print(f"\nError: Could not create agent. Make sure you have the framework and mcp insta
    print(f"Details: {e}")
except ValueError as e:
     print(f"\nError: Configuration error: {e}")
except Exception as e:
    print(f"\nAn unexpected error occurred: {e}")
```

**Explanation:**

- We import `MCPStdioParams` from `any_agent.config`.
- We create an instance of `MCPStdioParams`. We provide the `command` that would start the external MCP server program and any `args` it needs.
- The optional `tools` list within `MCPStdioParams` lets you specify *which* tools from the external server you want this specific agent to have access to. If you omit this, the agent might see *all* tools the server offers (depending on the framework and server implementation).
- We add this `mcp_stdio_tool_config` object to the `tools` list in our `AgentConfig`.
- When we create the agent with `AnyAgent.create()`, `any-agent` sees the `MCPStdioParams` object in the tools list. It understands that this isn't a function to wrap, but a description of an external connection.
- The framework handler (e.g., `SmolagentsAgent`) uses this configuration to set up the MCP client connection to the external server *before* the agent is ready to run.
- Once the connection is established, the framework queries the external server via the MCP protocol to discover the tools it provides (their names, descriptions, and parameters).
- These discovered tools are then presented to the agent's language model, just like regular function tools.
- When you call `agent.run()` with a prompt, the language model can de-

cide to use one of the tools provided by the external MCP server. The framework intercepts the tool call, sends the request over the MCP connection to the external server, waits for the result, and gives the result back to the language model.

Using `MCPSseParams` is similar, but instead of `command` and `args`, you provide the `url` of the SSE endpoint and optional `headers`:

```
from any_agent.config import MCPSseParams

# Configure an SSE-based MCP tool connection
mcp_sse_tool_config = MCPSseParams(
    url="http://localhost:8080/mcp/tools", # The URL of the external SSE server
    headers={"Authorization": "Bearer your_token"}, # Optional headers
    tools=["tool_c"] # (Optional) Filter tools
)

# Add mcp_sse_tool_config to your AgentConfig's tools list...
# ... then create and run the agent as shown in the previous example.
```

This allows your agent to connect to MCP servers running as standard web services.

## Under the Hood: Managing the Connection

When you add `MCPParams` to your `AgentConfig` and call `AnyAgent.create()`, here's a simplified view of what happens:

1. `AnyAgent.create()` selects the appropriate framework handler (like `SmolagentsAgent`).
2. The framework handler's `load_agent()` method is called.
3. Inside `load_agent()`, the handler processes the `tools` list from `AgentConfig`.
4. When it encounters an `MCPParams` object (either `MCPStdioParams` or `MCPSseParams`), it doesn't wrap a function. Instead, it recognizes this as a request to set up an MCP connection.
5. It uses framework-specific MCP client code (often provided by separate libraries like `mcp` or framework-specific adapters) to establish the connection based on the parameters in the `MCPParams` object.
6. This involves starting the external process (`MCPStdioParams`) or connecting to the URL (`MCPSseParams`).
7. Once the connection is active, the framework's MCP client component communicates with the external MCP server to discover the available tools and their specifications (name, description, parameters).
8. These discovered tools are then added to the list of capabilities the agent's language model is aware of.
9. When the agent needs to use one of these tools during `agent.run()`, the framework sends the tool call request over the established MCP connection

to the external server.
10. The external server executes the tool and sends the result back over the MCP connection.
11. The framework receives the result and provides it to the language model.

This process is managed by specialized classes within `any-agent`'s `tools.mcp` module, like `MCPServerBase` and its framework-specific subclasses (`SmolagentsMCPServer`, `LangchainMCPServer`, etc.). These classes handle the connection setup and teardown.

Here's a simplified sequence diagram focusing on the setup:

```
sequenceDiagram
    participant YourCode
    participant AnyAgentClass as AnyAgent.create()
    participant FrameworkAgent as Specific Framework Agent
    participant MCPServerHandler as Framework MCPServer Handler (e.g., SmolagentsMCPServer)
    participant ExternalMCP as External MCP Server Process/Service

    YourCode->>AnyAgentClass: Call AnyAgent.create(..., config...)
    AnyAgentClass->>FrameworkAgent: Create FrameworkAgent(config...)
    FrameworkAgent->>FrameworkAgent: Call load_agent()
    FrameworkAgent->>MCPServerHandler: Create MCPServerHandler(mcp_tool_config)
    MCPServerHandler->>MCPServerHandler: Call setup_tools()
    alt MCPStdioParams
        MCPServerHandler->>ExternalMCP: Start process (command, args)
        ExternalMCP-->>MCPServerHandler: Establish StdIO connection
    else MCPSseParams
        MCPServerHandler->>ExternalMCP: Connect to URL (SSE)
        ExternalMCP-->>MCPServerHandler: Establish SSE connection
    end
    MCPServerHandler->>ExternalMCP: Discover available tools (MCP Protocol)
    ExternalMCP-->>MCPServerHandler: Send tool descriptions
    MCPServerHandler-->>FrameworkAgent: Return discovered tools
    FrameworkAgent->>FrameworkAgent: Add discovered tools to agent's capabilities
    FrameworkAgent-->>AnyAgentClass: Return initialized agent
    AnyAgentClass-->>YourCode: Return agent object
```

Let's look at a simplified snippet from `src/any_agent/tools/mcp/mcp_server_base.py` and one of the framework handlers, like `src/any_agent/tools/mcp/smolagents.py`:

```python
# From src/any_agent/tools/mcp/mcp_server_base.py (simplified)
from abc import ABC, abstractmethod
from any_agent.config import MCPParams, MCPSseParams, MCPStdioParams, Tool

class MCPServerBase(ABC):
    """Base class for MCP tools managers across different frameworks."""
    def __init__(self, mcp_tool: MCPParams, libraries: str = "any-agent[mcp]", mcp_available
        # ... checks for library availability ...
        self.mcp_tool = mcp_tool # Store the MCP config object
        self.tools: Sequence[Tool] = [] # List to hold discovered tools

    async def setup_tools(self) -> None:
        """Set up tools based on MCP config type."""
        match self.mcp_tool: # Check if it's StdIO or SSE
            case MCPStdioParams():
                await self.setup_stdio_tools() # Call specific setup method
            case MCPSseParams():
                await self.setup_sse_tools() # Call specific setup method

    @abstractmethod
    async def setup_sse_tools(self) -> None:
        """Implemented by subclasses to set up SSE connection."""
        pass

    @abstractmethod
    async def setup_stdio_tools(self) -> None:
        """Implemented by subclasses to set up StdIO connection."""
        pass

# From src/any_agent/tools/mcp/smolagents.py (simplified)
from any_agent.config import MCPParams, MCPSseParams, MCPStdioParams
from .mcp_server_base import MCPServerBase
# ... imports for smolagents and mcp libraries ...

class SmolagentsMCPServer(MCPServerBase):
    """Implementation of MCP tools manager for smolagents."""
    def __init__(self, mcp_tool: MCPParams):
        super().__init__(mcp_tool, "any-agent[mcp,smolagents]", mcp_available)
        self.tool_collection: ToolCollection | None = None # Smolagents specific object

    async def setup_stdio_tools(self) -> None:
        if not isinstance(self.mcp_tool, MCPStdioParams):
            raise ValueError(...) # Ensure correct type

        # --- This uses the underlying smolagents/mcp library ---
        server_parameters = StdioServerParameters(
            command=self.mcp_tool.command,
            args=list(self.mcp_tool.args),
            env={**os.environ},
        )
        # ToolCollection.from_mcp sets up the connection and discovers tools
        self.tool_collection = self.exit_stack.enter_context(
            ToolCollection.from_mcp(server_parameters, trust_remote_code=True)
        )
        # ------------------------------------------------------------

    async def setup_sse_tools(self) -> None:
        if not isinstance(self.mcp_tool, MCPSseParams):
            raise ValueError(...) # Ensure correct type

