AI Service Deployment Notebook

This notebook contains steps and code to test, promote, and deploy an Agent as an AI Service.

Note: Notebook code generated using Agent Lab will execute successfully. If code is modified or reordered, there is no guarantee it will successfully execute. For details, see: [Saving your work in Agent Lab as a notebook.](#)

Some familiarity with Python is helpful. This notebook uses Python 3.11.

Contents

This notebook contains the following parts:

1. **Setup**
2. **Initialize all the variables needed by the AI Service**
3. **Define the AI service function**
4. **Deploy an AI Service**
5. **Test the deployed AI Service**

1. Set up the environment

Before you can run this notebook, you must perform the following setup tasks:

Connection to WML

This cell defines the credentials required to work with watsonx API for both the execution in the project, as well as the deployment and runtime execution of the function.

Action: Provide the IBM Cloud personal API key. For details, see [documentation](#).

In [ ]:

```
import os
from ibm_watsonx_ai import APIClient, Credentials
import getpass

credentials = Credentials(
    url="https://us-south.ml.cloud.ibm.com",
    api_key=getpass.getpass("Please enter your api key (hit enter): ")
)
```

In [ ]:

```
client = APIClient(credentials)
```

Connecting to a space

A space will be be used to host the promoted AI Service.

In [ ]:

```
space_id = "7c318d93-91fe-43e4-9916-5572f3ccd4a8"
client.set.default_space(space_id)
```

Promote asset(s) to space

We will now promote assets we will need to stage in the space so that we can access their data from the AI

We will now promote assets we will need to stage in the space so that we can access their data from the AI service.

