Build a multi-agent contract management system with BeeAI and Granite

In this tutorial, you will build a fully local multi-agent system with IBM® Granite® by using BeeAI in Python. These agents will collaborate to negotiate a contractual agreement for landscaping services between two companies by taking into account market trends and internal budget constraints. The workflow will be composed of a budget advisor agent, contract synthesizer agent, web search agent and procurement advisor agent. Given the contract, budget data, service industry and company names provided by the user, the agents collaborate to produce an email to negotiate the terms of the contract in favor of the client.

What is BeeAI?

BeeAI, released by IBM Research® and now donated to the Linux® Foundation, is an open-source agentic AI platform that provides developers with the ability to build AI agents from any framework.[1]

An artificial intelligence (AI) agent refers to a system or program built by using a large language model (LLM) to autonomously perform tasks on behalf of a user or another system by designing its workflow and by using utilizing available tools. AI agents are more advanced than traditional LLM chatbots as they can access predefined tools, plan future actions and require little to no human intervention to solve and automate complex problems.

The predecessor of BeeAI is the Bee Agent Framework, an open source framework specific to building single LLM agents. Conversely, BeeAI provides a more advanced ecosystem to build and orchestrate multi-agent workflows.

BeeAI is:

- Independent of framework.
- Available in both TypeScript and Python.
- Built on the IBM Research-designed Agent Communication Protocol (ACP), which takes Model Context Protocol (MCP) a step further by standardizing how agents communicate with one another. ACP brings agents from various frameworks such as LangGraph, crewAI and BeeAI into one constant runtime.[2]
- Integrated with Arize Phoenix, an open-source tool for tracing agent behavior, enabling observability. More information about debugging your agents through the available logging and telemetry can be found in the official documentation.
- Capable of running entirely locally on your machine. Agents can also be deployed to shared environments.
- Provides unified interfaces for various functionalities, including chat, embeddings and structured outputs, such as JSON, allowing for seamless model swapping without requiring any changes to your existing code.

Steps

This step-by-step guide can be found on our GitHub repository in the form of a Jupyter Notebook.

Step 1. Set up your environment

We first need to set up our environment by fulfilling some prerequisites.
1. In this tutorial, we will not be using an application programming interface (API) like the ones available through IBM® watsonx.ai® and OpenAI. Instead, we can install the latest version of Ollama to run the model locally.
The simplest way to install Ollama for macOS, Linux and Windows is through their webpage: https://ollama.com/download. This step will install a menu bar app to run the Ollama server in the background and keep you up to date with the latest releases.
As an alternative, you can install Ollama with Homebrew in your terminal:
brew install ollama
Powered by Granite
Powered by Granite

2. There are several LLMs that support tool calling such as Meta's latest Llama models and Mistral AI's Mistral models. For this tutorial, we will use IBM's open source Granite 3.3 model. This model features enhanced reasoning and instruction-following capabilities.[3] Pull the latest Granite 3.3 model by running the following command in your terminal.
ollama pull granite3.3:8b
Powered by Granite

3. To avoid package dependency conflicts, let's set up a virtual environment. To create a virtual environement with Python version 3.11.9, run the following command in your terminal.
python3.12 -m venv .venv
Powered by Granite

4. Your requirements.txt file should contain the following packages. These packages are necessary for building agents with the BeeAI framework and incorporating the necessary LangChain classes for tool initialization.
beeai-framework beeai-framework[duckduckgo] langchain-core langchain-community pandas
Powered by Granite

At the top of the fresh Python file, include the import statements for the necessary libraries and modules.
import asyncio import pandas as pd import os import traceback import sys from beeai_framework.backend import ChatModel from beeai_framework.tools.search.duckduckgo import DuckDuckGoSearchTool from beeai_framework.workflows.agent import AgentWorkflow, AgentWorkflowInput from beeai_framework.errors import FrameworkError from beeai_framework.adapters.langchain import LangChainTool from langchain_core.tools import StructuredTool from typing import Any
Powered by Granite

Step 2. Instantiate your agent and workflow

In an asynchronous main method using asyncio , let's incorporate the ChatModel and AgentWorkflow classes to instantiate our Granite LLM and workflow. We can simply provide the Ollama model ID as well as a workflow name. Instantiating this workflow allows us to add agents and create our multi-agent system. Add this main method to your Python file.
async def main() -> None: llm = ChatModel.from_name("ollama:granite3.3:8b")   workflow = AgentWorkflow(name="Procurement")
Powered by Granite

For a visual of the agentic workflow, refer to the following diagram.

We will assemble each component of this workflow in the following steps.