        # --- This uses the underlying smolagents/mcp library for SSE ---
        server_parameters = {"url": self.mcp_tool.url}
        self.tool_collection = self.exit_stack.enter_context(
            ToolCollection.from_mcp(server_parameters, trust_remote_code=True)
        )
        # ---------------------------------------------------------------------

    async def setup_tools(self) -> None:
        await super().setup_tools() # Calls setup_stdio_tools or setup_sse_tools

        if not self.tool_collection:
            raise ValueError(...) # Ensure connection was set up

        # Get the discovered tools from the smolagents ToolCollection object
        tools = self.tool_collection.tools

        # --- Apply filtering if mcp_tool.tools was specified ---
        requested_tools = self.mcp_tool.tools
        if not requested_tools:
            self.tools = tools # Use all discovered tools
        else:
            # Filter tools based on the requested list
            filtered_tools = [tool for tool in tools if tool.name in requested_tools]
            # ... check if all requested tools were found ...
            self.tools = filtered_tools
        # ------------------------------------------------------------
```

**Explanation:**

- `MCPServerBase` defines the common interface for setting up MCP connections (`setup_stdio_tools`, `setup_sse_tools`).
- Framework-specific classes like `SmolagentsMCPServer` inherit from `MCPServerBase` and implement these methods using the specific MCP client capabilities provided by their underlying framework or related libraries (`smolagents`, `mcp`).
- The `setup_stdio_tools` and `setup_sse_tools` methods in the framework handler are responsible for using the `MCPParams` configuration to initiate the connection to the external server.
- Once the connection is made, they use the MCP protocol (via the underlying libraries) to discover the tools offered by the external server.

- The discovered tools are stored in `self.tools` and are then passed back to the main framework agent code to be included in the agent's capabilities.
- The filtering based on `mcp_tool.tools` happens after discovery, allowing you to expose only a subset of the external server's tools to a particular agent.

This architecture allows `any-agent` to integrate tools from completely separate processes or services, providing a powerful way to extend your agent's capabilities beyond what's available within its own process.

## Conclusion

In this chapter, you learned about MCP Tools, a powerful way to connect your agent to external tool services running in separate processes. You saw how `MCPStdioParams` and `MCPSseParams` are used in your `AgentConfig` to configure these connections, and how `any-agent` and the underlying framework handlers manage the process of connecting to the external MCP server, discovering its tools, and making them available to your agent.

MCP Tools are essential for building agents that interact with complex, shared, or long-running external systems.

Now that you understand how to configure agents, use the unified interface, select frameworks, and provide tools (both simple functions and external MCP services), you're ready to learn how to observe and understand what your agent is doing. In the next chapter, we'll dive into Tracing.

Chapter 6: Tracing

Welcome back! In our previous chapters, we've learned how to configure our agent (Chapter 1: Agent Configuration), use the `AnyAgent` class as a unified interface (Chapter 2: AnyAgent (Unified Interface)), understand the different agent frameworks (Chapter 3: Agent Frameworks), and equip our agent with powerful tools (Chapter 4: Tools) including connecting to external services via MCP (Chapter 5: MCP Tools).

Now that you know how to build and run an agent that can think and act, a crucial question arises: **How do you know exactly what the agent did to arrive at its answer?**

When an agent runs, especially with tools, it might go through several steps: thinking, deciding which tool to use, calling the tool, getting a result, thinking again based on the result, maybe calling another tool, and finally formulating a response. If the final answer isn't what you expected, or if the agent gets stuck, it's hard to debug without seeing this step-by-step process.

## The Problem: The Agent is a Black Box

Imagine you ask your agent to find the weather forecast and it gives you a completely wrong answer. Did it: * Misunderstand the question? * Fail to use the search tool? * Use the search tool but get bad results? * Get good results but misinterpret them?

Without visibility into its internal steps, it's like trying to fix a car engine by just looking at the outside.

## The Solution: Tracing

**Tracing** is the system that provides this crucial visibility. It's like installing a "flight recorder" on your agent. Every significant event during the agent's execution – every time it interacts with the language model (LLM), every time it calls a tool, every major step in its internal process – is logged and recorded.

This log, or **trace**, gives you a detailed timeline of the agent's journey from receiving the prompt to producing the final response. You can see: * When the LLM was called and what prompt it received. * Which tool was called, with what arguments. * What result the tool returned. * How the agent processed the tool's result. * The sequence of decisions the agent made.

This information is invaluable for: * **Debugging:** Pinpointing exactly where an agent went wrong. * **Understanding:** Learning how the agent uses its tools and instructions. * **Optimizing:** Identifying inefficient steps or unnecessary tool calls. * **Evaluating:** Comparing different agent configurations or frameworks based on their execution paths (more on this in the next chapter!).

`any-agent` provides a built-in tracing system that works across supported frameworks, giving you a consistent way to observe agent execution.

## Configuring Tracing with `TracingConfig`

We briefly introduced `TracingConfig` in Chapter 1: Agent Configuration. This is the object you use to tell `any-agent` *how* you want the tracing to behave.

The key settings in `TracingConfig` are:

- **`output_dir`**: This is the most important setting. If you set this to a directory path (e.g., `"traces"`), `any-agent` will save a detailed trace of the agent's execution to a JSON file inside that directory. If you set it to `None`, no file will be saved.
- **`llm`**, **`tool`**, **`agent`**, **`chain`**: These settings allow you to specify colors for different types of events when they are printed to your console during execution. This makes the console output much easier to read and follow.
  - `llm`: Events related to interactions with the Language Model.
  - `tool`: Events related to calling tools.
  - `agent`: High-level agent steps.

- chain: Steps within a sequence or chain of operations.

## Enabling and Using Tracing

To enable tracing, you simply create a `TracingConfig` object and pass it to the `tracing` parameter when you create your agent using `AnyAgent.create()`.

Let's take our example from Chapter 4: Tools where the agent uses a web search tool, and add tracing to it.

```
import os
from any_agent import AgentConfig, AnyAgent, TracingConfig
from any_agent.tools import search_web # Import the pre-built tool

# --- 1. Configure Tracing ---
# We want to save the trace to a file AND see it in the console
tracing_config = TracingConfig(
    output_dir="my_agent_traces", # Save trace files here
    llm="yellow",               # LLM calls in yellow
    tool="blue",                # Tool calls in blue
    agent="green"               # Agent steps in green
)
print(f"Tracing configured to save to '{tracing_config.output_dir}' and use colors.")
# --------------------------

# 2. Configure the agent (same as before)
agent_config = AgentConfig(
    model_id="gpt-4o", # Or another model that supports tools
    instructions="You are a helpful assistant that can search the web.",
    tools=[search_web], # Add the search_web function
)

# 3. Create the agent, passing BOTH configs
# Make sure you have the framework installed: pip install 'any-agent[smolagents]'
try:
    print("\nCreating agent with tracing enabled...")
    agent = AnyAgent.create(
        "smolagents",      # Use the Smolagents framework (or another)
        agent_config,
        tracing=tracing_config # Pass the tracing config here!
    )
    print(f"Agent created with framework: {agent.framework.value}")

    # 4. Run the agent with a query that requires the tool
    print("\nRunning agent with web search query...")
    response = agent.run("What is the current weather in London?")
    print("\nAgent's Final Response:")
    print(response)

    # After execution, check the 'my_agent_traces' directory for a JSON file

except ImportError as e:
    print(f"\nError: Could not create agent. Make sure you have the framework installed.")
    print(f"Details: {e}")
except Exception as e:
    print(f"\nAn unexpected error occurred: {e}")
```

**Explanation:**

- We import `TracingConfig`.
- We create an instance of `TracingConfig`, specifying an `output_dir` and some colors.
- When calling `AnyAgent.create()`, we pass this `tracing_config` object to the `tracing` parameter.
- `any-agent` automatically detects that tracing is configured and sets up the tracing system *before* the agent starts running.
- When `agent.run()` is called, the underlying framework's actions (LLM calls, tool calls) are automatically captured by the tracing system.
- As these events happen, you will see colored output in your console (if colors are configured).
- After the agent finishes, a JSON file containing the full trace data will be saved in the directory specified by `output_dir` (e.g., `my_agent_traces/smolagents-YYYY-MM-DD-HH-MM-SS.json`).

The console output might look something like this (colors added conceptually):

```
Tracing configured to save to 'my_agent_traces' and use colors.

