

# Sharif LLM Agent Hackathon

May 28<sup>th</sup>, 2025

## Building LLM-powered multiagent systems with AG2

Alireza Ghafarollahi

Massachusetts Institute of Technology



Cite this: *Digital Discovery*, 2024, 3, 1389

## ProtAgents: protein discovery *via* large language model multi-agent collaborations combining physics and machine learning†

Alireza Ghafarollahi<sup>a</sup> and Markus J. Buehler<sup>id</sup> \*<sup>ab</sup>

Designing *de novo* proteins beyond those found in nature holds significant promise for advancements in both scientific and engineering applications. Current methodologies for protein design often rely on AI-based models, such as surrogate models that address end-to-end problems by linking protein structure



## Automating alloy design and discovery with physics-aware multimodal multiagent AI

Alireza Ghafarollahi<sup>a</sup> and Markus J. Buehler<sup>a,b,1</sup> <sup>id</sup>

Edited by Yonggang Huang, Northwestern University–Evanston, Glencoe, IL; received July 13, 2024; accepted November 13, 2024

The design of new alloys is a multiscale problem that requires a holistic approach that involves retrieving relevant knowledge, applying advanced computational methods, conducting experimental validations, and analyzing the results, a process that is typically slow and reserved for human experts. Machine learning can help accelerate this process, for instance, through the use of deep surrogate models that connect structural and chemical features to material properties, or vice versa. However, existing data-

### Significance

We construct a physics-aware AI model that integrates the advanced reasoning, rational

## SciAgents: Automating Scientific Discovery Through Bioinspired Multi-Agent Intelligent Graph Reasoning

Alireza Ghafarollahi and Markus J. Buehler\*

A key challenge in artificial intelligence (AI) is the creation of systems capable of autonomously advancing scientific understanding by exploring novel domains, identifying complex patterns, and uncovering previously unseen

### 1. Introduction

One of the grand challenges in the evolving landscape of scientific discovery is finding

## SPARKS: MULTI-AGENT ARTIFICIAL INTELLIGENCE MODEL DISCOVERS PROTEIN DESIGN PRINCIPLES \*

Alireza Ghafarollahi  
Laboratory for Atomistic and Molecular Mechanics (LAMM)  
Massachusetts Institute of Technology  
77 Massachusetts Ave.  
Cambridge, MA 02139, USA

Markus J. Buehler  
Laboratory for Atomistic and Molecular Mechanics (LAMM)  
Center for Computational Science and Engineering  
Schwarzman College of Computing  
Massachusetts Institute of Technology  
77 Massachusetts Ave.  
Cambridge, MA 02139, USA

Correspondence: mbuehler@MIT.EDU

# Overview

---

Build your first multi-agent system

A pre-programmed multi-agent system with Python

Fully automated multi-agent system (AG2)

- Basic Concepts
  - LLM Configuration
  - ConversableAgent
  - Human in the Loop
  - Agent Orchestration
  - Tools
- Agent Orchestration Deep Dive
  - Two Agent Chat
  - Sequential Chat
  - Group Chats
    - Tools and functions
    - Context Variables
    - Handoffs
    - Orchestration Patterns (if time allowed)

Build your first multi-agent system with ~~AG2~~ Python

# A pre-programmed multi-agent system with Python

- Pre-programmed sequence of interactions between agents
- More consistency and reliability
- Less flexibility
- No human in the loop

2024-9-4

## The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery

Chris Lu<sup>1,2,\*</sup>, Cong Lu<sup>3,4,\*</sup>, Robert Tjarko Lange<sup>1,\*</sup>, Jakob Foerster<sup>2,†</sup>, Jeff Clune<sup>3,4,5,†</sup> and David Ha<sup>1,†</sup>

\*Equal Contribution, <sup>1</sup>Sakana AI, <sup>2</sup>FLAIR, University of Oxford, <sup>3</sup>University of British Columbia, <sup>4</sup>Vector Institute, <sup>5</sup>Canada CIFAR AI Chair, <sup>†</sup>Equal Advising

