Autonomous Intelligence

Autonomous Multi-Agent AI Systems

Presentation Overview

We will explore how intelligent autonomous multi-agent systems can revolutionize workflows. By leveraging collaborative multi-agent AI systems, people can automate routine tasks and streamline complex processes. We will:

- Go over the architecture of building multi-agent systems,
- Talk about how to coordinate teams of AI agents that work together,
- Discuss how to monitor and optimize these systems to be intelligent,
- Show real-world applications that highlight their potential to enhance efficiency.

Terminology / Definitions

What is an agent?

An agent is an **autonomous unit** programmed to:

- Perform tasks
- Make decisions
- Communicate with other agents

Overview of a Task

In the CrewAI framework, a `Task` is a specific assignment completed by an `Agent`.

They provide all necessary details for execution, such as a description, the agent responsible, required tools, and more, facilitating a wide range of action complexities.

Tasks within CrewAI can be collaborative, requiring multiple agents to work together. This is managed through the task properties and orchestrated by the Crew's process, enhancing teamwork and efficiency.

What is a Crew?

A crew in crewAI represents a collaborative group of agents working together to achieve a set of tasks. Each crew defines the strategy for task execution, agent collaboration, and the overall workflow.

Process Implementations

- **Sequential**: Executes tasks sequentially, ensuring tasks are completed in an orderly progression.
- **Hierarchical**: Organizes tasks in a managerial hierarchy, where tasks are delegated and executed based on a structured chain of command. A manager language model (`manager_llm`) or a custom manager agent (`manager_agent`) must be specified in the crew to enable the hierarchical process, facilitating the creation and management of tasks by the manager.
- **Consensual Process (Planned)**: Aiming for collaborative decision-making among agents on task execution, this process type introduces a democratic approach to task management within CrewAI. It is planned for future development and is not currently implemented in the codebase.

Large Language Models (LLMs) in CrewAI

Large Language Models (LLMs) are the backbone of intelligent agents in the CrewAI framework. This guide will help you understand, configure, and optimize LLM usage for your CrewAI projects.

What is a Tool?

A tool in CrewAI is a skill or function that agents can utilize to perform various actions. This includes tools from the **CrewAI Toolkit** and **LangChain Tools**, enabling everything from simple searches to complex interactions and effective teamwork among agents.

Panacea

Download Private Version

Chat 1487

Chat 1468

Text Data Solutions Pla

Chat 373

Chat 372

AI Startup Panacea Sol

Chat 153

Chat 152

Healthcare Summariza

Chat 19

Answer Assistance Ser

Text Data Solutions Platform

Hello, I am your Panacea, your agentic AI assistant. What can I do to help?

what is anote's solution?

Anote's solution consists of three fully functional products and three main steps: 1) Label Text Data - This entails classifying text, extracting entities, and answering questions on documents. 2) Fine Tune Model - Allows you to run your Fine Tuned LLM locally with their model inference API. 3) Private Chatbot - Enables you to chat with your documents while keeping your data private and secure. The core technology is a fine-tuning library that leverages state-of-the-art few-shot learning to make high-quality predictions with a few labeled samples. The fine-tuning can be done in three ways: 1) Unsupervised Fine Tuning - Fine-tuning your LLM from your raw

Ask your document a question

AGENT SELECTION

Select...

MODEL SELECTION

Select...