In [ ]:

```
source_project_id = "76768c52-e794-4361-81ae-c6d303d1a9ad"
```

2. Create the AI service function

We first need to define the AI service function

2.1 Define the function

In [ ]:

```
params = {
    "space_id": space_id,
}

def gen_ai_service(context, params = params, **custom):
    # import dependencies
    from langchain_ibm import ChatWatsonx
    from ibm_watsonx_ai import APIClient
    from ibm_watsonx_ai.foundation_models.utils import Tool, Toolkit
    from langchain_core.messages import AIMessage, HumanMessage
    from langgraph.checkpoint.memory import MemorySaver
    from langgraph.prebuilt import create_react_agent
    import json
    import requests

    model = "meta-llama/llama-3-3-70b-instruct"

    service_url = "https://us-south.ml.cloud.ibm.com"
    # Get credentials token
    credentials = {
        "url": service_url,
        "token": context.generate_token()
    }

    # Setup client
    client = APIClient(credentials)
    space_id = params.get("space_id")
    client.set.default_space(space_id)



    def create_chat_model(watsonx_client):
        parameters = {
            "frequency_penalty": 0,
            "max_tokens": 200,
            "presence_penalty": 0,
            "temperature": 0,
            "top_p": 1
        }

        chat_model = ChatWatsonx(
            model_id=model,
            url=service_url,
            space_id=space_id,
            params=parameters,
            watsonx_client=watsonx_client,
        )
        return chat_model


    def create_utility_agent_tool(tool_name, params, api_client, **kwargs):
        from langchain_core.tools import StructuredTool
        utility_agent_tool = Toolkit(
            api_client=api_client
        ).get_tool(tool_name)

        tool_description = utility_agent_tool.get("description")

        if (kwargs.get("tool_description")):
            tool_description = kwargs.get("tool_description")
        elif (utility_agent_tool.get("agent_description")):
            tool_description = utility_agent_tool.get("agent_description")

        tool_schema = utility_agent_tool.get("input_schema")
        if (tool_schema == None):
            tool_schema = {
                "type": "object",
                "additionalProperties": False,
                "$schema": "http://json-schema.org/draft-07/schema#",
                "properties": {
                    "input": {
                        "description": "input for the tool",
                        "type": "string"
                    }
                }
            }

        def run_tool(**tool_input):
            query = tool_input
            if (utility_agent_tool.get("input_schema") == None):
                query = tool_input.get("input")

            results = utility_agent_tool.run(
                input=query,
                config=params
            )

            return results.get("output")

        return StructuredTool(
            name=tool_name,
            description = tool_description,
            func=run_tool,
            args_schema=tool_schema
        )


    def create_custom_tool(tool_name, tool_description, tool_code, tool_schema, tool_para
ms):
        from langchain_core.tools import StructuredTool
        import ast

        def call_tool(**kwargs):
            tree = ast.parse(tool_code, mode="exec")
            custom_tool_functions = [ x for x in tree.body if isinstance(x, ast.Function
Def) ]
            function_name = custom_tool_functions[0].name
            compiled_code = compile(tree, 'custom_tool', 'exec')
            namespace = tool_params if tool_params else {}
            exec(compiled_code, namespace)
            return namespace[function_name](**kwargs)

        tool = StructuredTool(
            name=tool_name,
            description = tool_description,
            func=call_tool,
            args_schema=tool_schema
        )
        return tool

    def create_custom_tools():
        custom_tools = []


    def create_tools(inner_client, context):
        tools = []
        config = None
        tools.append(create_utility_agent_tool("GoogleSearch", config, inner_client))
        config = {
        }
        tools.append(create_utility_agent_tool("DuckDuckGo", config, inner_client))
        config = {
            "maxResults": 5
        }
        tools.append(create_utility_agent_tool("Wikipedia", config, inner_client))
        config = {
        }
        tools.append(create_utility_agent_tool("Weather", config, inner_client))
        config = {
        }
        tools.append(create_utility_agent_tool("WebCrawler", config, inner_client))
        return tools

    def create_agent(model, tools, messages):
        memory = MemorySaver()
        instructions = """# Notes
- Use markdown syntax for formatting code snippets, links, JSON, tables, images, files.
- Any HTML tags must be wrapped in block quotes, for example ```<html>```.
- When returning code blocks, specify language.
- Sometimes, things don't go as planned. Tools may not provide useful information on the
first few tries. You should always try a few different approaches before declaring the pr
oblem unsolvable.
- When the tool doesn't give you what you were asking for, you must either use another to
ol or a different tool input.
- When using search engines, you try different formulations of the query, possibly even i
n a different language.
- You cannot do complex calculations, computations, or data manipulations without using t
ools.
- If you need to call a tool to compute something, always call it instead of saying you w
ill call it.

If a tool returns an IMAGE in the result, you must include it in your answer as Markdown.

Example:

Tool result: IMAGE({commonApiUrl}/wx/v1-beta/utility_agent_tools/cache/images/plt-04e3c91
ae04b47f8934a4e6b7d1fdc2c.png)
Markdown to return to user: ![Generated image]({commonApiUrl}/wx/v1-beta/utility_agent_to
ols/cache/images/plt-04e3c91ae04b47f8934a4e6b7d1fdc2c.png)

You are a multilingual AI health assistant. A user will describe their symptoms in natura
l language.
Your job is to:
1. List possible conditions (based on WHO and trusted medical sources).
2. Rate the urgency: Low, Medium, High, Emergency.
3. Give home remedies and preventive care.
4. Tell if a doctor should be consulted.
5. Provide this info in user's preferred language.

Never give direct medical diagnosis. Avoid treatment instructions that require prescripti
ons. Encourage users to seek licensed healthcare professionals.

Respond in bullet format. Be polite, professional, and helpful.
"""
        for message in messages:
            if message["role"] == "system":
                instructions += message["content"]
        graph = create_react_agent(model, tools=tools, checkpointer=memory, state_modifi
er=instructions)
        return graph

    def convert_messages(messages):
        converted_messages = []