SCIAGENTS: AUTOMATING SCIENTIFIC DISCOVERY THROUGH  
MULTI-AGENT INTELLIGENT GRAPH REASONING \*

SPARKS: MULTI-AGENT ARTIFICIAL INTELLIGENCE MODEL  
DISCOVERS PROTEIN DESIGN PRINCIPLES \*

**Alireza Ghafarollahi**  
Laboratory for Atomistic and Molecular Mechanics (LAMM)  
Massachusetts Institute of Technology  
77 Massachusetts Ave.  
Cambridge, MA 02139, USA

**Markus J. Buehler**  
Laboratory for Atomistic and Molecular Mechanics (LAMM)  
Center for Computational Science and Engineering  
Schwarzman College of Computing  
Massachusetts Institute of Technology  
77 Massachusetts Ave.  
Cambridge, MA 02139, USA

Correspondence: mbuehler@MIT.EDU

**Alireza Ghafarollahi**  
Laboratory for Atomistic and Molecular Mechanics (LAMM)  
Massachusetts Institute of Technology  
77 Massachusetts Ave.  
Cambridge, MA 02139, USA

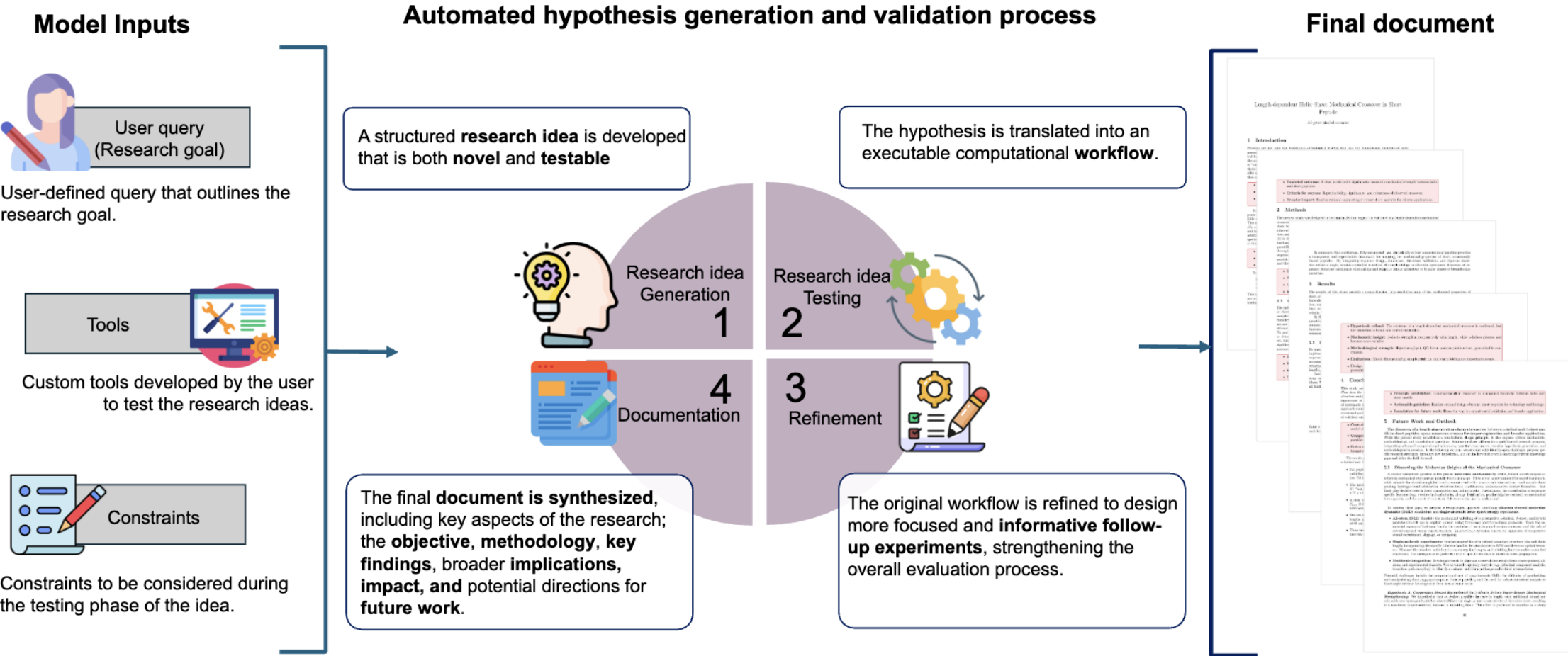
**Markus J. Buehler**  
Laboratory for Atomistic and Molecular Mechanics (LAMM)  
Center for Computational Science and Engineering  
Schwarzman College of Computing  
Massachusetts Institute of Technology  
77 Massachusetts Ave.  
Cambridge, MA 02139, USA