UPLOADED FILES

PitchDeck_Anote.pdf

PrivateChatbotDeck.pdf

Anote - Solution.pdf

SOURCES

Our Solution

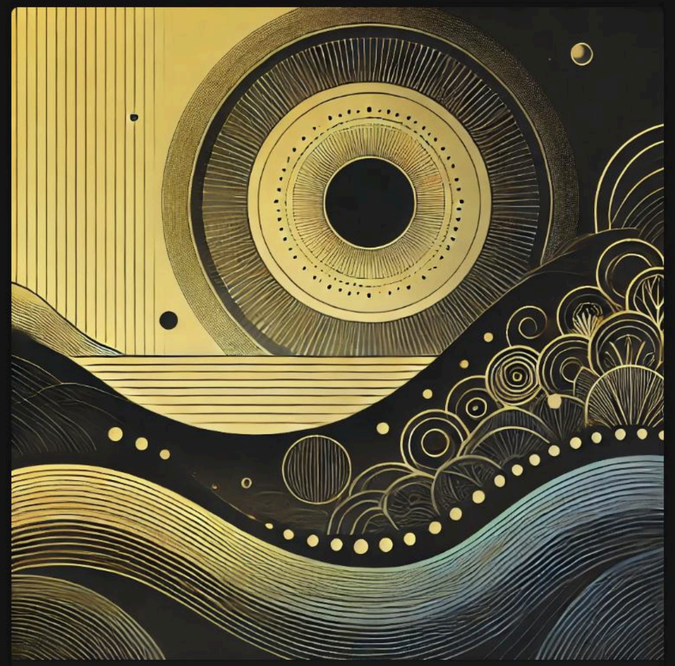
A platform to create, deploy, and monitor autonomous multi-agent systems.

Build Autonomous Agents

Develop agents equipped with memory, reasoning, and tool capabilities. Each agent is designed to operate independently, handling tasks and managing state within a multi-agent environment.

Coordinate Agent Teams

Orchestrate interactions between agents as they collaborate on complex tasks. Panacea's system enables seamless communication, task handoff, and dynamic adjustment within agent teams.



Monitor and Optimize

Track agent performance through detailed metrics and logs. Optimize behavior in real-time, adjusting parameters and workflows to improve efficiency and achieve goals with minimal supervision.

Agent Registry

Upreach

Email Marketing Agent

More Info

Private Chatbot

Financial Analyst Agent

More Info

Anote

Data Labeler Agent

More Info

AI-RFP

Grant Writer Agent

More Info

Autocode

Software Engineer Agent

More Info

Applico

Job Seeker Agent

More Info

Upreach Problem

Traditional outreach is tedious and ineffective

1

TIME CONSUMING

Finding the perfect leads is difficult and takes too long

2

INEFFICIENT

Hard to find a tailored way to reach out at scale

3

REPETITIVE

A daily iterative process that requires too much manual labor

Example Multi Agent AI System - Upreach

A user (i.e. sales development representative) comes to Upreach and requests a specific list of leads and a tailored email to send.

data foundation

the web

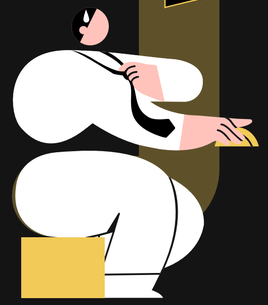
	First Name	Last Name	Role	Company	Verified?	Status
☐	David	Zhu	Software Engineer	Google		
	Abby	Viel	Product Designer	Microsoft		
✓	Johnny	Young	Associate Manager	Facebook		
✓	Audrey	Smith	Business Analyst	McKinsey		
	Christy	Lin	Product Designer	Google		

2 out of 200 results selected Page 1 of 20

Upreach will curate a large list of leads with contacts and a tailored message that will be automatically sent

CUSTOMER INQUIRY

Reach out to a list of 10,000 New York based heads of AI who work in mid-sized finance companies



AI WORKFLOW

The AI workflow processes this command and then creates and follows specific guidelines or rules to sift through the data to find the most useful results based on this input

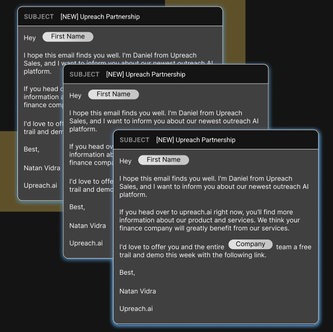
- Have Job Title **"Data Scientist"**
- Work at Location **"California, US"**
- Work in Industry **"Technology"**
- Work at Company Name **"Apple"**
- Have Company Size **">1000"**
- Are Located in **"United States"**

THE DATA

Sourcing our AI-powered data foundation plus the web, the agent sources out the most reliable leads and creates a tailored email

EMAIL AND LIST

AUTOMATIONS



Emails will automatically be sent and shows the progress and amount of emails you send out to leads everyday

AI RFP Problem

Companies want non-dilutive funding, but don't have the time and resources to apply to all these proposal opportunities by hand. The current process is very tedious and manual, but is important to obtain funding.

