Agent Bridge

1 Overview

While Inspect provides facilities for native agent development, you can also very easily integrate agents created with 3rd party frameworks like AutoGen or LangChain, or use fully custom agents you have developed or ported from a research paper. The basic mechanism for integrating external agents works like this:

1. Write an agent function that takes a sample `dict` as input and a returns a results `dict` with output. This function won't have any dependencies on Inspect, rather it will depend on whatever agent framework or custom code you are using.
2. This function should use the OpenAI API for model access, however calls to the OpenAI API will be *redirected* to Inspect (using whatever model is configured for the current task).
3. Use the agent function with Inspect by passing it to the `bridge()` function, which will turn it into a standard Inspect `Solver`.

2 Example

Here is an agent written with the AutoGen framework. You'll notice that it is structured similar to an Inspect `Solver` where an enclosing function returns the function which handles the sample (this enables you to share initialisation code and pass options to configure the behaviour of the agent):

```
from typing import Any, cast

from autogen_agentchat.agents import AssistantAgent
from autogen_agentchat.conditions import SourceMatchTermination
from autogen_agentchat.messages import TextMessage
from autogen_agentchat.teams import RoundRobinGroupChat
from autogen_core.models import ModelInfo
from autogen_ext.agents.web_surfer import MultimodalWebSurfer
from autogen_ext.models.openai import OpenAIChatCompletionClient

def web_surfer_agent():

    # Use OpenAI interface (redirected to Inspect model)
    model = OpenAIChatCompletionClient(
        model="inspect",
        model_info=ModelInfo(
            vision=True, function_calling=True,
            json_output=False, family="unknown"
        ),
```

```
    )

    # Sample handler
    async def run(sample: dict[str, Any]) -> dict[str, Any]:
        # Read input (convert from OpenAI format)
        input = [
            TextMessage(source=msg["role"], content=str(msg["content"]))
            for msg in sample["input"]
        ]

        # Create agents and team
        web_surfer = MultimodalWebSurfer("web_surfer", model)
        assistant = AssistantAgent("assistant", model)
        termination = SourceMatchTermination("assistant")
        team = RoundRobinGroupChat(
            [web_surfer, assistant],
            termination_condition=termination
        )

        # Run team
        result = await team.run(task=input)

        # Extract output from last message and return
        message = cast(TextMessage, result.messages[-1])
        return dict(output=message.content)

    return run
```

To use this agent in an Inspect `Task`, import it and use the `bridge()` function:

```
from agent import web_surfer_agent

from inspect_ai import Task, task
from inspect_ai.dataset import json_dataset
from inspect_ai.scorer import model_graded_fact
from inspect_ai.solver import bridge

@task
def research() -> Task:
    return Task(
        dataset=json_dataset("dataset.json"),
        solver=bridge(web_surfer_agent()),
        scorer=model_graded_fact(),
    )
```

The `bridge()` function takes the agent function and hooks it up to a standard Inspect `Solver`, updating the `TaskState` and providing the means of redirecting OpenAI calls to the current Inspect model.

You can find a runable version of this example as well as one for LangChain here:

- AutoGen Example
- LangChain Example

3 Types

In the example above we reference two `dict` fields from the agent function interface:

`sample["input"]`	List of OpenAI compatible chat messages.
`result["output"]`	Agent output as a string

For many agents these fields will be all you need. In some circumstances other available fields will be useful. Here are the full type declarations for the `sample` and `result`:

```
from typing import NotRequired, TypedDict

from openai.types.chat import ChatCompletionMessageParam

class SampleDict(TypedDict):
    sample_id: str
    epoch: int
    input: list[ChatCompletionMessageParam]
    metadata: dict[str, Any]
    target: str | list[str]

class ResultDict(TypedDict):
    output: str
    messages: NotRequired[list[ChatCompletionMessageParam]]
    scores: NotRequired[dict[str, ScoreDict]]
```

You aren't required to use these types exactly (they merely document the interface) so long as you consume and produce `dict` values that match their declarations (the result `dict` is type validated at runtime).

Returning `messages` is not required—messages are automatically synced to the task state during generate, return `messages` if you want to customise the default behaviour.

3.1 Scores

Returning `scores` is also optional and not common (most agent will rely on native Inspect scorers, this is here as an escape hatch for agents that want to do their own scoring). If you do return scores use this format (which is based on Inspect `Score` objects):

```
class ScoreDict(TypedDict):
    value: (
        str
        | int
        | float
        | bool
        | list[str | int | float | bool]
        | dict[str, str | int | float | bool | None]
    )
    answer: NotRequired[str]
    explanation: NotRequired[str]
    metadata: NotRequired[dict[str, Any]]
```

4 CLI Usage

Above we import the `web_surfer_agent()` directly as a Python function. It's also possible to reference external agents at the command line using the `--solver` parameter. For example:

```
inspect eval task.py --solver agent.py
```

This also works with `--solver` arguments passed via `-S`. For example:

```
inspect eval task.py --solver agent.py -S max_requests=5
```

The `agent.py` source file will be searched for public top level functions that include `agent` in their name. If you want to explicitly reference an agent function you can do that as follows:

```
inspect eval task.py --solver agent.py@web_surfer_agent
```

5 Models

As demonstrated above, communication with Inspect models is done by using the OpenAI API with `model="inspect"`. You can use the same technique to interface with other Inspect models. To do this, preface the model name with "inspect" followed by the rest of the fully qualified model name.

For example, in a LangChain agent, you would use this syntax:

```
model = ChatOpenAI(model="inspect/google/gemini-1.5-pro")
```

6 Sandboxes

If you need to execute untrusted LLM generated code in your agent, you can still use the Inspect `sandbox()` facility within bridged agent functions. Typically agent tools that can run code can be customised with an executor, and this is where you would plug in the Inspect `sandbox()`.

For example, the AutoGen `PythonCodeExecutionTool` takes a `CodeExecutor` in its constructor. AutoGen provides several built in code executors (e.g. local, docker, azure, etc.) and you can create custom ones. For example, you could create an `InspectSandboxCodeExecutor` which in turn delegates to the `sandbox().exec()` function.

7 Transcript

Custom agents run through the `bridge()` function still get most of the benefit of the Inspect transcript and log viewer. All model calls are captured and produce the same transcript output as when using normal solver. The message history is also automatically captured and logged.

Calls to the Python `logging` module for levels `info` and above are also handled as normal and show up within sample transcripts.

If you want to use additional features of Inspect transcripts (e.g. steps, markdown logging, etc.) you can still import and use the `transcript` function as normal. For example:

```
from inspect_ai.log import transcript

transcript().info("custom *markdown* content")
```

If your agent code that calls `transcript()` is not running within Inspect it will simply be ignored (no runtime error will occur).