Correspondence: mbuehler@MIT.EDU

AIJ 9 Sep 2024

AIJ 26 Apr 2025

# Sparks: Automated Scientific Discovery



# Building your first agent

---

Agents use LLMs as key components to understand and react

```
client = OpenAI()
response = client.chat.completions.create(
    model,
    messages,
    temperature,
    max_completion_tokens,
)

content = response.choices[0].message.content
```

## OpenAI models

The screenshot displays the OpenAI models interface. At the top, three large model cards are featured: GPT-4.1 (blue background), o4-mini (orange background), and o3 (yellow background). Below these, a section titled 'Reasoning models' lists the o-series models. The models are arranged in two columns. The left column includes o4-mini, o3-mini, and o1-mini. The right column includes o3, o1, and o1-pro. Each model card includes a small icon, the model name, and a brief description.

Model	Description
GPT-4.1	Flagship GPT model for complex tasks
o4-mini	Faster, more affordable reasoning model
o3	Our most powerful reasoning model
o4-mini	Faster, more affordable reasoning model
o3-mini	A small model alternative to o3
o1	Previous full o-series reasoning model
o1-mini	A small model alternative to o1
o1-pro	Version of o1 with more compute for better responses

# Pricing

---

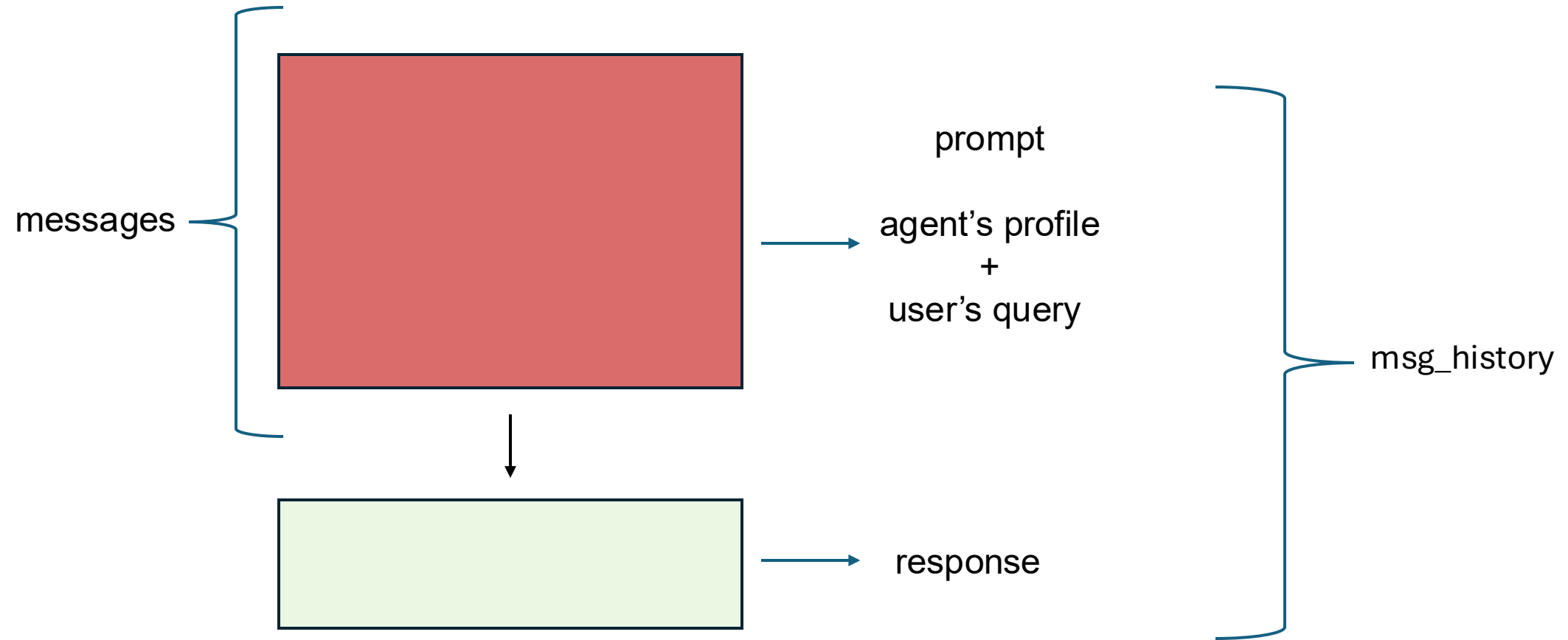
Model	Input Price (per 1M tokens)	Output Price (per 1M tokens)
gpt-4.1	2.0	8.00
gpt-4.1-mini	0.4	1.6
gpt-4o	2.5	10.00
gpt-4o-mini	0.15	0.6
o1	15.00	60.00
o1-pro	150.00	600.00
o3	10.00	40.00
o3-mini	1.10	4.40
o4-mini	1.10	4.40