Number of Grants

Thousands of grants from federal government and globally are available to various sectors including education, healthcare, technology, and the arts.

Customization Effort

High effort needed to align proposals with unique business contexts and niche problems.

150
hours spent

10-50
pages per grant to fill

10-20%
success rate

Grant Submission Requirements

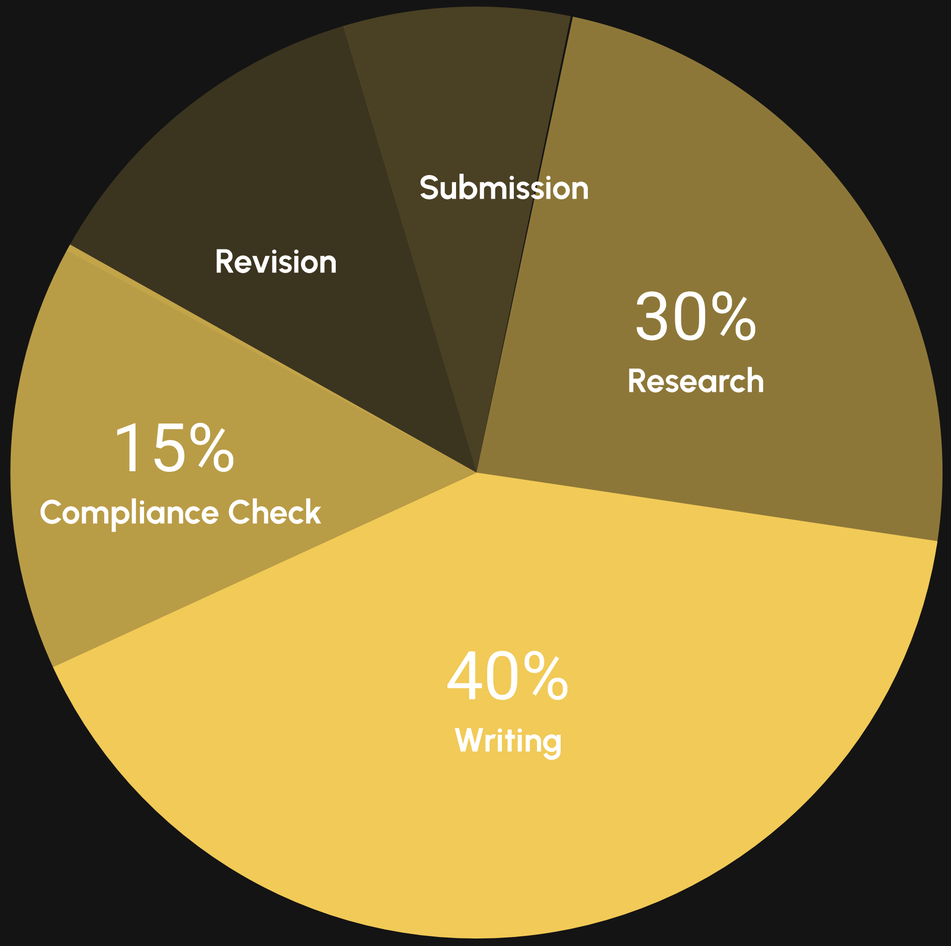
Tailored Applications

Each grant has specific guidelines and objectives.

Business Context

Highlight your business's relevant past experiences and successes.