Step 3. Prompt the user for input

Our workflow relies on user input. The initial inputs required are the names of both the client and contractor companies. The user will be prompted with the following text upon execution of the workflow in a later step. Add the following code to the main method.
client_company = input("Please enter the client company: ") #Example: Company A
contractor_company = input("Please enter the contractor company: ") #Example: Company B
Powered by Granite

We will also need the names of the files containing the client company's budget report,budget-data.csv , as well as the file containing the contract between the two companies, contract.txt . In our example, the contract pertains to landscaping services provided by the contractor company to the client company. You can find the sample files in our GitHub repository. The project structure should resemble the following:
├── .venv/ # Virtual environment ├── bee-script.py # The Python script ├── contract.txt # The contract └── budget-data.csv # Client's budget report
Powered by Granite

In the following code, we also verify the file extensions to help ensure that they align with our anticipated format. If either file is the incorrect file type, the user will be prompted to try again.
client_budget_file = input(f"Enter the file name of the budget report for {client_company} (in the same directory level): ") #Example: budget_data.csv while os.path.splitext(client_budget_file)[1].lower() != ".csv": client_budget_file = input(f"Budget report must be in .csv format, please try again: ") contract_file = input(f"Enter the file name of the contract between {client_company} and {contractor_company} (in the same directory level): ") #Example: contract.txt while os.path.splitext(contract_file)[1].lower() != ".txt": contract_file = input(f"Contract must be in .txt format, please try again: ")
Powered by Granite

The last user input required is the industry of the service described in the contract. This input can be finance, construction or others.
service_industry = input(f"Enter the industry of the service described in this contract (e.g., finance, construction, etc.): ") #Example: landscaping
Powered by Granite

Step 4. Set up the budget tool

The first tool that we can build in our multi-agent system is for the budget advisor agent. This agent is responsible for reading the client's budget data. The function provided to the agent isget_budget_data , in which the CSV file containing the budget data is read and an error message is returned in case the file does not exist, or an unexpected error occurs. To access the file by using the file name provided by the user, we first need to retrieve the current directory. We can do so by using the followingos method.
current_directory = os.getcwd()
Powered by Granite

Now, let's set up the driving force of the agent, the get_budget_data function, which uses the current directory as well as the user input to access and read the file.

```
def get_budget_data():     try:         budget = pd.read_csv(os.path.join(current_directory,
client_budget_file))     except FileNotFoundError:         return client_budget_file + " not found. Please
check correct file name."     except Exception as e:         return f"An error occurred: {e}"     return
budget
```

Powered by Granite

To help ensure the appropriate usage of this tool, let's use LangChain's StructuredTool class. Here, we provide the function, tool name, tool description and set the return_direct parameter to be either true or false. This last parameter simply informs the agent whether to return the tool output directly or to synthesize it.

```
get_budget = StructuredTool.from_function(     func=get_budget_data,     name="GetBudgetData",
  description=f"Returns the budget data for {client_company}.",     return_direct=True, )
```

Powered by Granite

Using BeeAI's LangChain adapter, LangChainTool, we can finalize the initialization of our first tool.

```
budget_tool = LangChainTool[Any](get_budget)
```

Powered by Granite

Step 5. Set up the contract tool

The next tool that we can build is for the contract synthesizer agent. This agent is responsible for reading the contract between the client and contractor. The function provided to the agent is get_contract_data , in which the text file containing the contract is read, and an error message is returned in case the file does not exist, or an unexpected error occurs. The code required for this step is similar to step 3.

```
def get_contract_data():     try:         with open(os.path.join(current_directory, contract_file), 'r') as file:
            content = file.read()     except FileNotFoundError:         return contract_file + " not found.
Please check correct file name."     except Exception as e:         return f"An error occurred: {e}"
    return content
```

Powered by Granite

```
get_contract = StructuredTool.from_function(     func=get_contract_data,     name="GetContractData",
    description=f"Returns the contract details.",     return_direct=True, )
```

Powered by Granite

Step 6. Establish the agentic workflow

In this step, we can add the various agents to our workflow. Let's provide the budget advisor and contract synthesizer agents with their corresponding custom tools. We can also set the agent name, role, instructions, list of tools and LLM.

```
workflow.add_agent(     name="Budget Advisor",     role="A diligent budget advisor",
    instructions="You specialize in reading internal budget data in CSV format.",     tools=[budget_tool],
    llm=llm, ) workflow.add_agent(     name="Contract Synthesizer",     role="A diligent contract
synthesizer",     instructions=f"You specialize in reading contracts.",     tools=[contract_tool],
    llm=llm, )
```

