Getting started with Fetch.ai x CrewAI

Fetch.ai provides a dynamic communication layer that allows you to modularize components into distinct Agents . In this ecosystem, Agents act as microservices, programmed to communicate either with other agents or with humans. While other frameworks also create agents, here we'll explore how Fetch.ai's unique agent architecture can complement and extend CrewAI's functionality.

By using Fetch.ai's Agents to represent various elements of your Crew, your project can be designed to interact with third parties to enhance both flexibility and scalability and increase potential economic benefits.

In this guide, we'll walk you through a simple CrewAI example to understand its fundamentals and then demonstrate how we can enhance this model by integrating Fetch.ai Agents for advanced communication and workflow management.

A simple CrewAI example

Let's use a shortened example that CrewAI provides; in this example we define an Agent and a Task. At a high level here, Agent is defining the LLMs "personality" and the Task is what that LLM is to solve.

Self hosted crew.py fimport os from crewai import Agent , Task , Crew , Process from crewai_tools import SerperDevTool

os . environ ["OPENAI_API_KEY"]

=

"YOUR_API_KEY" os . environ ["SERPER_API_KEY"]

=

"Your Key"

serper.dev API key

search_tool

SerperDevTool ()

Define your agents with roles and goals

researcher

Agent (role = 'Senior Research Analyst' , goal = 'Uncover cutting-edge developments in AI and data science' , backstory = """You work at a leading tech think tank. Your expertise lies in identifying emerging trends. You have a knack for dissecting complex data and presenting actionable insights.""" , verbose = True , allow_delegation = False , tools = [search_tool])

Create tasks for your agents

task1

Task (description = """Conduct a comprehensive analysis of the latest advancements in AI in 2024. Identify key trends, breakthrough technologies, and potential industry impacts.""" , expected_output = "Full analysis report in bullet points" , agent = researcher)

Instantiate your crew with a sequential process

crew

Crew (agents = [researcher,], tasks = [task1,], verbose = True , process = Process.sequential)

Get your crew to work!

result

crew . kickoff ()

print ("######################") print (result) This is a nice feature; we love the idea of having many tasks and then orchestrating to agents which can execute them. But what if we want to create a workflow, perhaps one where information is passed back and forwards?

Next, let us show you two Agents, which communicate, just like that.

A simple communication with agents

Fetch.ai has the concept of an agent where this is an agent that is the component that links agents together.

You can read more about agents communication in ourguides

Let's install what we need:

poetry

init poetry

add

uagents Check out more detailed instructions forinstallation ofuagents library on your end.

First Agent

Our first agent is simple; it sends a message every two seconds to a static address. When this agent receives a message, it prints that to log:

Self hosted sender_agent.py from uagents import Agent , Context , Model from uagents . setup import fund_agent_if_low

class

Message (Model): message :

str

RECIPIENT_ADDRESS

"agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp"

agent

Agent (name = "agent" , port = 8000 , seed = "" , endpoint = ["http://127.0.0.1:8000/submit"],)

fund_agent_if_low (agent.wallet. address ())

@agent . on_interval (period = 2.0) async

def

send_message (ctx : Context): await ctx . send (RECIPIENT_ADDRESS, Message (message = "hello there"))

@agent . on_message (model = Message) async

def

message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

if

name

==

"**main**" : agent . run () This first agent introduces a few core concepts you will need to be aware of when creating any agent.

Agents are defined with theAgent class:

Self hosted sender_agent.py agent =

Agent (name = "agent" , port = 8000 , seed = "" , endpoint = ["http://127.0.0.1:8000/submit"],) Aseed is a unique phrase whichuagents library uses to create a unique private key pair for your agent. If you change yourseed you may lose access to previous messages, and also, the agent's address registered to the Almanac will change subsequently. Theport allows us to define a local port for messages to be received. Theendpoint defines the path to the in-built Rest API. Thename defines the name of the agent.

There are more options for theAgent class; seeAgent Class for further reference.

We then need to define ourcommunication model :

Self hosted sender_agent.py class

Message (Model): message :

str TheModel defines the object sent from agent to agent and represents the type of messages the agent is able to handle. For explicit communication, both agents must handle the sameModel class.Model is the base class that inherits from Pydantic BaseModel.

With thefund_agent_if_low(agent.wallet.address()) function, agents will ultimately pay for discoverability as the economy of agents matures. There is a placeholder for registration here.

Finally, agents have two decorated functions.

The first one is theagent.on_interval() function. This one sends a message every 2 seconds.ctx.send() has the args ofdestination_address andMessage which we defined earlier.