Distribution of Time Spent on Grants

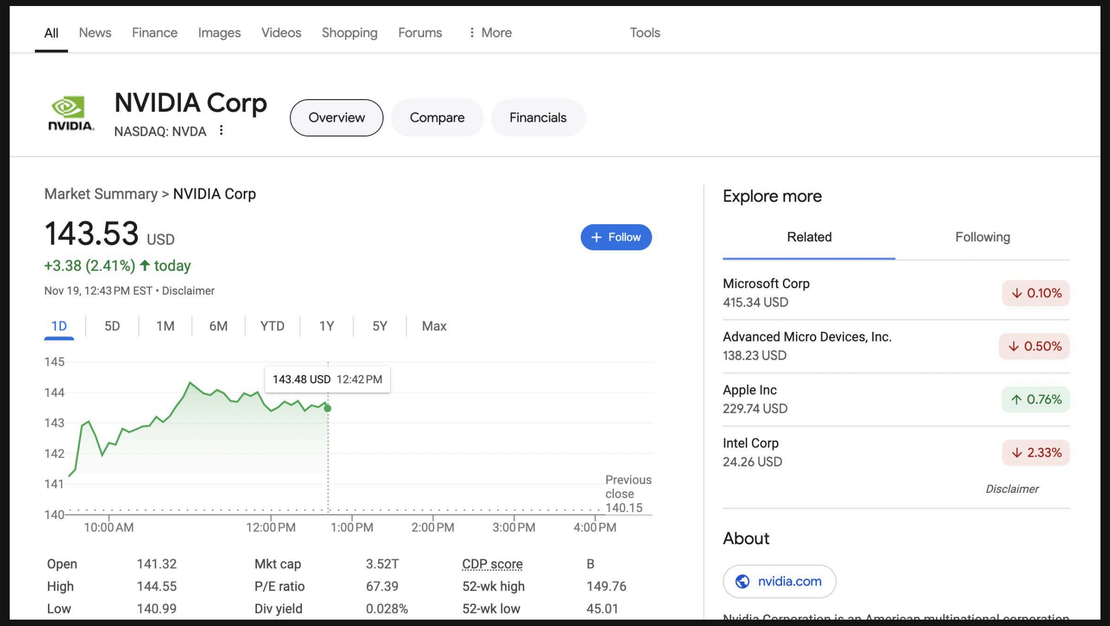


Phidata Financial Agent

Agent 1: Searches the web

Agent 2: Gets financial data

Agent Team: Combines Output of Agent 1 and Agent 2 to summarize analyst recs for NVDA




```python
agent_team.py > ...
from phi.agent import Agent
from phi.model.openai import OpenAIChat
from phi.tools.duckduckgo import DuckDuckGo
from phi.tools.yfinance import YFinanceTools

web_agent = Agent(
    name="Web Agent",
    role="Search the web for information",
    model=OpenAIChat(id="gpt-4o"),
    tools=[DuckDuckGo()],
    instructions=["Always include sources"],
    show_tool_calls=True,
    markdown=True,
)

finance_agent = Agent(
    name="Finance Agent",
    role="Get financial data",
    model=OpenAIChat(id="gpt-4o"),
    tools=[YFinanceTools(stock_price=True, analyst_recommendations=True, company_info=Tru
    instructions=["Use tables to display data"],
    show_tool_calls=True,
    markdown=True,
)

agent_team = Agent(
    team=[web_agent, finance_agent],
    instructions=["Always include sources", "Use tables to display data"],
    show_tool_calls=True,
    markdown=True,
)

agent_team.print_response("Summarize analyst recommendations \
                          and share the latest news for NVDA", stream=True)
```