        for message in messages:
            if (message["role"] == "user"):
                converted_messages.append(HumanMessage(content=message["content"]))
            elif (message["role"] == "assistant"):
```

```
                converted_messages.append(AIMessage(content=message["content"]))
        return converted_messages

    def generate(context):
        payload = context.get_json()
        messages = payload.get("messages")
        inner_credentials = {
            "url": service_url,
            "token": context.get_token()
        }

        inner_client = APIClient(inner_credentials)
        model = create_chat_model(inner_client)
        tools = create_tools(inner_client, context)
        agent = create_agent(model, tools, messages)

        generated_response = agent.invoke(
            { "messages": convert_messages(messages) },
            { "configurable": { "thread_id": "42" } }
        )

        last_message = generated_response["messages"][-1]
        generated_response = last_message.content

        execute_response = {
            "headers": {
                "Content-Type": "application/json"
            },
            "body": {
                "choices": [{
                    "index": 0,
                    "message": {
                       "role": "assistant",
                       "content": generated_response
                    }
                }]
            }
        }

        return execute_response

    def generate_stream(context):
        print("Generate stream", flush=True)
        payload = context.get_json()
        headers = context.get_headers()
        is_assistant = headers.get("X-Ai-Interface") == "assistant"
        messages = payload.get("messages")
        inner_credentials = {
            "url": service_url,
            "token": context.get_token()
        }
        inner_client = APIClient(inner_credentials)
        model = create_chat_model(inner_client)
        tools = create_tools(inner_client, context)
        agent = create_agent(model, tools, messages)

        response_stream = agent.stream(
            { "messages": messages },
            { "configurable": { "thread_id": "42" } },
            stream_mode=["updates", "messages"]
        )

        for chunk in response_stream:
            chunk_type = chunk[0]
            finish_reason = ""
            usage = None
            if (chunk_type == "messages"):
                message_object = chunk[1][0]
                if (message_object.type == "AIMessageChunk" and message_object.content !
= ""):
                    message = {
                        "role": "assistant",
                        "content": message_object.content
                    }
                else:
                    continue
            elif (chunk_type == "updates"):
                update = chunk[1]
                if ("agent" in update):
                    agent = update["agent"]
                    agent_result = agent["messages"][0]
                    if (agent_result.additional_kwargs):
                        kwargs = agent["messages"][0].additional_kwargs
                        tool_call = kwargs["tool_calls"][0]
                        if (is_assistant):
                            message = {
                                "role": "assistant",
                                "step_details": {
                                    "type": "tool_calls",
                                    "tool_calls": [
                                        {
                                            "id": tool_call["id"],
                                            "name": tool_call["function"]["name"],
                                            "args": tool_call["function"]["arguments"]
                                        }
                                    ]
                                }
                            }
                        else:
                            message = {
                                "role": "assistant",
                                "tool_calls": [
                                    {
                                        "id": tool_call["id"],
                                        "type": "function",
                                        "function": {
                                            "name": tool_call["function"]["name"],
                                            "arguments": tool_call["function"]["argument
s"]
                                        }
                                    }
                                ]
                            }
                    elif (agent_result.response_metadata):
                        # Final update
                        message = {
                            "role": "assistant",
                            "content": agent_result.content
                        }
                        finish_reason = agent_result.response_metadata["finish_reason"]
                        if (finish_reason):
                            message["content"] = ""

                        usage = {
                            "completion_tokens": agent_result.usage_metadata["output_tok
ens"],
                            "prompt_tokens": agent_result.usage_metadata["input_tokens"]
,
                            "total_tokens": agent_result.usage_metadata["total_tokens"]
                        }
                elif ("tools" in update):
                    tools = update["tools"]
                    tool_result = tools["messages"][0]
                    if (is_assistant):
                        message = {
                            "role": "assistant",
                            "step_details": {
                                "type": "tool_response",
                                "id": tool_result.id,
                                "tool_call_id": tool_result.tool_call_id,
                                "name": tool_result.name,
                                "content": tool_result.content
                            }
                        }
                    else:
                        message = {
                            "role": "tool",
                            "id": tool_result.id,
                            "tool_call_id": tool_result.tool_call_id,
                            "name": tool_result.name,
                            "content": tool_result.content
                        }
                else:
                    continue

            chunk_response = {
                "choices": [{
                    "index": 0,
                    "delta": message
                }]
            }
            if (finish_reason):
                chunk_response["choices"][0]["finish_reason"] = finish_reason
            if (usage):
                chunk_response["usage"] = usage
            yield chunk_response

    return generate, generate_stream
```