## Building your first agent

---

```
messages=[  
  {"role": "user", "content": "prompt_1"}  
]
```

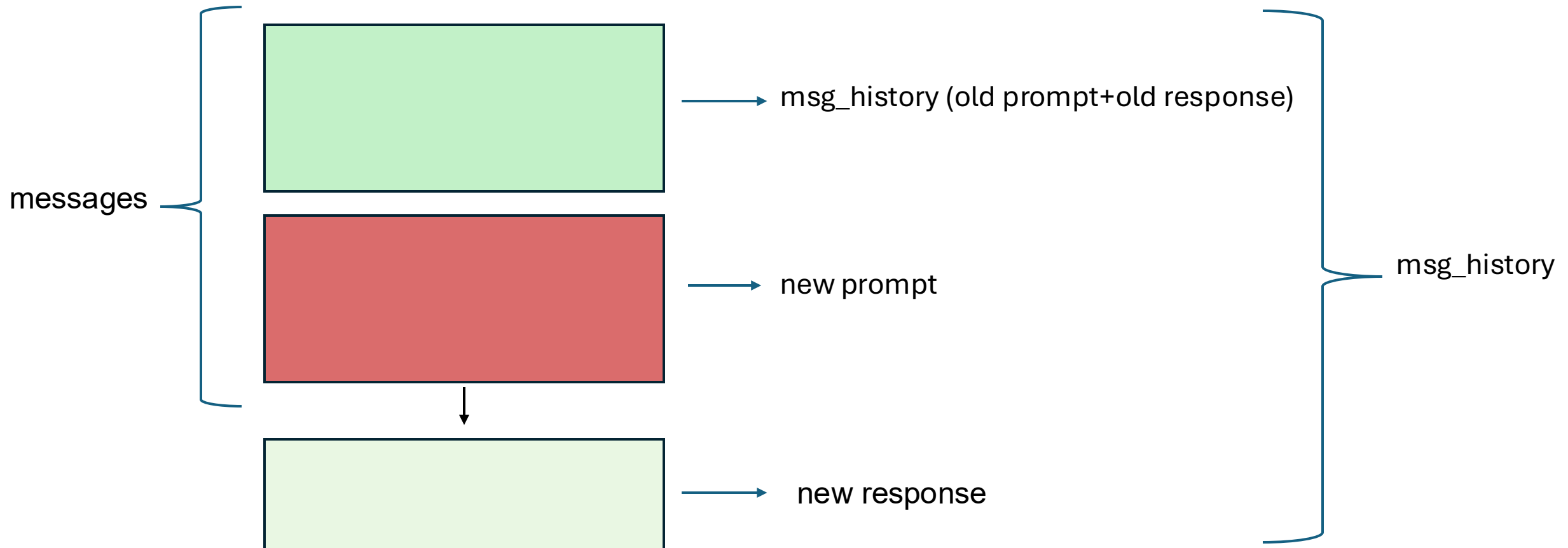


Now let's add another agent that interacts with the first agent

---

```
messages=[  
  {"role": "user", "content": "prompt_1"},  