Crew Event Planning Agent

event_agent_team.py > ...

```python
from crewai import Agent, Crew, Task
from crewai_tools import ScrapeWebsiteTool, SerperDevTool
# Initialize the tools
search_tool = SerperDevTool()
scrape_tool = ScrapeWebsiteTool()

# Agent 1: Venue Coordinator
venue_coordinator = Agent(
    role="Venue Coordinator",
    goal="Identify and book an appropriate venue "
    "based on event requirements",
    tools=[search_tool, scrape_tool],
    verbose=True,
    backstory=(
        "With a keen sense of space and "
        "understanding of event logistics, "
        "you excel at finding and securing "
        "the perfect venue that fits the event's theme, "
        "size, and budget constraints."
    )
)

# Agent 2: Logistics Manager
logistics_manager = Agent(
    role='Logistics Manager',
    goal=(
        "Manage all logistics for the event "
        "including catering and equipmen"
    ),
    tools=[search_tool, scrape_tool],
    verbose=True,
    backstory=(
        "Organized and detail-oriented, "
        "you ensure that every logistical aspect of the event "
        "from catering to equipment setup "
        "is flawlessly executed to create a seamless experience."
    )
)

# Agent 3: Marketing and Communications Agent
marketing_communications_agent = Agent(
    role="Marketing and Communications Agent",
    goal="Effectively market the event and "
         "communicate with participants",
    tools=[search_tool, scrape_tool],
    verbose=True,
    backstory=(
        "Creative and communicative, "
        "you craft compelling messages and "
        "engage with potential attendees "
        "to maximize event exposure and participation."
    )
)
```

event_agent_team.py > ...

```python
from pydantic import BaseModel
# Define a Pydantic model for venue details
# (demonstrating Output as Pydantic)
class VenueDetails(BaseModel):
    name: str
    address: str
    capacity: int
    booking_status: str

venue_task = Task(
    description="Find a venue in {event_city} "
                "that meets criteria for {event_topic}.",
    expected_output="All the details of a specifically chosen"
                    "venue you found to accommodate the event.",
    human_input=True,
    output_json=VenueDetails,
    output_file="venue_details.json",
      # Outputs the venue details as a JSON file
    agent=venue_coordinator
)

logistics_task = Task(
    description="Coordinate catering and "
                "equipment for an event "
                "with {expected_participants} participants "
                "on {tentative_date}.",
    expected_output="Confirmation of all logistics arrangements "
                    "including catering and equipment setup.",
    human_input=True,
    async_execution=True,
    agent=logistics_manager
)

marketing_task = Task(
    description="Promote the {event_topic} "
                "aiming to engage at least"
                "{expected_participants} potential attendees.",
    expected_output="Report on marketing activities "
                    "and attendee engagement formatted as markdown.",
    async_execution=True,
    output_file="marketing_report.md",  # Outputs the report as a text fi
    agent=marketing_communications_agent
)
```

```python
# Define the crew with agents and tasks
event_management_crew = Crew(
    agents=[venue_coordinator,
            logistics_manager,
            marketing_communications_agent],

    tasks=[venue_task,
           logistics_task,
           marketing_task],

    verbose=True
)

event_details = {
    'event_topic': "Tech Innovation Conference",
    'event_description': "A gathering of tech innovators "
                         "and industry leaders "
                         "to explore future technologies.",
    'event_city': "San Francisco",
    'tentative_date': "2024-09-15",
    'expected_participants': 500,
    'budget': 20000,
    'venue_type': "Conference Hall"
}

result = event_management_crew.kickoff(inputs=event_details
```

DSPY (Declarative Self-improving Python)

DSPY is an open-source framework for programming—rather than prompting—language models. It allows you to iterate fast on building modular AI systems and provides algorithms for optimizing their prompts and weights, whether you're building simple classifiers, sophisticated RAG pipelines, or Agent loops. Instead of brittle prompts, you write compositional Python code and use DSPy's tools to teach your LM to deliver high-quality outputs

1) **Modules** express AI behavior in a portable, declarative way.

To build reliable AI systems, you must iterate fast. But maintaining prompts makes that hard: it forces you to tinker with strings or data *every time you change your LM, metrics, or pipeline*. Having built over a dozen best-in-class compound LM systems since 2020, we learned this the hard way—and built DSPy so you don't have to.

DSPy shifts your focus from tinkering with prompt strings to **programming with structured natural-language modules**. For every AI component in your system, you specify input/output behavior as a *signature* and select a *module* to assign a strategy for invoking your LM. DSPy expands your signatures into prompts and parses your typed outputs, so you can write ergonomic, portable, and optimizable AI systems.