Creating agent with tracing enabled...
Agent created with framework: smolagents

Running agent with web search query...
--- agent ---
Agent Start
--- agent ---
--- llm ---
LLM Start
Input: Human: What is the current weather in London?
Tool Descriptions:
- Name: search_web
  Description: Searches the web for a query.
  Parameters:
    query (str): The search query.
--- llm ---
--- llm ---
LLM End
Output: Thinking Process: The user is asking for current information about the weather in Lo
Tool Call: search_web({"query": "current weather in London"})
--- llm ---
--- tool ---
Tool Call: search_web({"query": "current weather in London"})
--- tool ---
--- tool ---
Tool End
Output: [Search results about London weather...]
--- tool ---
--- llm ---
LLM Start
Input: Human: What is the current weather in London?
Tool Descriptions:
- Name: search_web
  Description: Searches the web for a query.
  Parameters:
    query (str): The search query.
Tool Results:
search_web: [Search results about London weather...]
--- llm ---
--- llm ---
LLM End
Output: Based on the search results, the current weather in London is [Summary of weather fr
--- llm ---
--- agent ---
Agent End
--- agent ---

Agent's Final Response:
Based on the search results, the current weather in London is [Summary of weather from resul
```

This console output gives you a live view of the agent's steps. The JSON file provides the same information in a structured format that can be parsed and analyzed programmatically.

## Under the Hood: How Tracing Works

`any-agent`'s tracing system is built on top of the **OpenTelemetry** standard. OpenTelemetry is a set of tools and APIs used to collect telemetry data (like traces, metrics, and logs) from your applications.

Here's a simplified look at how it works in `any-agent`:

1. **Instrumentation:** `any-agent` uses libraries from the `openinference.instrumentation`

project. These libraries provide "instrumentors" for specific agent frameworks (like LangChain, Smolagents, etc.). When you call `setup_tracing`, `any-agent` finds the correct instrumentor for your chosen framework and tells it to start monitoring the framework's activity.
2. **Capturing Spans:** As the framework runs (e.g., makes an LLM call, executes a tool), the instrumentor intercepts these events and creates **spans**. A span represents a single operation or step in the trace (like "LLM Call" or "Tool Execution"). Spans contain information like the operation name, start/end times, input/output data, and any errors.
3. **Tracer Provider:** The spans are sent to an OpenTelemetry **Tracer Provider**. This provider manages the tracing process and sends the spans to processors.
4. **Span Processors: Span Processors** decide what to do with the spans. `any-agent` uses two main processors:
    - `SimpleSpanProcessor`: Sends spans immediately to an exporter (used for the JSON file).
    - `BatchSpanProcessor`: Collects spans in batches before sending them to an exporter (used for the console output, which is less time-sensitive).
5. **Exporters: Exporters** are responsible for sending the spans to their final destination. `any-agent` uses custom exporters:
    - `JsonFileSpanExporter`: Writes the span data to a JSON file.
    - `RichConsoleSpanExporter`: Formats the span data and prints it to the console using `rich` for colors and formatting. This exporter uses a `TelemetryProcessor` (which we'll cover in the next chapter) to understand the specific structure of spans from different frameworks.

When you call `AnyAgent.create(..., tracing=tracing_config)`, the `setup_tracing` function is called internally.

```
# Simplified flow within AnyAgent.create()
# ... inside AnyAgent.create() method ...
if tracing is not None:
    # Call the setup_tracing function
    trace_file_name = setup_tracing(
        agent_framework, # The framework you chose
        tracing # Your TracingConfig object
    )
    # Store the trace_file_name if needed
    # ...
# ... continue creating and loading the framework agent ...
```

The `setup_tracing` function itself looks something like this (simplified from `src/any_agent/tracing.py`):

```
# From src/any_agent/tracing.py (simplified)
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import SimpleSpanProcessor, BatchSpanProcessor
```

```python
from any_agent.config import AgentFramework, TracingConfig
# ... other imports for exporters and instrumentors ...

def setup_tracing(
    agent_framework: AgentFramework,
    tracing_config: TracingConfig,
) -> str:
    """Setup tracing for agent_framework using openinference.instrumentation."""

    tracer_provider = TracerProvider() # Create the main provider

    file_name = None
    if tracing_config.output_dir is not None:
        # Setup JSON file exporter if output_dir is specified
        # ... create directory if needed ...
        file_name = f"{tracing_config.output_dir}/{agent_framework.value}-timestamp.json"
        json_file_exporter = JsonFileSpanExporter(file_name=file_name)
        span_processor = SimpleSpanProcessor(json_file_exporter)
        tracer_provider.add_span_processor(span_processor) # Add processor for file export

    # Setup console exporter (always active if tracing_config is provided)
    console_exporter = RichConsoleSpanExporter(agent_framework, tracing_config)
    processor = BatchSpanProcessor(console_exporter)
    tracer_provider.add_span_processor(processor) # Add processor for console output

    # Set this provider as the global one for OpenTelemetry
    trace.set_tracer_provider(tracer_provider)

    # Get the correct instrumentor for the chosen framework
    instrumenter = get_instrumenter_by_framework(agent_framework)
    # Tell the instrumentor to start monitoring, using our provider
    instrumenter.instrument(tracer_provider=tracer_provider)

    return file_name # Return the name of the trace file

# Helper function to get the right instrumentor based on framework
def get_instrumenter_by_framework(framework: AgentFramework):
    # ... logic to return OpenAIAgentsInstrumentor(), SmolagentsInstrumentor(), etc. ...
    # This maps the AgentFramework enum to the correct openinference class
    pass # Simplified
```

**Explanation:**

- `setup_tracing` creates a `TracerProvider`.
- It conditionally adds a `SimpleSpanProcessor` with a `JsonFileSpanExporter` if `output_dir` is set in `TracingConfig`.
- It always adds a `BatchSpanProcessor` with a `RichConsoleSpanExporter`

for console output.
- It sets this `tracer_provider` as the active one for OpenTelemetry.
- It calls `get_instrumenter_by_framework` to find the specific `openinference` class that knows how to monitor the chosen agent framework.
- It calls `instrumenter.instrument()`, passing the `tracer_provider`. This activates the monitoring for that framework.

Once this setup is complete, any subsequent LLM calls, tool calls, or other instrumented events within the framework will automatically generate spans, which will be processed and exported to your console and/or file.

Here's a simplified sequence diagram of the tracing flow during agent execution:

```
sequenceDiagram
    participant YourCode
    participant AnyAgentInstance as AnyAgent Object
    participant FrameworkAgent as Specific Framework Agent
    participant FrameworkLib as Underlying Framework Library
    participant Instrumentor as openinference Instrumentor
    participant TracerProvider as OpenTelemetry TracerProvider
    participant SpanProcessors as Span Processors
    participant Exporters as Exporters (Console/File)