  {"role": "assistant", "content": "response_1"},  
  {"role": "user", "content": "prompt_2"},  
]
```

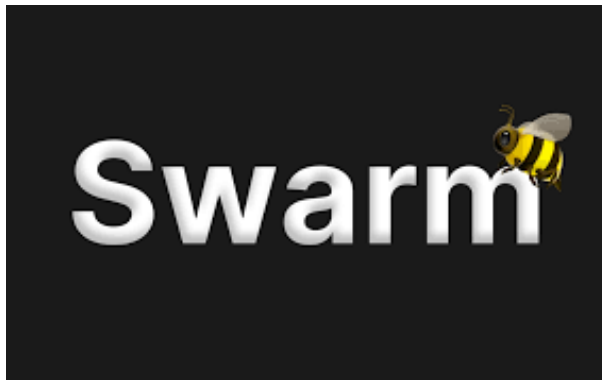
msg\_history



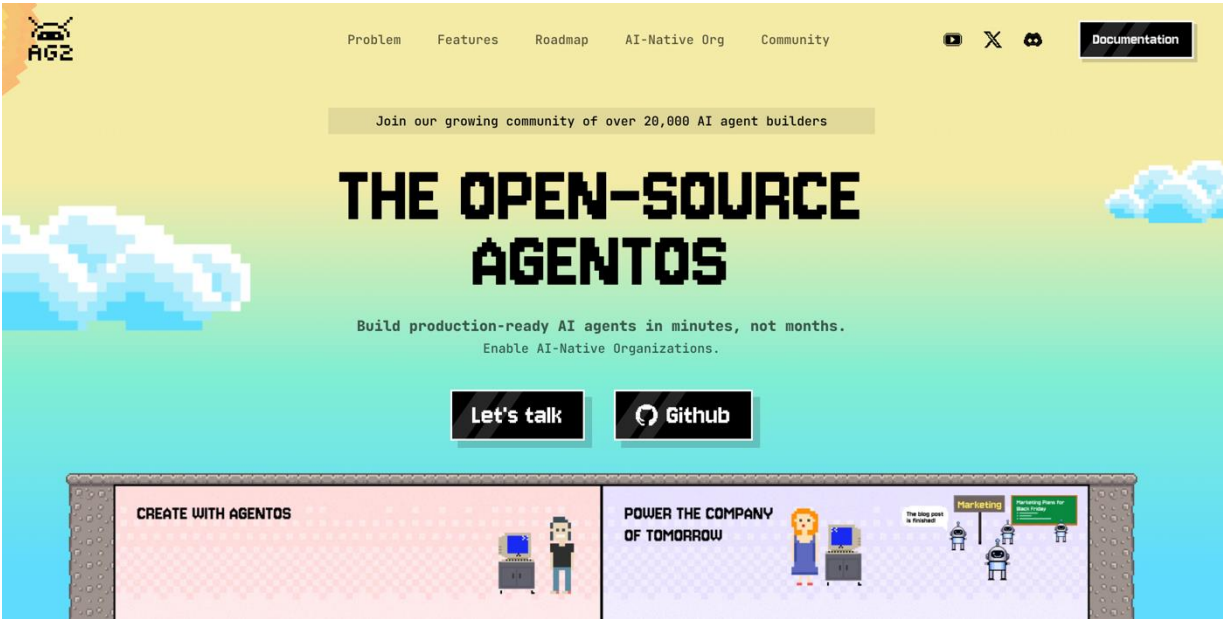
AG2: Building fully automated multi-agent systems