Getting Started II: Build DSPy modules for various tasks

Try the examples below after configuring your `lm` above. Adjust the fields to explore what tasks your LM can do well out of the box. Each tab below sets up a DSPy module, like `dspy.Predict`, `dspy.ChainOfThought`, or `dspy.ReAct`, with a task-specific *signature*. For example, `question -> answer: float` tells the module to take a question and to produce a `float` answer.

Math | Retrieval-Augmented Generation | Classification | Information Extraction | Agents

```
math = dspy.ChainOfThought("question -> answer: float")
math(question="Two dice are tossed. What is the probability that the sum equals two?")
```

Possible Output:

```
Prediction(
    reasoning='When two dice are tossed, each die has 6 faces, resulting in a total of 6 x 6 = 36 possible outcomes. The sum of the numbers
    answer=0.0277776
)
```

Using DSPy in practice: from quick scripting to building sophisticated systems.

2) **Optimizers** tune the prompts and weights of your AI modules.

DSPy provides you with the tools to compile high-level code with natural language annotations into the low-level computations, prompts, or weight updates that align your LM with your program's structure and metrics.

Given a few tens or hundreds of representative *inputs* of your task and a *metric* that can measure the quality of your system's outputs, you can use a DSPy optimizer. Different optimizers in DSPy work by **synthesizing good few-shot examples** for every module, like `dspy.BootstrapRS`,[1] **proposing and intelligently exploring better natural-language instructions** for every prompt, like `dspy.MIPROv2`,[2] and **building datasets for your modules and using them to finetune the LM weights** in your system, like `dspy.BootstrapFinetune`.[3]

Getting Started III: Optimizing the LM prompts or weights in DSPy programs

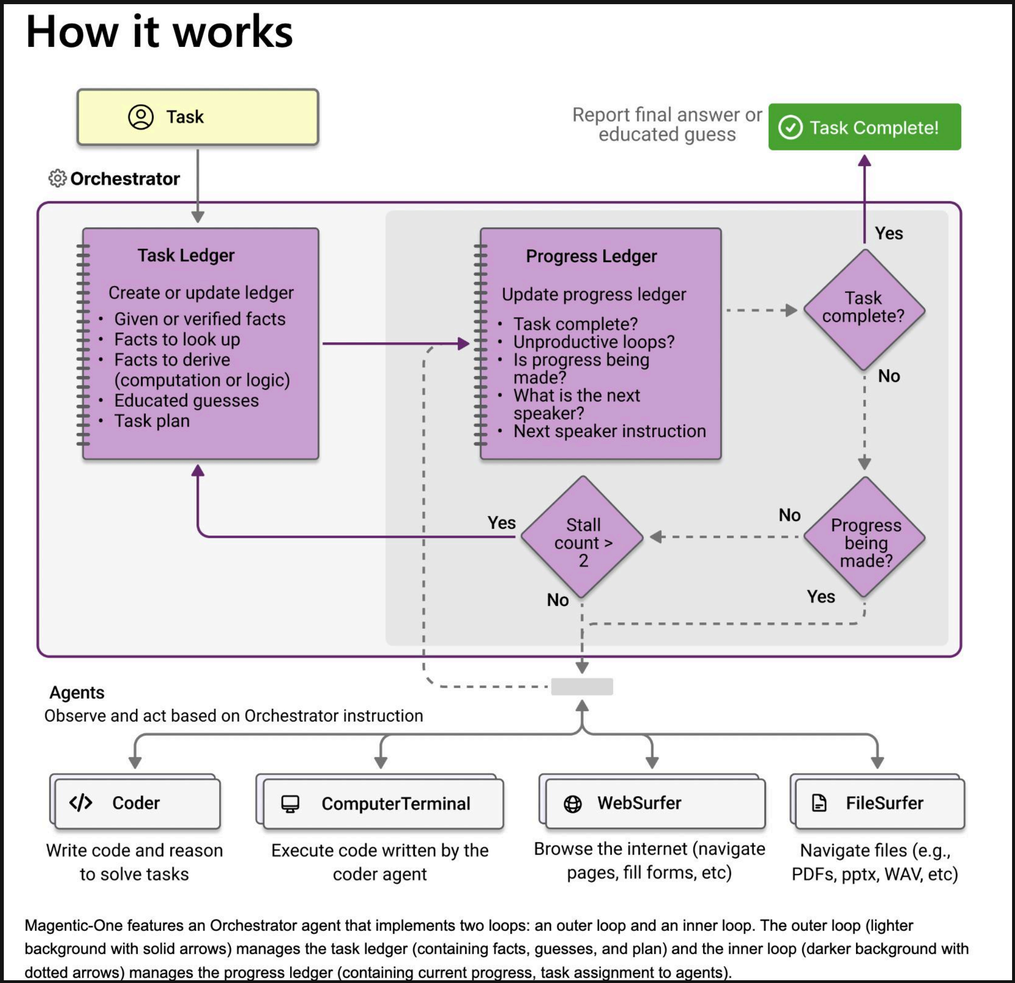
A typical simple optimization run costs on the order of $2 USD and takes around 20 minutes, but be careful when running optimizers with very large LMs or very large datasets. Optimization can cost as little as a few cents or up to tens of dollars, depending on your LM, dataset, and configuration.