Self hosted sender_agent.py @agent . on_interval (period = 2.0) async

def

send_message (ctx : Context): await ctx . send (RECIPIENT_ADDRESS, Message (message = "hello there")) The second one isagent.on_message() which is a little different; when the agent receives a message at theendpoint we defined earlier, theuagent library unpacks the message and triggers any function which handles that message; in our case, theagent.on_message() function:

Self hosted sender_agent.py @agent . on_message (model = Message) async

def

message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

Second Agent

Agent two doesn't do anything different to agent one; it has different args for the Agent instantiation, and instead of sending a messageon_event("startup") , agent two just logs its address to screen. Whenever agent two receives a message matchingMessage data model, it will send a response to the sender.

Self hosted receiver_agent.py from uagents . setup import fund_agent_if_low from uagents import Agent , Context , Model

class

Message (Model): message :

str

agent

Agent (name = "agent 2" , port = 8001 , seed = "" , endpoint = ["http://127.0.0.1:8001/submit"],)

fund_agent_if_low (agent.wallet. address ())

@agent . on_event ("startup") async

def

start (ctx : Context): ctx . logger . info (f "agent address is { agent.address } ")

@agent . on_message (model = Message) async

def

message_handler (ctx : Context ,

sender :

str ,

msg : Message): ctx . logger . info (f "Received message from { sender } : { msg.message } ")

await ctx . send (sender, Message (message = "hello there"))

if

name

==

"**main**" : agent . run () Okay, let's now run these agents.

Running the agents

Let's run the second agent's script first using this command:poetry run python receiver_agent.py

We must run the second agent first to get its unique address . This is shown in output in the log. Let's updatesender_agent.py script by filling theRECIPIENT_ADDRESS field with the address of the second agent from of the output we previously got by runningreceiver_agent.py script.

Updatedsender_agent.py script sample:

agent1.py from uagents import Agent , Context , Model from uagents . setup import fund_agent_if_low

class

Message (Model): message :

bool

RECIPIENT_ADDRESS

"agent...."

agent

Agent (... Then, let's run the script for the first agent using this command:poetry run python sender_agent.py

Great! You should now be seeing some log out output with our messages being displayed.

Output

- Sender Agent
- :

- INFO: [agent]: Registering on almanac contract...
- INFO: [agent]: Registering on almanac contract...complete
- INFO: [agent]: Starting server on http://0.0.0.0:8000 (Press CTRL+C to quit)
- INFO: [agent]: Received message from agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp: hello there
- INFO: [agent]: Received message from agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp: hello there
- INFO: [agent]: Received message from agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp: hello there
-
    - Receiver Agent
-
    - :
-
    - INFO: [agent 2]: Registering on almanac contract...
-
    - INFO: [agent 2]: Registering on almanac contract...complete
-
    - INFO: [agent 2]: agent address is agent1qf4au6rzaauxhy2jze6v85rspgvredx9m42p0e0cukz0hv4dh2sqjuhujpp
-
    - INFO: [agent 2]: Starting server on http://0.0.0.0:8001 (Press CTRL+C to quit)

happy to here

Wrapping them together - Building a service

Let's go further now and change our agents scripts by splitting the logic of the CrewAI example above. Let's have one agent that creates a task and another agent which fulfills it.

Agent one: Senior Research Analyst Agent

Thesenior_research_analyst_agent , is designed to handle requests for city-specific information by researching advancements in artificial intelligence (AI) and providing current weather updates. It utilizes theSerperDevTool for web searches and employs models for managing input and output.

- create_task(city: str) -> Task
- : This method formulates a task description that outlines the specific research and weather update needed for the given city, returning a Task object.
- run_process(city: str)
- : This method coordinates the execution of the created task. It initializes a Crew process that includes the research agent and the task, allowing it to gather results efficiently.
- handle_city_request(ctx: Context, sender: str, msg: CityRequestModel)
- : This function listens for incoming messages containing city names. Upon receiving a request, it logs the city name, initiates the research process, and sends back a report that includes both AI advancements and weather data.

When a request is received with a city name, the agent logs the request, conducts a comprehensive analysis, and returns a report that combines AI insights with real-time weather data. This functionality makes it a valuable tool for research organizations and tech consultancies seeking relevant information.