## Multi-agent frameworks

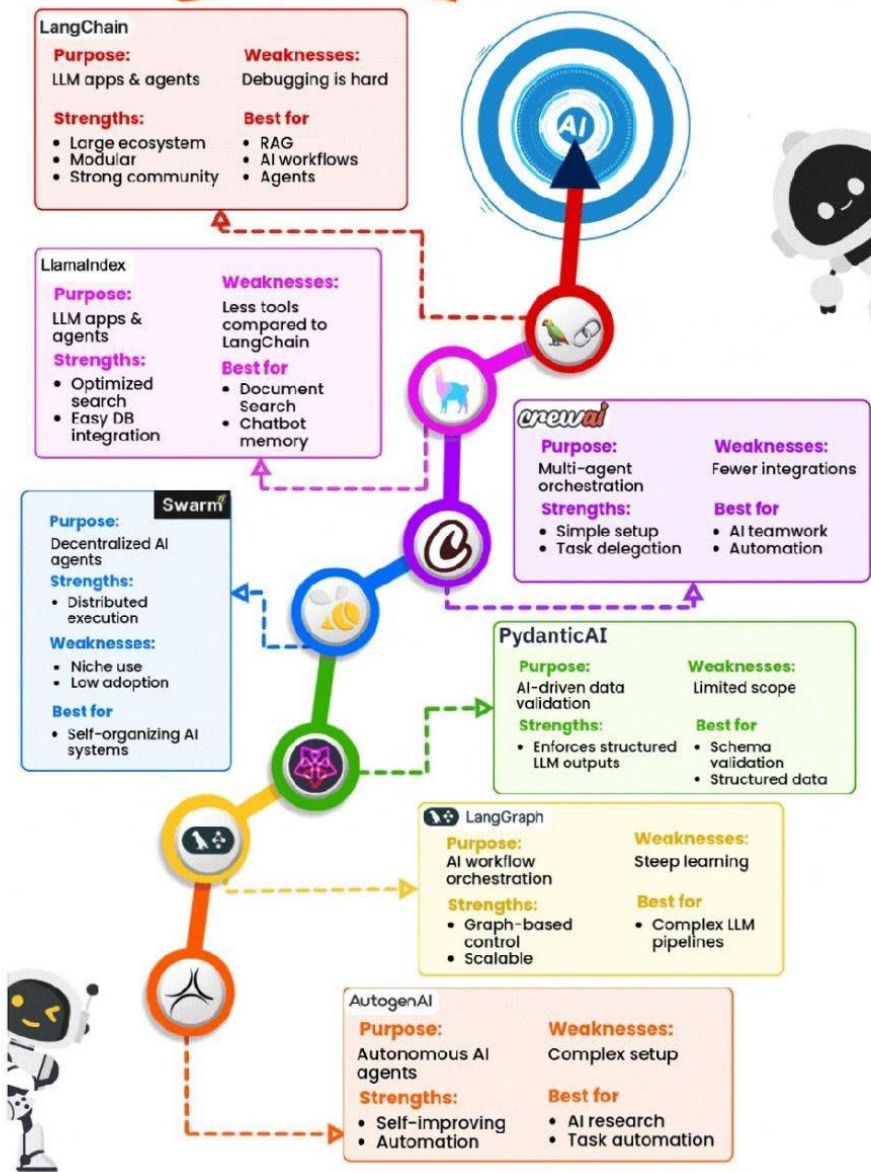
---



# Multi-agent frameworks



# Top Agentic AI Frameworks



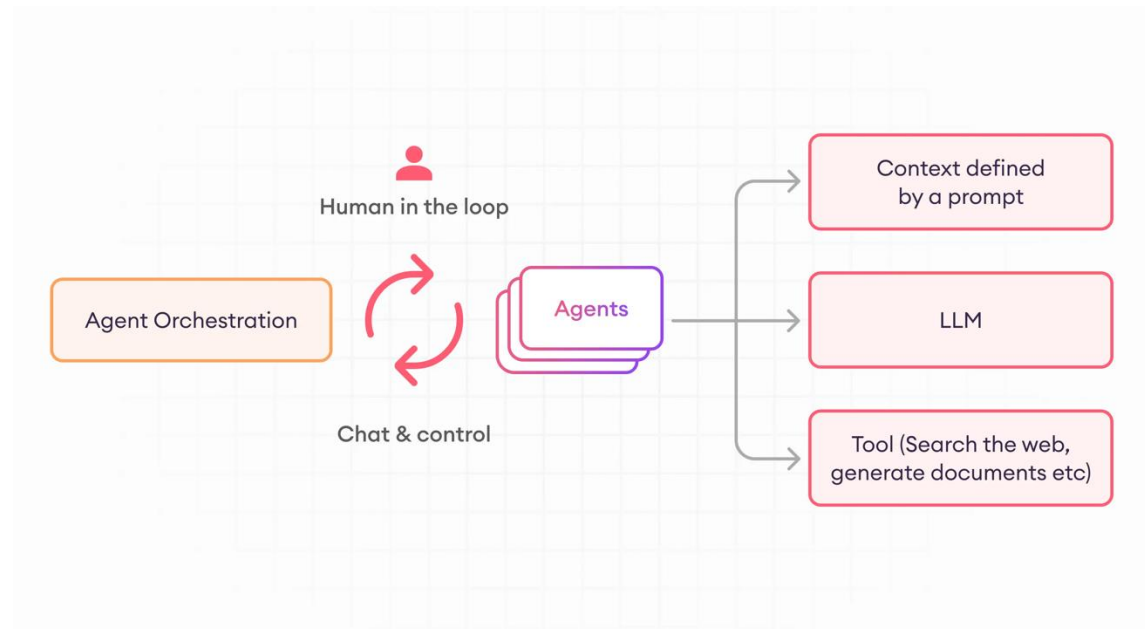
# AG2

---

## Installing AG2

```
pip install "ag2[openai]"  
pip install "ag2[gemini]"  
pip install "ag2[anthropic,cohere,mistral]"
```

- Basic Concepts
  - LLM Configuration
  - ConversableAgent
  - Human in the Loop
  - Agent Orchestration
  - Tools
  - Structured Outputs



## LLM configuration

---

LLM Configuration controls how an agent:

- Connects to and authenticates with language model providers
- Selects models and sets parameters
- Thinks, reasons, and generates responses

AG2 supports a variety of LLM providers

- **Cloud Models:** OpenAI, Anthropic, Google (Gemini), Amazon (Bedrock), Mistral AI, Cerebras, Together AI, and Groq
- **Local Models:** Ollama, LiteLLM, and LM Studio

```
import os
from autogen import LLMConfig

llm_config = LLMConfig(
    api_type="openai",           # The provider
    model="gpt-4o-mini",        # The specific model
    api_key=os.environ["OPENAI_API_KEY"], # Authentication
)
```

## ConversableAgents: Building Intelligent Agents

---

The [ConversableAgent](#) is the fundamental building block of AG2.

With an LLM configuration providing its thinking power, a ConversableAgent can:

- Communicate with other agents and humans
- Process information and generate responses
- Follow instructions defined in its system message
- Execute tools and functions when needed

```
from autogen import ConversableAgent, LLMConfig

# Create LLM configuration first
llm_config = LLMConfig(api_type="openai", model="gpt-4o-mini")

# Create the agent using the context manager approach
my_agent = ConversableAgent(
    name="helpful_agent", # Give your agent a unique name
    system_message="You are a helpful AI assistant", # Define its personality and purpose
    llm_config=llm_config # Pass the LLM configuration
)
```



## Human in the loop: Adding human oversight

---

A powerful pattern that enables AG2 agents to **collaborate with humans** throughout the workflow.