Optimizing prompts for a ReAct agent | Optimizing prompts for RAG | Optimizing weights for Classification

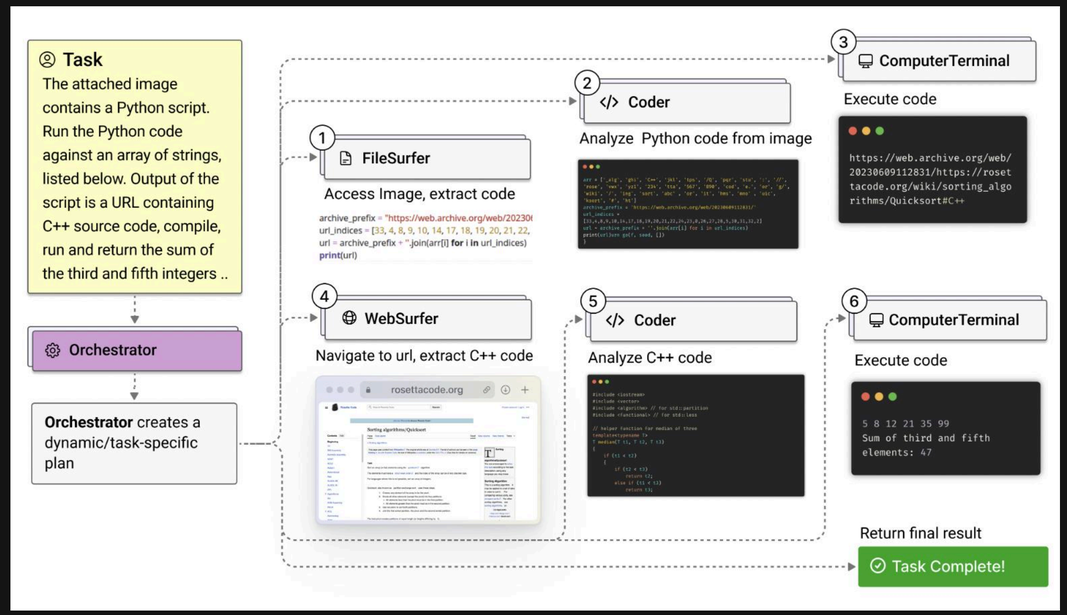
This is a minimal but fully runnable example of setting up a `dspy.ReAct` agent that answers questions via search from Wikipedia and then optimizing it using `dspy.MIPROv2` in the cheap `light` mode on 500 question-answer pairs sampled from the `HotPotQA` dataset.