    YourCode->>AnyAgentInstance: Call run(prompt)
    AnyAgentInstance->>FrameworkAgent: Call framework-specific run
    FrameworkAgent->>FrameworkLib: Call LLM/Tool method
    FrameworkLib->>Instrumentor: Event happens (e.g., LLM call)
    Instrumentor->>TracerProvider: Create & Send Span
    TracerProvider->>SpanProcessors: Forward Span
    SpanProcessors->>Exporters: Send Span Data
    Exporters->>Console: Print formatted output
    Exporters->>TraceFile: Write JSON data
    Exporters-->>SpanProcessors: Export Result
    SpanProcessors-->>TracerProvider: Processing Result
    TracerProvider-->>Instrumentor: Span Processed
    Instrumentor-->>FrameworkLib: Event Handled
    FrameworkLib-->>FrameworkAgent: Return Result
    FrameworkAgent-->>AnyAgentInstance: Return Result
    AnyAgentInstance-->>YourCode: Return Final Response
```

Tracing provides an essential window into the agent's decision-making and execution process, turning the black box into a transparent system you can understand and debug.

## Conclusion

In this chapter, you learned that Tracing is the vital system for observing your agent's step-by-step execution, acting like a flight recorder for debugging and analysis. You saw how to enable tracing by creating a `TracingConfig` object and passing it to `AnyAgent.create()`, controlling where traces are saved (`output_dir`) and how they appear in the console (color settings). You also got a peek under the hood, understanding that `any-agent` leverages OpenTelemetry and `openinference.instrumentation` to automatically capture events from the underlying frameworks and export them to both console output and structured JSON files.

With tracing enabled, you now have the power to see exactly what your agent is doing. This detailed execution data is incredibly useful, especially when it comes to evaluating how well your agent performs. In the next chapter, we'll explore Evaluation and how tracing plays a key role in it.

Chapter 7: Evaluation

Welcome back! In our previous chapters, we've learned how to configure our agent (Chapter 1: Agent Configuration), use the `AnyAgent` class as a unified interface (Chapter 2: AnyAgent (Unified Interface)), understand the different agent frameworks (Chapter 3: Agent Frameworks), and equip our agent with powerful tools (Chapter 4: Tools) including connecting to external services via MCP (Chapter 5: MCP Tools). Most recently, in Chapter 6: Tracing, we learned how to enable tracing to get a detailed log of our agent's execution steps.

Now that you can build agents and see exactly what they did, a critical question remains: **How do you know if your agent actually did a *good* job?**

## The Problem: Is the Agent's Work Good Enough?

Imagine you've built an agent to answer questions about a specific document using a tool that can read files. You give it a question, it uses the tool, and it gives you an answer. But is the answer correct? Did it follow all the instructions? Did it use the tool efficiently?

If you're just building one agent for yourself, you can manually check its output. But what if you're: * Trying out different agent frameworks to see which performs best? * Experimenting with different prompts or instructions? * Making changes to your tools? * Building many agents for different tasks?

Manually checking every agent's output every time is slow, inconsistent, and doesn't give you a clear, objective score. You need a systematic way to measure performance.

## The Solution: Evaluation

**Evaluation** is the process of automatically measuring how well an agent performs on a specific task. It's like having a quality control inspector who checks the agent's work against a predefined checklist of requirements and standards.

In `any-agent`, evaluation uses two main things:

1. **The Agent's Execution Trace:** The detailed log of steps the agent took, which we learned about in Chapter 6: Tracing. This shows *how* the agent arrived at its answer.
2. **Predefined Criteria:** A set of rules or expected outcomes that define what a "good" performance looks like for a given task.

By comparing the trace and the final answer against these criteria, `any-agent` can give your agent a score and tell you *why* it passed or failed certain checks.

## Defining What "Good" Means: Test Cases

To evaluate an agent, you first need to define the task and the criteria for success. In `any-agent`, you do this using a `TestCase`.

A `TestCase` is an object that holds everything needed to evaluate one specific run of an agent:

- **`prompt`**: The input you give to the agent.
- **`ground_truth`**: The correct answer or expected output (optional, but useful for direct comparison).
- **`checkpoints`**: Criteria for evaluating the *steps* the agent took (checking the trace). Did it use a specific tool? Did it reach a certain intermediate state?
- **`final_answer_criteria`**: Criteria for evaluating the agent's *final answer*. Does it contain specific information? Is it in the correct format?
- **`llm_judge`**: The language model to use as an "LLM-as-a-judge" to evaluate the criteria (more on this below).

Let's define a simple `TestCase` for an agent that should use a web search tool to find the capital of France.

```
from any_agent.evaluation.test_case import TestCase, CheckpointCriteria, GroundTruthAnswer

# Define the criteria for a successful run
test_case = TestCase(
    prompt="What is the capital of France?", # The input for the agent
    ground_truth=[
        GroundTruthAnswer(value="Paris", type="text") # The expected answer
    ],
    checkpoints=[
        CheckpointCriteria(
            criteria="The agent must use the 'search_web' tool.", # Check if the tool was c
            points=1 # How many points this check is worth
        )
    ],
    final_answer_criteria=[
        CheckpointCriteria(
            criteria="The final answer must explicitly state 'Paris'.", # Check the final o
            points=2 # This is more important, worth more points
        )
    ],
    llm_judge="gpt-4o", # The model to use for LLM-as-a-judge evaluation
    # output_path="evaluation_results.jsonl" # Where to save the results (optional)
)

print("Test Case defined:")
print(f"  Prompt: {test_case.prompt}")
print(f"  Expected Answer: {test_case.ground_truth[0].value}")
print(f"  Checkpoint Criteria: {len(test_case.checkpoints)}")
print(f"  Final Answer Criteria: {len(test_case.final_answer_criteria)}")
print(f"  LLM Judge Model: {test_case.llm_judge}")
```

**Explanation:**

- We import the necessary classes from `any_agent.evaluation.test_case`.
- We create a `TestCase` object.
- We provide the `prompt`.
- We specify the `ground_truth` as a list containing the expected value ("Paris").
- We add a `CheckpointCriteria` to the `checkpoints` list, requiring the agent to use the `search_web` tool.
- We add a `CheckpointCriteria` to the `final_answer_criteria` list, requiring the final answer to mention "Paris".
- We specify `gpt-4o` as the `llm_judge` model. This model will be used to read the trace and the final answer and determine if the criteria were met.

This `TestCase` now clearly defines what a successful run looks like for this specific task.

## Running the Evaluation

To evaluate an agent run, you need two things: the `TestCase` you just defined, and the trace file generated by the agent's execution (as covered in Chapter 6: Tracing).

The main function for running evaluation is `evaluate_telemetry`.

Here's the typical flow:

1. Create your `AgentConfig` and `TracingConfig` (with `output_dir` set).
2. Create your agent using `AnyAgent.create()`.

3. Run the agent with the prompt from your `TestCase`. This saves the trace file.
4. Find the path to the saved trace file.
5. Create your `TestCase`.
6. Call `evaluate_telemetry`, passing the `TestCase` and the trace file path.

Let's put it together (assuming you have a `search_web` tool available and the necessary framework installed, e.g., `pip install 'any-agent[smolagents]'`):

```
import os
import glob # To find the latest trace file

from any_agent import AgentConfig, AnyAgent, TracingConfig
from any_agent.tools import search_web
from any_agent.evaluation.test_case import TestCase, CheckpointCriteria, GroundTruthAnswer
from any_agent.evaluation.evaluate import evaluate_telemetry # Import the evaluation functi

# --- 1. Setup Agent with Tracing ---
trace_output_dir = "eval_traces"
tracing_config = TracingConfig(output_dir=trace_output_dir)

agent_config = AgentConfig(
    model_id="gpt-4o",
    instructions="You are a helpful assistant that can search the web.",
    tools=[search_web],
)

# Ensure the trace directory exists
os.makedirs(trace_output_dir, exist_ok=True)

try:
    print("Creating agent with tracing...")
    agent = AnyAgent.create("smolagents", agent_config, tracing=tracing_config)
    print(f"Agent created with framework: {agent.framework.value}")

    # --- 2. Define the Test Case ---
    test_case = TestCase(
        prompt="What is the capital of France?",
        ground_truth=[GroundTruthAnswer(value="Paris", type="text")],
        checkpoints=[CheckpointCriteria(criteria="The agent must use the 'search_web' tool.'
        final_answer_criteria=[CheckpointCriteria(criteria="The final answer must explicitly
        llm_judge="gpt-4o",
        output_path="evaluation_results.jsonl" # Save evaluation results here
    )
    print("\nTest Case defined.")