Self hosted crewai_agent_1.py from uagents import Agent , Context , Model import os from crewai import Agent as CrewAIAgent , Task , Crew , Process from crewai_tools import SerperDevTool

senior_research_analyst_agent

Agent (name = "senior_research_analyst_agent" , seed = "senior_research_analyst_agent_seed" , port = 8001 , endpoint = ["http://127.0.0.1:8001/submit"],)

class

CityRequestModel (Model): city :

str

class

ResearchReportModel (Model): report :

str

os . environ ["OPENAI_API_KEY"]

=

"" os . environ ["SERPER_API_KEY"]

=

""

class

SeniorResearchAnalyst : def

init (self): """ Initializes the Senior Research Analyst agent with a search tool. """ self . search_tool =

SerperDevTool ()

self . researcher =

CrewAIAgent (role = "Senior Research Analyst" , goal = "Uncover cutting-edge developments in AI and provide weather updates." , backstory = """You work at a leading tech think tank. Your expertise lies in identifying emerging trends and understanding external factors like weather.""" , verbose = True , allow_delegation = False , tools = [self.search_tool],)

def

create_task (self ,

city :

str) -> Task: """ Creates a task for conducting research on AI advancements and retrieving weather updates.

Parameters: - city: str, the city for which the weather update is requested.

Returns: - Task: The created task with the specified description and expected output. """ task_description = (f "Conduct a comprehensive analysis of the latest advancements in AI in 2024. " f "Also, use the search tool to provide the current weather update for { city } .")

return

Task (description = task_description, expected_output = "Full analysis report with weather data" , agent = self.researcher,)

def

run_process (self ,

city :

str): """ Runs the process for the created task and retrieves the result.

Parameters: - city: str, the city for which the task is run.

Returns: - result: The output from the CrewAI process after executing the task. """ task = self . create_task (city) crew =

Crew (agents = [self.researcher], tasks = [task], verbose = True , process = Process.sequential,) result = crew . kickoff () return result

@senior_research_analyst_agent . on_message (model = CityRequestModel, replies = ResearchReportModel) async

def

handle_city_request (ctx : Context ,

sender :

str ,

msg : CityRequestModel): """ Handles incoming messages requesting city information.

What it does: - Logs the received city name. - Runs the research process for the specified city and sends the report back to the sender.

Parameters: - ctx: Context, provides the execution context for logging and messaging. - sender: str, the address of the sender agent. - msg: CityRequestModel, the received message containing the city name.

Returns: - None: Sends the research report to the sender agent. """ ctx . logger . info (f "Received message from { sender } with city: { msg.city } ") research_analyst =

SeniorResearchAnalyst () gather_task_result = research_analyst . run_process (msg.city) await ctx . send (sender, ResearchReportModel (report = str (gather_task_result)))

if

name

==

"**main**" : """ Starts the communication agent and begins listening for messages.

What it does: - Runs the agent, enabling it to send/receive messages and handle events.

Returns: - None: Runs the agent loop indefinitely. """ senior_research_analyst_agent . run ()

Agent two: Research Asking Agent

Theresearch_asking_agent agent is designed to initiate requests for city-specific research information and manage responses related to research reports.

- Models:CityRequestModel
- for storing the city name andResearchReportModel
- for the research findings.
- Startup Event: Theon_startup
- function logs the agent's name and address, then sends a message to a target agent with the default city (London).
- Message Handling: Thehandle_research_report
- function processes incoming reports, logging the sender's address and the content of the research report.

This agent operates continuously, enabling effective communication and retrieval of city-related research insights.

Self hosted crewai_agent_2.py from uagents import Agent , Context , Model

class

CityRequestModel (Model): city :

str

class

ResearchReportModel (Model): report :

str

research_asking_agent

Agent (name = "research_asking_agent" , seed = "research_asking_agent_seed" , port = 8000 , endpoint = ["http://127.0.0.1:8000/submit"],)

TARGET_AGENT_ADDRESS

("agent1qgxfhzy78m2qfdsg726gtj4vnd0hkqx96xwprng2e4rmn0xfq7p35u6dz8q") DEFAULT_CITY =

"London"

@research_asking_agent . on_event ("startup") async

def

on_startup (ctx : Context): """ Triggered when the agent starts up.

What it does: - Logs the agent's name and address. - Sends a message to the target agent with the default city (e.g., 'London').

Parameters: - ctx: Context, provides the execution context for logging and messaging.

Returns: - None: Sends the message to the target agent asynchronously. """ ctx . logger . info (f "Hello, I'm { research_asking_agent.name } , and my address is { research_asking_agent.address } .")

await ctx . send (TARGET_AGENT_ADDRESS, CityRequestModel (city = DEFAULT_CITY))

@research_asking_agent . on_message (model = ResearchReportModel) async

def

handle_research_report (ctx : Context ,

sender :

str ,

msg : ResearchReportModel): """ Triggered when a message of type ResearchReportModel is received.