Powered by Granite

To search the web for market trends in the relevant industry, we can create an agent with access to the prebuilt LangChain DuckDuckGoSearchTool . This tool fetches data from the web by using the DuckDuckGo search engine.

```
workflow.add_agent(     name="Web Search",     role="A web searcher.",     instructions=f"You can
search the web for market trends, specifically in the {service_industry} industry.",     tools=
[DuckDuckGoSearchTool()],     llm=llm, )
```
Powered by Granite

The fourth and final agent in our multi-agent system is the procurement advisor. This agent is responsible for using the information retrieved and synthesized by the other agents to formulate a convincing email to the contractor company in favor of the client. The email should consider market trends and the client's internal budget constraints to negotiate the terms of the contract. This agent does not require any external tools but rather, is driven by its instructions.

```
workflow.add_agent(     name="Procurement Advisor",     role="A procurement advisor",
    instructions=f"You write professional emails to {contractor_company} with convincing negotiations
that factor in market trends and internal budget constraints. You represent {client_company}.",
    llm=llm, )
```
Powered by Granite

Step 7. Execute the agentic workflow

We can now finalize our main method with all of our code so far. At the tail end of the main method, we can include the agentic workflow execution. Given theawait keyword, we can efficiently manage concurrent execution of tasks as well as await the execution of each task. Your Python file should contain the following code.

```
import asyncio import pandas as pd import os import traceback import sys from
beeai_framework.backend import ChatModel from beeai_framework.tools.search.duckduckgo import
DuckDuckGoSearchTool from beeai_framework.workflows.agent import AgentWorkflow,
AgentWorkflowInput from beeai_framework.errors import FrameworkError from
beeai_framework.adapters.langchain import LangChainTool from langchain_core.tools import
StructuredTool from typing import Any async def main() -> None:     llm =
ChatModel.from_name("ollama:granite3.3:8b")     workflow = AgentWorkflow(name="Procurement
Agent")     client_company = input("Please enter the client company: ")     contractor_company =
input("Please enter the contractor company name: ")     client_budget_file = input(f"Enter the file name
of the budget report for {client_company} (in the same directory level): ") while
os.path.splitext(client_budget_file)[1].lower() != ".csv":         client_budget_file = input(f"Budget
report must be in .csv format, please try again: ")     contract_file = input(f"Enter the file name of the
contract between {client_company} and {contractor_company} (in the same directory level): ") while
os.path.splitext(contract_file)[1].lower() != ".txt":         contract_file = input(f"Contract must be in .txt
format, please try again: ")     service_industry = input(f"Enter the industry of the service described in
this contract (e.g., finance, construction, etc.): ")
```
Powered by Granite

```
def get_budget_data():         try:             budget = pd.read_csv(os.path.join(current_directory,
client_budget_file))         except FileNotFoundError:             return client_budget_file + " not found.
Please check correct file name."         except Exception as e:             return f"An error occurred: {e}"
        return budget     get_budget = StructuredTool.from_function(             func=get_budget_data,
            name="GetBudgetData",             description=f"Returns the budget data for {client_company}.",
            return_direct=True,         )     budget_tool = LangChainTool[Any](get_budget)     def
get_contract_data():         try:             with open(os.path.join(current_directory, contract_file), 'r') as
file:                 content = file.read()         except FileNotFoundError:             return contract_file + " not
found. Please check correct file name."         except Exception as e:             return f"An error occurred:
{e}"         return content     get_contract = StructuredTool.from_function(
            func=get_contract_data,             name="GetContractData",             description=f"Returns the
contract details.",             return_direct=True,         )     contract_tool = LangChainTool[Any]
(get_contract)     workflow.add_agent(         name="Budget Advisor",         role="A diligent budget
advisor",         instructions="You specialize in reading internal budget data in CSV format.",
        tools=[budget_tool],         llm=llm,     )     workflow.add_agent(         name="Contract
Synthesizer",         role="A diligent contract synthesizer",         instructions=f"You specialize in reading
contracts.",        tools=[contract_tool],        llm=llm,     )     workflow.add_agent(        name="Web
Search",        role="A web searcher.",        instructions=f"You can search the web for market trends,
specifically in the {service_industry} industry.",        tools=[DuckDuckGoSearchTool()],
        llm=llm,     )     workflow.add_agent(        name="Procurement Advisor",        role="A
procurement advisor",        instructions=f"You write professional emails to {contractor_company}
with convincing negotiations that factor in market trends and internal budget constraints. You represent
{client_company}.",        llm=llm,     )     response = await workflow.run(        inputs=[
            AgentWorkflowInput(                prompt=f"Extract and summarize the key obligations,
deliverables, and payment terms from the contract between {client_company} and
{contractor_company}.",            ),            AgentWorkflowInput(                prompt=f"Analyze the
internal budget data for {client_company}.",            ),            AgentWorkflowInput(
                prompt=f"Write a formal email to {contractor_company}. In the email, negotiate the contract
terms in favor of {client_company}, factoring in market trends and internal budget constraints.",
            ),        ]     ).on(        "success",        lambda data, event: print(            f"-> Step '{data.step}'
has been completed with the following outcome.\n\n{data.state.final_answer}"            ),     )
    print("Final email: ")     print(response.state.final_answer) if __name__ == "__main__":     try:
        asyncio.run(main())     except FrameworkError as e:        traceback.print_exc()
        sys.exit(e.explain())
```