    # --- 3. Run the Agent to generate a trace ---
    print("\nRunning agent to generate trace...")
    response = agent.run(test_case.prompt)
    print("\nAgent's Final Response:")
    print(response)

    # --- 4. Find the generated trace file ---
    # Trace files are named like framework-timestamp.json
    # Find the latest one in the output directory
    trace_files = glob.glob(os.path.join(trace_output_dir, "*.json"))
    trace_files.sort(key=os.path.getmtime, reverse=True) # Sort by modification time
    if not trace_files:
        raise FileNotFoundError(f"No trace files found in {trace_output_dir}")
    latest_trace_file = trace_files[0]
    print(f"\nFound latest trace file: {latest_trace_file}")

    # --- 5. Run the Evaluation ---
    print("\nRunning evaluation...")
    evaluate_telemetry(test_case=test_case, telemetry_path=latest_trace_file)
    print("\nEvaluation complete. Check console output and evaluation_results.jsonl")

except ImportError as e:
    print(f"\nError: Could not run. Make sure you have the framework and litellm installed.'
    print(f"Details: {e}")
    print("Try: pip install 'any-agent[smolagents,evaluation]'") # Need evaluation extra fo
except Exception as e:
    print(f"\nAn unexpected error occurred: {e}")
```

**Explanation:**

- We set up the agent and tracing, ensuring `output_dir` is set.
- We define the `TestCase` as described before, including an `output_path` to save the evaluation results.
- We run the agent with the prompt from the `TestCase`. This creates a trace file in `eval_traces`.
- We use `glob` to find the path to the most recently created trace file.
- We call `evaluate_telemetry`, passing our `test_case` object and the path to the trace file (`telemetry_path`).

When `evaluate_telemetry` runs, it will: 1. Load the trace data from the JSON file. 2. Use a `TelemetryProcessor` (which we'll cover in the next chapter) to understand the trace structure and extract key information like the final answer and evidence from tool calls. 3. Use the `llm_judge` model specified in the `TestCase` to evaluate the `checkpoints` against the trace evidence and the `final_answer_criteria` against the extracted final answer and ground truth. 4. Print a summary of the results to the console, showing which criteria passed or failed and why (based on the LLM judge's reasoning). 5. Calculate a final score based on the points assigned to each criterion. 6. Save the detailed results (including the score, messages, and paths to the trace/test case) to the file specified by `test_case.output_path` (`evaluation_results.jsonl`).

The console output from the evaluation step might look something like this:

```
... (agent run output) ...

Running evaluation...
Telemetry loaded from eval_traces/smolagents-YYYY-MM-DD-HH-MM-SS.json
Hypothesis Final answer extracted: Based on the search results, the capital of France is Par

Passed:
- The agent must use the 'search_web' tool.
- The LLM judge confirmed the trace shows a call to search_web.

Passed:
- The final answer must explicitly state 'Paris'.
- The LLM judge confirmed the final answer contains 'Paris'.

All checkpoints passed!
Passed checkpoints: 2
Failed checkpoints: 0
=====================================
Score: 3/3
=====================================
Evaluation complete. Check console output and evaluation_results.jsonl
```

This output clearly shows that the agent passed both criteria, giving it a perfect score. If it had failed, the output would show the failed criteria and the reason provided by the LLM judge.

## Under the Hood: How Evaluation Works

The evaluation process in `any-agent` involves several components working together:

1. **Loading Telemetry:** The `evaluate_telemetry` function starts by loading the raw trace data from the specified JSON file.
2. **Telemetry Processing:** It uses a `TelemetryProcessor` (specific to the agent framework used, determined from the trace data) to parse the raw trace. The processor extracts the agent's final answer and gathers "evidence" from the trace, such as tool inputs/outputs and key LLM interactions. This is a crucial step because different frameworks structure their traces differently, and the processor normalizes this. (We'll explore Telemetry Processing in detail in the next chapter).
3. **Evaluators:** `any-agent` uses different types of evaluators:
    - `CheckpointEvaluator`: Takes the extracted trace "evidence" and the `checkpoints` criteria from the `TestCase`. It uses the `llm_judge`

model to determine if each checkpoint criterion was met based on the evidence.
- `HypothesisEvaluator`: Takes the extracted final answer, the `ground_truth`, and the `final_answer_criteria` from the `TestCase`. It uses the `llm_judge` model to determine if each final answer criterion was met by comparing the agent's answer to the ground truth.
- `QuestionAnsweringSquadEvaluator`: This is a simpler evaluator that performs a direct string comparison between the agent's final answer and the `ground_truth` using metrics like Exact Match and F1 score (common in Question Answering tasks). It doesn't use an LLM judge.

4. **LLM-as-a-Judge:** For `CheckpointEvaluator` and `HypothesisEvaluator`, the core logic is handled by the `LLMEvaluator` base class. This class constructs a prompt for the `llm_judge` model, including the criterion to check, the relevant evidence (trace snippets) or the final answer/ground truth. It asks the LLM to output a structured JSON response indicating `passed` (true/false) and a `reason`. `LLMEvaluator` parses this JSON response.
5. **Aggregating Results:** `evaluate_telemetry` collects the results from all the evaluators, calculates the total score based on the points, formats the output message, and saves everything using the `save_evaluation_results` helper function.

Here's a simplified sequence diagram of the evaluation process:

```
sequenceDiagram
    participant User
    participant YourCode
    participant EvaluateTelemetry as evaluate_telemetry()
    participant TelemetryProcessor as Telemetry Processor
    participant CheckpointEval as CheckpointEvaluator
    participant HypothesisEval as HypothesisEvaluator
    participant DirectEval as Direct Evaluator
    participant LLMJudge as LLM Judge Model
    participant ResultsSaver as Results Saver

    User->>YourCode: Call evaluate_telemetry(test_case, trace_path)
    YourCode->>EvaluateTelemetry: Pass test_case, trace_path
    EvaluateTelemetry->>EvaluateTelemetry: Load trace from trace_path
    EvaluateTelemetry->>TelemetryProcessor: Create processor for framework
    TelemetryProcessor->>EvaluateTelemetry: Extract final answer, evidence
    EvaluateTelemetry->>CheckpointEval: Evaluate checkpoints (evidence, criteria, LLM)
    CheckpointEval->>LLMJudge: Ask LLM: "Does evidence meet criterion X?"
    LLMJudge-->>CheckpointEval: Return JSON: {"passed":..., "reason":...}
    CheckpointEval-->>EvaluateTelemetry: Return Checkpoint Results
    EvaluateTelemetry->>HypothesisEval: Evaluate final answer (answer, ground truth, criter
    HypothesisEval->>LLMJudge: Ask LLM: "Does answer meet criterion Y?"
    LLMJudge-->>HypothesisEval: Return JSON: {"passed":..., "reason":...}
    HypothesisEval-->>EvaluateTelemetry: Return Hypothesis Results
    EvaluateTelemetry->>DirectEval: Evaluate direct match (answer, ground truth)
    DirectEval-->>EvaluateTelemetry: Return Direct Match Result
    EvaluateTelemetry->>EvaluateTelemetry: Aggregate results, calculate score
    EvaluateTelemetry->>ResultsSaver: Save results (output_path)
    ResultsSaver-->>EvaluateTelemetry: Results saved
    EvaluateTelemetry-->>YourCode: Evaluation complete
    YourCode-->>User: Display summary
```

Let's look at simplified snippets from the evaluation code files:

```python
# From src/any_agent/evaluation/evaluate.py (simplified)
import json
from any_agent.evaluation.test_case import TestCase
from any_agent.evaluation.evaluators import CheckpointEvaluator, HypothesisEvaluator, Questi
from any_agent.telemetry import TelemetryProcessor
from any_agent.evaluation.results_saver import save_evaluation_results
from any_agent.logging import logger # Use the shared logger

def evaluate_telemetry(test_case: TestCase, telemetry_path: str) -> None:
    # 1. Load Telemetry
    with open(telemetry_path) as f:
        telemetry = json.loads(f.read())

    # 2. Telemetry Processing
    agent_framework = TelemetryProcessor.determine_agent_framework(telemetry)
    processor = TelemetryProcessor.create(agent_framework)
    hypothesis_answer = processor.extract_hypothesis_answer(trace=telemetry)
    evidence = processor.extract_evidence(telemetry) # Used by CheckpointEvaluator

    # 3. Evaluators
    checkpoint_evaluator = CheckpointEvaluator(model=test_case.llm_judge)
    checkpoint_results = checkpoint_evaluator.evaluate(
        telemetry=telemetry, # Pass full telemetry (though evaluator uses extracted evidenc
        checkpoints=test_case.checkpoints,
        processor=processor, # Pass processor to evaluator
    )

    hypothesis_evaluator = HypothesisEvaluator(model=test_case.llm_judge)
    hypothesis_answer_results = hypothesis_evaluator.evaluate(
        hypothesis_final_answer=hypothesis_answer,
        ground_truth_answer_dict=test_case.ground_truth,
        ground_truth_checkpoints=test_case.final_answer_criteria,
    )
    direct_results = []
    if test_case.ground_truth: # Only run direct comparison if ground truth exists
        direct_evaluator = QuestionAnsweringSquadEvaluator()
        direct_results = direct_evaluator.evaluate(
            hypothesis_answer=hypothesis_answer,
            ground_truth_answer=test_case.ground_truth,
        )

    # 4. Aggregate Results
    verification_results = checkpoint_results + hypothesis_answer_results + direct_results
    # ... calculate score and format output_message ...

    # 5. Save Results
    save_evaluation_results(
        test_case=test_case,
        output_path=test_case.output_path,
        output_message=output_message,
        telemetry_path=telemetry_path,
        hypothesis_answer=hypothesis_answer,
        passed_checks=len(passed_checks),
        failed_checks=len(failed_checks),
        score=score,
    )
    logger.info(output_message) # Print summary to console
# From src/any_agent/evaluation/evaluators/LLMEvaluator.py (simplified)
import json
import re
from abc import ABC
from litellm import completion # Uses litellm for LLM calls

class LLMEvaluator(ABC):
    def __init__(self, model: str):
        self.model = model # The LLM judge model ID

    def llm_evaluate_with_criterion(
        self,
        criteria: str,
        points: int,
        # ... other arguments like evidence, hypothesis_final_answer, ground_truth_output .
    ) -> EvaluationResult:
        # --- Construct the prompt for the LLM judge ---
        prompt = f"""
        Evaluate if the following criterion was met...
        Criterion: {criteria}
        ... include evidence/answers ...
```

```
Output valid JSON: {{"passed": true/false, "reason": "..."}}
"""
# ---------------------------------------------

# --- Call the LLM judge model ---
response = completion(
    model=self.model,
    messages=[{"role": "user", "content": prompt}],
)
content = response.choices[0].message.content
# ------------------------------

# --- Parse the LLM's JSON response ---
try:
    json_match = re.search(r"```(?:json)?\s*(\{.*?\})\s*```|(\{.*?\})", content, re.
    if json_match:
         json_str = next(group for group in json_match.groups() if group)
         evaluation_data = json.loads(json_str)
    else:
         evaluation_data = json.loads(content) # Fallback
    evaluation_data["criteria"] = criteria
    evaluation_data["points"] = points
    return EvaluationResult.model_validate(evaluation_data)
except Exception as e:
    # Handle parsing errors
    return EvaluationResult(
        passed=False,
        reason=f"Parsing error: {e!s} Response: {content}",
        criteria=criteria,
        points=points
    )
# -------------------------------------
```

**Explanation:**

- `evaluate_telemetry` orchestrates the process, loading data, creating the `TelemetryProcessor`, and calling the specific evaluators.
- `LLMEvaluator` is the base class for evaluators that use an LLM. Its `llm_evaluate_with_criterion` method builds the prompt for the LLM judge, calls the LLM (using `litellm`), and parses the JSON response.
- `CheckpointEvaluator` and `HypothesisEvaluator` inherit from `LLMEvaluator` and implement their `evaluate` methods, which loop through the criteria and call `llm_evaluate_with_criterion` for each one, passing the relevant data (evidence or answers).
- `QuestionAnsweringSquadEvaluator` is a separate evaluator for direct comparison, not using the LLM judge.

This structured approach allows any-agent to provide flexible and detailed evaluation, using the power of LLMs to judge complex criteria based on the agent's full execution trace.

## Conclusion

In this chapter, you learned that Evaluation is the process of measuring agent performance against predefined criteria, using the agent's execution trace. You saw how to define these criteria using a `TestCase` object, specifying the prompt, expected outcomes, and the LLM judge model. You also learned how to run the evaluation using the `evaluate_telemetry` function, which takes a `TestCase` and a trace file, and provides a detailed report and score. Finally, you got a peek under the hood at how `any-agent` uses `TelemetryProcessor` and different evaluators (including LLM-as-a-judge) to perform this analysis.

Evaluation is powerful because it turns subjective assessment into objective measurement, allowing you to compare agents and improve their performance systematically. A key part of this process is understanding and processing the trace data itself. In the next chapter, we'll dive deeper into Telemetry Processing.

Chapter 8: Telemetry Processing

Welcome back! In our journey through `any-agent`, we've learned how to configure our agent (Chapter 1: Agent Configuration), use the `AnyAgent` class as a unified interface (Chapter 2: AnyAgent (Unified Interface)), understand the different agent frameworks (Chapter 3: Agent Frameworks), and equip our agent with powerful tools (Chapter 4: Tools) including connecting to external services via MCP (Chapter 5: MCP Tools). In Chapter 6: Tracing, we saw how to get a detailed log (a trace) of our agent's execution steps, and in Chapter 7: Evaluation, we learned how this trace is essential for automatically measuring our agent's performance.

But there's a hidden step between getting the raw trace data and using it for evaluation or visualization: **understanding and processing that data**.

## The Problem: Raw Traces Speak Different Languages

Remember from Chapter 6: Tracing that `any-agent` uses the OpenTelemetry standard and `openinference.instrumentation` to capture trace data. This system generates detailed logs of events (spans) like LLM calls, tool executions, etc.

However, the *exact structure* and *naming conventions* within these spans can vary slightly depending on the specific agent framework being used. For example: * One framework might store the LLM's input prompt in an attribute called `"llm.input.messages"`. * Another might use `"input.value"` and require you to parse a JSON string within it. * The way a tool's output is recorded might also differ. * Identifying the "final answer" span might require looking for different patterns in the trace depending on the framework's internal workflow.

If you wanted to write code to, say, extract the final answer from a trace, you'd have to write different parsing logic for LangChain traces, Smolagents traces, LlamaIndex traces, etc. This is complicated and defeats the purpose of having a unified interface like `AnyAgent`!

This is like having trace data written in different dialects. The evaluation system and tracing visualization tools need to understand a single, consistent "language" of trace information, not learn every framework's dialect.

## The Solution: Telemetry Processing

This is where **Telemetry Processing** comes in. The `TelemetryProcessor` component acts as a **translator** or **standardizer** for the raw trace data.

Its job is to: 1. Look at the raw trace data generated by a specific framework. 2. Identify which framework generated the trace. 3. Use framework-specific logic to parse the trace. 4. Extract key pieces of information (like the final answer, tool inputs/outputs, LLM prompts/responses) in a consistent, standardized format, regardless of the original trace structure.

Think of the `TelemetryProcessor` as the part of `any-agent` that reads the raw flight recorder data (the trace) and translates the framework-specific jargon into a clear report that the evaluation system and visualization tools can easily understand.

You, as the user, typically **do not interact directly** with the `TelemetryProcessor`. It works behind the scenes when you use features like:

- **Evaluation:** When you call `evaluate_telemetry` (Chapter 7: Evaluation), the first thing it does is use a `TelemetryProcessor` to understand the trace data before it can evaluate criteria against it.
- **Tracing Visualization (Console Output):** The `RichConsoleSpanExporter` used for printing colored trace output (Chapter 6: Tracing) also uses a `TelemetryProcessor` to extract meaningful details from each span to display in a readable format.

## How it Works (Behind the Scenes)

Let's revisit the evaluation flow from Chapter 7: Evaluation and see where Telemetry Processing fits in.

When you call `evaluate_telemetry(test_case, telemetry_path)`:

1. `evaluate_telemetry` loads the raw JSON trace data from the file at `telemetry_path`.
2. It needs to figure out *which* framework generated this trace so it knows how to parse it. It calls a helper method, `TelemetryProcessor.determine_agent_framework`,

which inspects the trace data for framework-specific clues (like certain attribute names).
3. Once the framework is identified (e.g., `AgentFramework.SMOLAGENTS`), `evaluate_telemetry` calls the `TelemetryProcessor.create()` factory method, passing the identified framework.
4. `TelemetryProcessor.create()` looks up the correct framework-specific processor class (e.g., `SmolagentsTelemetryProcessor`) and creates an instance of it.
5. `evaluate_telemetry` then calls methods on this processor instance, such as `processor.extract_hypothesis_answer(trace)` to get the agent's final response and `processor.extract_evidence(trace)` to get a formatted summary of key steps (tool calls, LLM interactions) from the trace.
6. This standardized final answer and evidence are then used by the evaluators (like `HypothesisEvaluator` and `CheckpointEvaluator`) to perform the actual evaluation using the LLM judge.

Here's a simplified sequence diagram showing this flow:

```
sequenceDiagram
    participant YourCode
    participant EvaluateTelemetry as evaluate_telemetry()
    participant TelemetryProcessorBase as TelemetryProcessor (Base Class)
    participant FrameworkProcessor as Specific Framework Processor (e.g., SmolagentsTelemet

    YourCode->>EvaluateTelemetry: Call evaluate_telemetry(..., trace_path)
    EvaluateTelemetry->>EvaluateTelemetry: Load raw trace data
    EvaluateTelemetry->>TelemetryProcessorBase: Call determine_agent_framework(raw_trace)
    TelemetryProcessorBase-->>EvaluateTelemetry: Return AgentFramework (e.g., SMOLAGENTS)
    EvaluateTelemetry->>TelemetryProcessorBase: Call create(SMOLAGENTS)
    TelemetryProcessorBase->>FrameworkProcessor: Create SmolagentsTelemetryProcessor()
    FrameworkProcessor-->>TelemetryProcessorBase: Return instance
    TelemetryProcessorBase-->>EvaluateTelemetry: Return processor instance
    EvaluateTelemetry->>FrameworkProcessor: Call extract_hypothesis_answer(raw_trace)
    FrameworkProcessor->>FrameworkProcessor: Parse raw trace (Smolagents logic)
    FrameworkProcessor-->>EvaluateTelemetry: Return standardized final answer
    EvaluateTelemetry->>FrameworkProcessor: Call extract_evidence(raw_trace)
    FrameworkProcessor->>FrameworkProcessor: Parse raw trace & format (Smolagents logic)
    FrameworkProcessor-->>EvaluateTelemetry: Return standardized evidence string
    EvaluateTelemetry->>EvaluateTelemetry: Use standardized data for evaluation...
```

## Looking at the Code

Let's peek at the code to see how this is structured.

First, the base `TelemetryProcessor` class in `src/any_agent/telemetry/base.py` defines the common interface and the factory method:

```
# From src/any_agent/telemetry/base.py (simplified)
```

```python
from __future__ import annotations
from abc import ABC, abstractmethod
from typing import TYPE_CHECKING, Any, ClassVar

from any_agent import AgentFramework # Our Enum from Chapter 3
from any_agent.logging import logger

if TYPE_CHECKING:
    from collections.abc import Mapping, Sequence

class TelemetryProcessor(ABC): # It's an Abstract Base Class
    """Base class for processing telemetry data."""

    @classmethod
    def create(cls, agent_framework_raw: AgentFramework | str) -> TelemetryProcessor:
        """Factory method to create the appropriate telemetry processor."""
        agent_framework = AgentFramework.from_string(agent_framework_raw)

        # --- This is where it maps the framework to the specific class ---
        if agent_framework is AgentFramework.LANGCHAIN:
            from any_agent.telemetry.langchain_telemetry import LangchainTelemetryProcessor
            return LangchainTelemetryProcessor()
        if agent_framework is AgentFramework.SMOLAGENTS:
            from any_agent.telemetry.smolagents_telemetry import SmolagentsTelemetryProcesso
            return SmolagentsTelemetryProcessor()
        # ... other frameworks ...
        # -----------------------------------------------------------------
        else:
             msg = f"Unsupported framework: {agent_framework}"
             raise NotImplementedError(msg)

    @staticmethod
    def determine_agent_framework(trace: Sequence[Mapping[str, Any]]) -> AgentFramework:
        """Determine the agent type based on the trace."""
        # --- This logic inspects the raw trace data ---
        for span in trace:
            if "smolagents.max_steps" in span.get("attributes", {}):
                logger.info("Agent type is SMOLAGENTS")
                return AgentFramework.SMOLAGENTS
            if "langchain" in span.get("attributes", {}).get("input.value", ""):
                 logger.info("Agent type is LANGCHAIN")
                 return AgentFramework.LANGCHAIN
            # ... other checks for other frameworks ...
        # --------------------------------------------
        msg = "Could not determine agent type from trace..."
        raise ValueError(msg)
    @abstractmethod
    def extract_hypothesis_answer(self, trace: Sequence[Mapping[str, Any]]) -> str:
        """Extract the hypothesis agent final answer from the trace."""
        # Each subclass MUST implement this based on its framework's trace structure

    @abstractmethod
    def extract_evidence(self, telemetry: Sequence[Mapping[str, Any]]) -> str:
        """Extract relevant telemetry evidence (formatted string)."""
        # Each subclass MUST implement this