2.2 Test locally

In [ ]:

```
# Initialize AI Service function locally
from ibm_watsonx_ai.deployments import RuntimeContext

context = RuntimeContext(api_client=client)

streaming = False
findex = 1 if streaming else 0
local_function = gen_ai_service(context,  space_id=space_id)[findex]
messages = []
```

In [ ]:

```
local_question = "Change this question to test your function"

messages.append({ "role" : "user", "content": local_question })

context = RuntimeContext(api_client=client, request_payload_json={"messages": messages})

response = local_function(context)

result = ''

if (streaming):
    for chunk in response:
        print(chunk, end="\n\n", flush=True)
else:
    print(response)
```

3. Store and deploy the AI Service

Before you can deploy the AI Service, you must store the AI service in your watsonx.ai repository.

In [ ]:

```
# Look up software specification for the AI service
software_spec_id_in_project = "45f12dfe-aa78-5b8d-9f38-0ee223c47309"
software_spec_id = ""

try:
    software_spec_id = client.software_specifications.get_id_by_name("runtime-24.1-py3.11
")
except:
    software_spec_id = client.spaces.promote(software_spec_id_in_project, source_project
_id, space_id)
```

In [ ]:

```
# Define the request and response schemas for the AI service
request_schema = {
    "application/json": {
        "$schema": "http://json-schema.org/draft-07/schema#",
        "type": "object",
        "properties": {
            "messages": {
                "title": "The messages for this chat session.",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "role": {
                            "title": "The role of the message author.",
                            "type": "string",
                            "enum": ["user","assistant"]
                        },
                        "content": {
                            "title": "The contents of the message.",
                            "type": "string"
                        }
                    },
                    "required": ["role","content"]
                }
            }
        },
        "required": ["messages"]
    }
}

response_schema = {
    "application/json": {
        "oneOf": [{"$schema":"http://json-schema.org/draft-07/schema#","type":"object","
description":"AI Service response for /ai_service_stream","properties":{"choices":{"descr
iption":"A list of chat completion choices.","type":"array","items":{"type":"object","pr
operties":{"index":{"type":"integer","title":"The index of this result."},"delta":{"desc
ription":"A message result.","type":"object","properties":{"content":{"description":"The
contents of the message.","type":"string"},"role":{"description":"The role of the author
of this message.","type":"string"}},"required":["role"]}}}}},"required":["choices"]},{"$s
chema":"http://json-schema.org/draft-07/schema#","type":"object","description":"AI Servic
e response for /ai_service","properties":{"choices":{"description":"A list of chat comple
tion choices","type":"array","items":{"type":"object","properties":{"index":{"type":"int
eger","description":"The index of this result."},"message":{"description":"A message resu
lt.","type":"object","properties":{"role":{"description":"The role of the author of this
message.","type":"string"},"content":{"title":"Message content.","type":"string"}},"requi
red":["role"]}}}}},"required":["choices"]}]
    }
}
```

In [ ]:

```
# Store the AI service in the repository
ai_service_metadata = {
    client.repository.AIServiceMetaNames.NAME: "symptoms",
    client.repository.AIServiceMetaNames.DESCRIPTION: "",
    client.repository.AIServiceMetaNames.SOFTWARE_SPEC_ID: software_spec_id,
    client.repository.AIServiceMetaNames.CUSTOM: {},
    client.repository.AIServiceMetaNames.REQUEST_DOCUMENTATION: request_schema,
    client.repository.AIServiceMetaNames.RESPONSE_DOCUMENTATION: response_schema,
    client.repository.AIServiceMetaNames.TAGS: ["wx-agent"]
}

ai_service_details = client.repository.store_ai_service(meta_props=ai_service_metadata, a
i_service=gen_ai_service)
```

In [ ]:

```
# Get the AI Service ID

ai_service_id = client.repository.get_ai_service_id(ai_service_details)
```

In [ ]:

```
# Deploy the stored AI Service
deployment_custom = {
    "avatar_icon": "Chemistry",
    "avatar_color": "supportCautionMajor",
    "placeholder_image": "placeholder2.png"
}
deployment_metadata = {
    client.deployments.ConfigurationMetaNames.NAME: "symptoms",
    client.deployments.ConfigurationMetaNames.ONLINE: {},
    client.deployments.ConfigurationMetaNames.CUSTOM: deployment_custom,
    client.deployments.ConfigurationMetaNames.DESCRIPTION: "predicts the disease by sympt
oms",
    client.repository.AIServiceMetaNames.TAGS: ["wx-agent"]
}

function_deployment_details = client.deployments.create(ai_service_id, meta_props=deploy
ment_metadata, space_id=space_id)
```

4. Test AI Service

In [ ]:

```
# Get the ID of the AI Service deployment just created

deployment_id = client.deployments.get_id(function_deployment_details)
print(deployment_id)
```

In [ ]:

```
messages = []
remote_question = "Change this question to test your function"
messages.append({ "role" : "user", "content": remote_question })
payload = { "messages": messages }
```

In [ ]:

```
result = client.deployments.run_ai_service(deployment_id, payload)
if "error" in result:
    print(result["error"])
else:
    print(result)
```

Next steps

You successfully deployed and tested the AI Service! You can now view your deployment and test it as a REST API endpoint.

Copyrights