Powered by Granite

Use this *sample user input*:

- *Please enter the client company:* company A
- *Please enter the contractor company name:* company B
- *Enter the file name of the budget report for company A (in the same directory level):* budget_data.csv
- *Enter the file name of the contract between company A and company B (in the same directory level):* contract.txt
- *Enter the industry of the service described in this contract (for example, finance, construction, or others):* landscaping

The following text depicts *example output* that we received from running this multi-agent workflow.

Output:

-> Step 'Budget Advisor' has been completed with the following outcome.

Company A's budget for the period shows a total variance of USD -12,700. The largest variances are in Employee Salaries (USD -5000), Online Advertising (USD -3000), Print Advertising (USD -2000), and Maintenance and Repairs (USD -1000). There are also smaller variances in Rent, Electricity, Water, Landscaping, and Janitorial Services. -> Step 'Contract Synthesizer' has been completed with the following outcome.

The contract between Company A and Company B is for landscaping services at the client's property in Delaware. The total payment to be made by Company A is USD 5,500 upon completion of the work. Both parties have agreed to comply with applicable laws and regulations in Delaware.

-> Step 'Web Search' has been completed with the following outcome.

Subject: Negotiation Proposal for Landscaping Services

Dear Company B Team,

I hope this message finds you well.

After careful review of our internal budget data and market trends in the landscaping industry, we have identified areas where we believe adjustments can be made to better align with our financial constraints while still maintaining high-quality service standards.

1. **Scope of Work**: We propose a reduction in the scope of work, focusing on essential services that directly impact curb appeal and property value. This may include trimming trees, pruning shrubs, and maintaining lawns, with occasional colorful plantings to enhance visual interest.

2. **Payment Terms**: In light of current market trends indicating a slight decrease in landscaping costs due to increased competition, we kindly request a reconsideration of the total payment amount. We propose a revised total payment of USD 4,800 upon completion of the work, reflecting a 12% reduction.

3. **Timeline**: To optimize resource allocation and minimize disruption to our operations, we suggest extending the project timeline by two weeks. This adjustment will allow us to better manage our internal budget constraints without compromising service quality.

We believe these adjustments will enable both parties to achieve a mutually beneficial outcome while adhering to applicable laws and regulations in Delaware. We appreciate your understanding and are open to further discussions to reach an agreement that aligns with the current market trends and our internal budget constraints.

Thank you for your attention to this matter. Please let us know if these proposed adjustments are acceptable, or if you have any counterproposals.

Best regards,

[Your Name]

Company A

-> Step 'Procurement Advisor' has been completed with the following outcome.

The final answer has been sent to Company B, proposing a revised total payment of USD 4,800 upon completion of the work, reflecting a 12% reduction. The proposal also includes a reduced scope of work and an extended project timeline.

Final email: The final answer has been sent to Company B, proposing a revised total payment of USD 4,800 upon completion of the work, reflecting a 12% reduction. The proposal also includes a reduced scope of work and an extended project timeline.

Evidently, the agents correctly invoked their available tools to read and synthesize the contract and budget data to then formulate an effective email in which the terms of the contract are negotiated in favor of the client. We can see the output of each agent within the workflow and the importance of each agent's role. Key details such as the scope of landscaping work, payment terms and the contract timeline are highlighted in the email. We can also see that the negotiation utilizes market trends in landscaping to the client's benefit as well. Finally, the revised total payment of USD 4,800 proposed in the email falls within the client's landscaping budget of USD 5,200. This looks great!

Summary

With this tutorial, you built several BeeAI agents, each with custom tools. Each agent played a critical role in the contract management system use case. Some next steps can include exploring the various GitHub repositories available in the i-am-bee GitHub organization and building more custom tools. In the repositories, you will also find starter Python notebooks to better understand the core components of BeeAI such as PromptTemplates , Messages , Memory and Emitter , for observability.