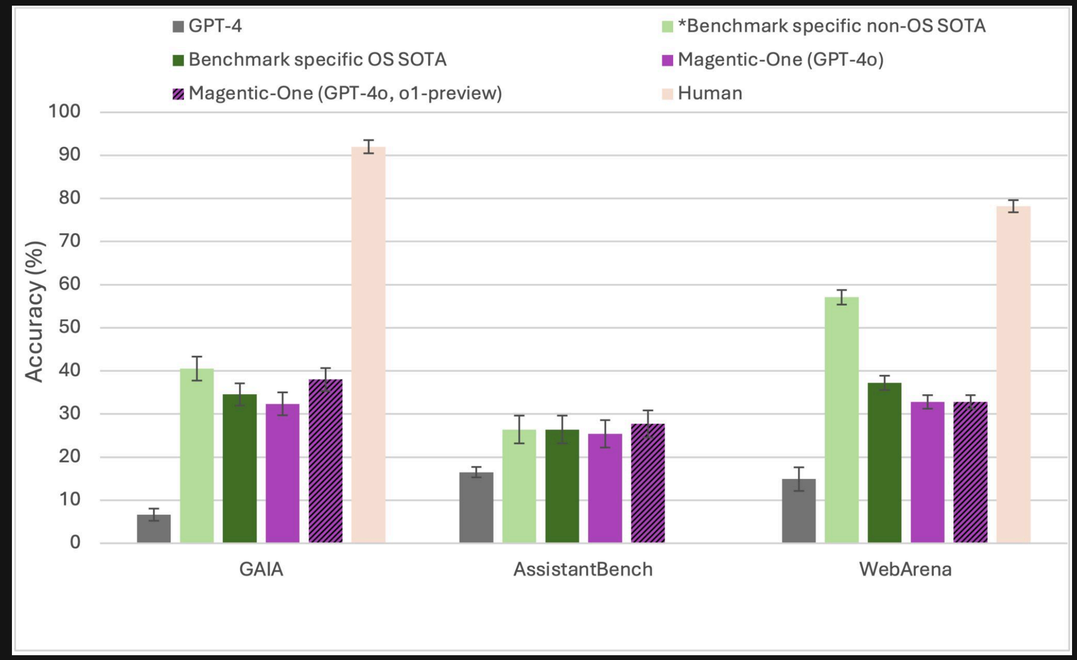
```
import dspy
from dspy.datasets import HotPotQA

dspy.configure(lm=dspy.LM('openai/gpt-4o-mini'))

def search_wikipedia(query: str) -> list[str]:
    results = dspy.ColBERTv2(url='http://20.102.90.50:2017/wiki17_abstracts')(query, k=3)
    return [x['text'] for x in results]

trainset = [x.with_inputs('question') for x in HotPotQA(train_seed=2024, train_size=500).train]
react = dspy.ReAct("question -> answer", tools=[search_wikipedia])

tp = dspy.MIPROv2(metric=dspy.evaluate.answer_exact_match, auto="light", num_threads=24)
optimized_react = tp.compile(react, trainset=trainset)
```

Magenta One Generalist Agent



Magentic-One features an Orchestrator agent that implements two loops: an outer loop and an inner loop. The outer loop (lighter background with solid arrows) manages the task ledger (containing facts, guesses, and plan) and the inner loop (darker background with dotted arrows) manages the progress ledger (containing current progress, task assignment to agents).





Open Ended Questions

- How can you optimize the orchestrator to automatically spawn these agents in the correct order / process, without predefined agents?
- How do you evaluate agentic systems to ensure reliability?
- How do you train agentic models, and what data do you need? Can you fine tune teams agents to do better than zero shot agent-teams?
- Can you run these agents privately, on premise, with models like Llama3?
- How do you effectively monitor these agents via logs to ensure that they are doing the correct tasks?

If interested in learning more and building these agentic AI solutions, contact me at **vidranatan@gmail.com**, or connect with me at **https://www.linkedin.com/in/natanvidra/**