    # ... other abstract methods for extracting specific interaction types ...
    @abstractmethod
    def _extract_llm_interaction(self, span: Mapping[str, Any]) -> Mapping[str, Any]: pass
    @abstractmethod
    def _extract_tool_interaction(self, span: Mapping[str, Any]) -> Mapping[str, Any]: pass
    # ... etc.
```

**Explanation:**

- `TelemetryProcessor` is an `ABC` (Abstract Base Class), meaning you can't create an instance of *this* class directly. It defines the blueprint.
- The `@classmethod create()` is the factory. You call `TelemetryProcessor.create(...)`, and it returns an instance of the *correct framework-specific subclass*.
- `determine_agent_framework()` is a helper that tries to guess the framework by looking for unique patterns in the raw trace data.
- Methods like `extract_hypothesis_answer` and `extract_evidence` are `@abstractmethod`, meaning any class that inherits from `TelemetryProcessor` *must* provide its own implementation of these methods.

Now, let's look at a simplified example of a framework-specific processor, like `SmolagentsTelemetryProcessor` in `src/any_agent/telemetry/smolagents_telemetry.py`:

```
# From src/any_agent/telemetry/smolagents_telemetry.py (simplified)
import json
from collections.abc import Mapping, Sequence
from typing import Any

from any_agent import AgentFramework
from any_agent.telemetry import TelemetryProcessor # Import the base class

class SmolagentsTelemetryProcessor(TelemetryProcessor): # It inherits from the base
    """Processor for SmoL Agents telemetry data."""

    def _get_agent_framework(self) -> AgentFramework:
        # This tells the base class which framework this processor handles
        return AgentFramework.SMOLAGENTS
```

```python
def extract_hypothesis_answer(self, trace: Sequence[Mapping[str, Any]]) -> str:
    """Extract the hypothesis agent final answer from the trace."""
    # --- This is the Smolagents-specific logic ---
    for span in reversed(trace): # Look backwards from the end
        # Find the span that represents the final agent output
        if span.get("attributes", {}).get("openinference.span.kind") == "AGENT":
            # The final answer is stored in the 'output.value' attribute for Smolagents
            return str(span.get("attributes", {}).get("output.value", ""))
    # ---------------------------------------------
    msg = "No agent final answer found in trace for Smolagents"
    raise ValueError(msg)

def _extract_llm_interaction(self, span: Mapping[str, Any]) -> dict[str, Any]:
    """Extract LLM interaction details from a span (Smolagents specific)."""
    attributes = span.get("attributes", {})
    span_info = {
        "model": attributes.get("llm.model_name", "Unknown model"),
        "type": "reasoning",
    }
    # --- Smolagents specific attribute names for input/output ---
    if "llm.input_messages.0.message.content" in attributes:
         span_info["input"] = attributes["llm.input_messages.0.message.content"]
    if "llm.output_messages.0.message.content" in attributes:
         span_info["output"] = attributes["llm.output_messages.0.message.content"]
    # -----------------------------------------------------------
    return span_info

def _extract_tool_interaction(self, span: Mapping[str, Any]) -> dict[str, Any]:
    """Extract tool interaction details from a span (Smolagents specific)."""
    attributes = span.get("attributes", {})
    tool_info = {
        "tool_name": attributes.get("tool.name", span.get("name", "Unknown tool")),
        "status": "success" if span.get("status", {}).get("status_code") == "OK" else "
    }
    # --- Smolagents specific attribute names and parsing for tool input/output ---
    if "input.value" in attributes:
        try:
            input_value = json.loads(attributes["input.value"])
            if "kwargs" in input_value: # Smolagents often puts tool args in 'kwargs'
                tool_info["input"] = input_value["kwargs"]
            else:
                tool_info["input"] = input_value
        except (json.JSONDecodeError, TypeError):
            tool_info["input"] = attributes["input.value"]

    if "output.value" in attributes:
        try:
            output_value = json.loads(attributes["output.value"])
            tool_info["output"] = output_value
        except (json.JSONDecodeError, TypeError):
            tool_info["output"] = attributes["output.value"]
    # ----------------------------------------------------------------------------
    return tool_info

# ... similar methods for _extract_chain_interaction, _extract_agent_interaction ...

def extract_evidence(self, telemetry: Sequence[Mapping[str, Any]]) -> str:
     """Extract relevant telemetry evidence (formatted string)."""
     # This method in the base class calls the _extract_... methods
     # implemented here and formats the output consistently.
     return super().extract_evidence(telemetry)
```

**Explanation:**

- `SmolagentsTelemetryProcessor` inherits from `TelemetryProcessor`.
- It implements the abstract methods like `extract_hypothesis_answer`, `_extract_llm_interaction`, `_extract_tool_interaction`, etc.
- Inside these methods, it contains the specific logic needed to find and extract the relevant data from a raw trace generated by the Smolagents framework. It knows which attributes to look for and how to parse their values (e.g., handling JSON strings, looking for `kwargs`).
- The `extract_evidence` method (which is implemented in the base class but calls the framework-specific `_extract_...` methods) then takes the extracted, standardized data and formats it into a readable string, often used for the LLM judge prompt or console output.

Similar processor classes exist for LangChain (`LangchainTelemetryProcessor`), LlamaIndex (`LlamaIndexTelemetryProcessor`), and OpenAI (`OpenAITelemetryProcessor`), each containing the specific parsing logic for their respective frameworks.

This design ensures that even though the raw trace data might look different, the information extracted by the `TelemetryProcessor` is always in a consistent format, allowing the rest of `any-agent` (evaluation, visualization) to work seamlessly regardless of the underlying framework.

## Conclusion

In this chapter, you learned that Telemetry Processing is the crucial step that translates raw, framework-specific trace data into a standardized format that `any-agent`'s evaluation and visualization components can understand. You saw that the `TelemetryProcessor` base class defines the interface, while framework-specific subclasses (like `SmolagentsTelemetryProcessor`) contain the logic to parse traces from their respective frameworks. You also understood that you typically don't interact directly with these processors, but they work behind the scenes when you use features like `evaluate_telemetry`.

Telemetry Processing is a key abstraction that upholds `any-agent`'s goal of providing a unified experience across different agent frameworks, ensuring that powerful features like tracing and evaluation are consistently available.

This chapter concludes our exploration of the core components of the `any-agent` project as outlined in this tutorial structure. You now have a foundational understanding of how to configure agents, use the unified interface, leverage different frameworks and tools (including MCP), and observe and evaluate agent execution using tracing and telemetry processing.

Congratulations on completing the tutorial! You're now ready to start building and experimenting with your own agents using `any-agent`.

`any-agent` is a Python library designed to provide a *single, unified interface* for interacting with many different **AI agent frameworks**. It allows developers to **switch between frameworks** easily without significant code changes. The library also includes built-in support for **tracing** agent execution steps and **evaluating** agent performance based on these traces.

**Source Repository:** https://github.com/mozilla-ai/any-agent

```
flowchart TD
    A0["AnyAgent (Unified Interface)"]
    A1["Agent Frameworks"]
    A2["Agent Configuration"]
    A3["Tools"]
    A4["Tracing"]
    A5["Evaluation"]
    A6["Telemetry Processing"]
    A7["MCP Tools"]
    A0 -- "Uses Frameworks" --> A1
    A0 -- "Configured by" --> A2
    A0 -- "Uses Tools" --> A3
    A0 -- "Uses Tracing" --> A4
    A0 -- "Uses MCP Tools" --> A7
    A2 -- "Configures" --> A4
    A3 -- "Includes MCPParams" --> A7
    A4 -- "Framework Specific" --> A1
    A4 -- "Uses Telemetry Processing" --> A6
    A5 -- "Analyzes Traces from" --> A4
    A5 -- "Processes Traces with" --> A6
    A6 -- "Framework Specific" --> A1
    A7 -- "Framework Specific" --> A1
    A7 -- "Provides Tools" --> A3
```

## Chapters