What it does: - Logs the sender's address and the research report received.

Parameters: - ctx: Context, provides the execution context for logging and messaging. - sender: str, the address of the sender agent. - msg: ResearchReportModel, the received research report.

Returns: - None: Processes the message and logs it. """ ctx . logger . info (f "Received research report from { sender } : { msg.report } ")

if

name

==

"**main**" : """ Starts the research analyst agent and begins listening for events.

What it does: - Runs the agent, enabling it to send/receive messages and handle events.

Returns: - None: Runs the agent loop indefinitely. """ research_asking_agent . run ()

Output

Runpoetry run python crewai_agent_1.py first and thenpoetry run python crewai_agent_2.py .

You should get something similar to the following for each agent:

- Agent 2
- :
- INFO: [research_asking_agent]: Registration on Almanac API successful
- INFO: [research_asking_agent]: Almanac contract registration is up to date!
- INFO: [research_asking_agent]: Hello, I'm research_asking_agent, and my address is agent1qvfw094hmsfrvmlw8amg4ypyvd2wm8hvahxt0zwmareus6fzajpm5dpgh3f.
- INFO: [research_asking_agent]: Starting server on http://0.0.0.0:8000 (Press CTRL+C to quit)
- INFO: [research_asking_agent]: Received research report from agent1qgxfhzy78m2qfdsg726gtj4vnd0hkqx96xwprng2e4rmn0xfq7p35u6dz8q: **Comprehensive Analysis of the Latest Advancements in AI in 2024**
- The year 2024 has emerged as a pivotal period for artificial intelligence, with several groundbreaking trends and developments. After analyzing multiple sources, here are the key advancements in AI for this year:
- 1. **Multimodal AI**:
- Multimodal AI involves the integration of multiple forms of data including text, audio, and visual inputs to create more sophisticated and context-aware AI models. This development allows for more accurate and comprehensive AI systems capable of better mimicking human understanding.
- 1. **Small Language Models**:
- As opposed to massive language models like GPT-3, smaller language models are becoming more prevalent. These models are more efficient, faster, and require fewer resources while maintaining a high level of performance.
- 1. **Customizable Generative AI**:
- The rise of generative AI that can be fine-tuned for specific tasks has been significant. Such AI systems can be adapted to meet the unique needs of different users and industries, providing more personalized and effective solutions.

- 1. **Agentic AI**:
- These are AI systems designed to act autonomously to achieve specific goals, showing more complex behavior and decision-making capabilities. This advancement heralds a future where AI can conduct complex tasks with minimal human intervention.
- 1. **Open Source AI**:
- There is a growing trend towards open-sourcing AI models which facilitates collaboration and innovation among researchers and developers worldwide. This movement is democratizing access to advanced AI technologies.
- 1. **Retrieval-Augmented Generation**:
- This involves AI systems that can search for and retrieve external information to enhance their responses. This technology is particularly useful in creating more accurate and up-to-date AI systems.
- 1. **Generative AI and Copyright Challenges**:
- The rise of generative AI has brought about new discussions around copyright and usage laws. As AI becomes more creative, legal frameworks must adapt to address ownership and licensing issues.
- 1. **Licensing Innovations**:
- To keep up with the rapid development in AI, new licensing models are being explored. These models aim to balance innovation with fair use and protection of intellectual property.
- 1. **AI Patent Developments**:
- Innovations in AI are leading to an increase in AI-related patents. This increase highlights the competitive landscape and the importance of intellectual property in the field of AI.
- 1. **Healthcare Applications**:
- AI's role in healthcare continues to grow, particularly in diagnostics, personalized medicine, and patient care. AI technologies are aiding in early detection of diseases and creating more effective treatment plans.
- These developments collectively signify a year of significant progress in AI, promising transformative impacts across various industries.
- **Current Weather Update for London**
- As of the latest reports, the current weather in London, England, United Kingdom is as follows:
- Morning: 65°F, partly cloudy with an 11% chance of rain.
- Afternoon: 71°F, sunny with 0% chance of rain.
- Evening: 62°F, partly cloudy with a 6% chance of rain.
- Overnight: 60°F, clear skies.
- Overall, London is experiencing mild weather with clear skies and minimal chances of rain throughout the day.
- This comprehensive analysis and weather update should provide valuable insights into the latest advancements in AI and the current weather conditions in London.`

Last updated on October 17, 2024

Was this page helpful?

You can also leave detailed feedback[on Github]

[Startup Idea analyzer] [Getting started with Fetch.ai x Langchain]

On This Page