Humans can:

- Oversee the entire process
- Intervene when necessary
- Inject new queries dynamically

```
from autogen import ConversableAgent

# Create a human agent that will always prompt for input
human = ConversableAgent(
    name="human",
    human_input_mode="ALWAYS", # Always ask for human input ["ALWAYS"|"NEVER"|"TERMINATE"]
)

# Create an AI agent that never asks for human input directly
ai_agent = ConversableAgent(
    name="ai_assistant",
    system_message="You are a helpful AI assistant",
    human_input_mode="NEVER", # Never ask for human input directly
)
```

## ConversableAgents: Building Intelligent Agents

---

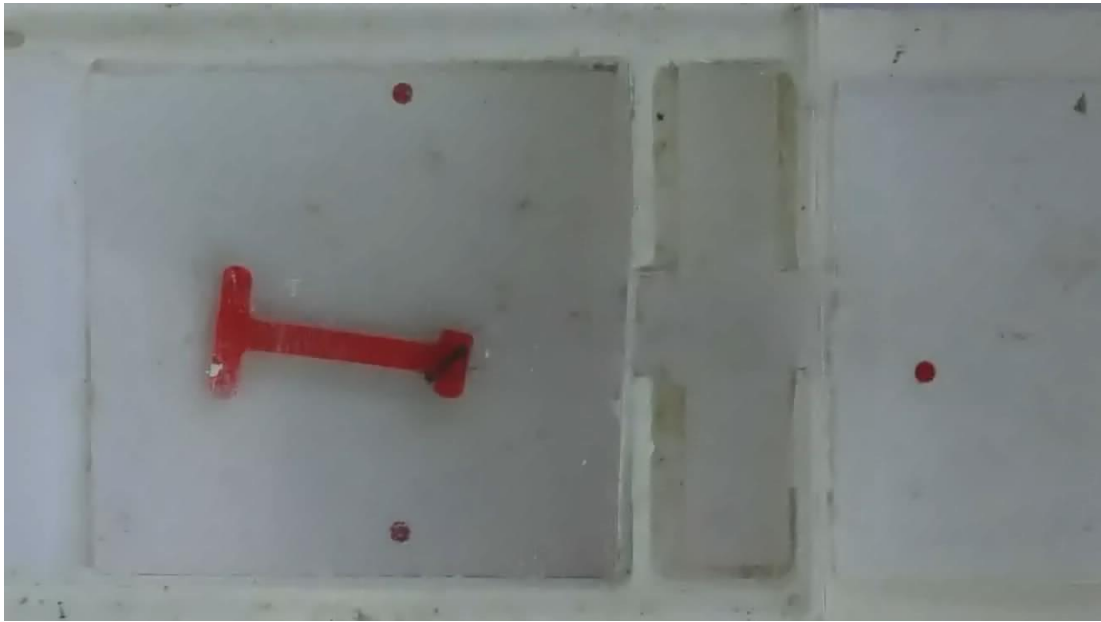
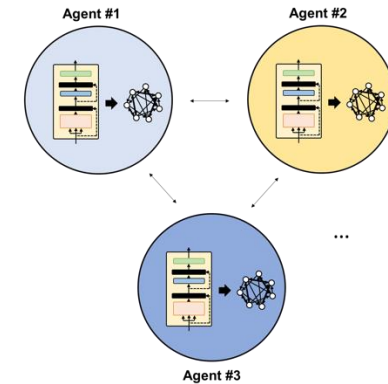
In Notebook 1&2 we'll see how to:

- load and set openai api key
- Define LLM configuration
- Build our first agent with AG2
- Interact with the agent using `run()` and `process()`
- add human in the loop capacity

# Agent Orchestration: Coordinating Multiple Agents

## Why multi-agent systems?

- Using multiple "simple" unitary components to build out higher complexity
- Improving LLMs through reflection, planning, and tool use strategies to produce more accurate results
- Interact to model "thought" – iterations, critique, improvements



T. Dreyer et al. *PNAS* (2024)



We find that when ants work in groups, their performances rise significantly...

## Overview

---

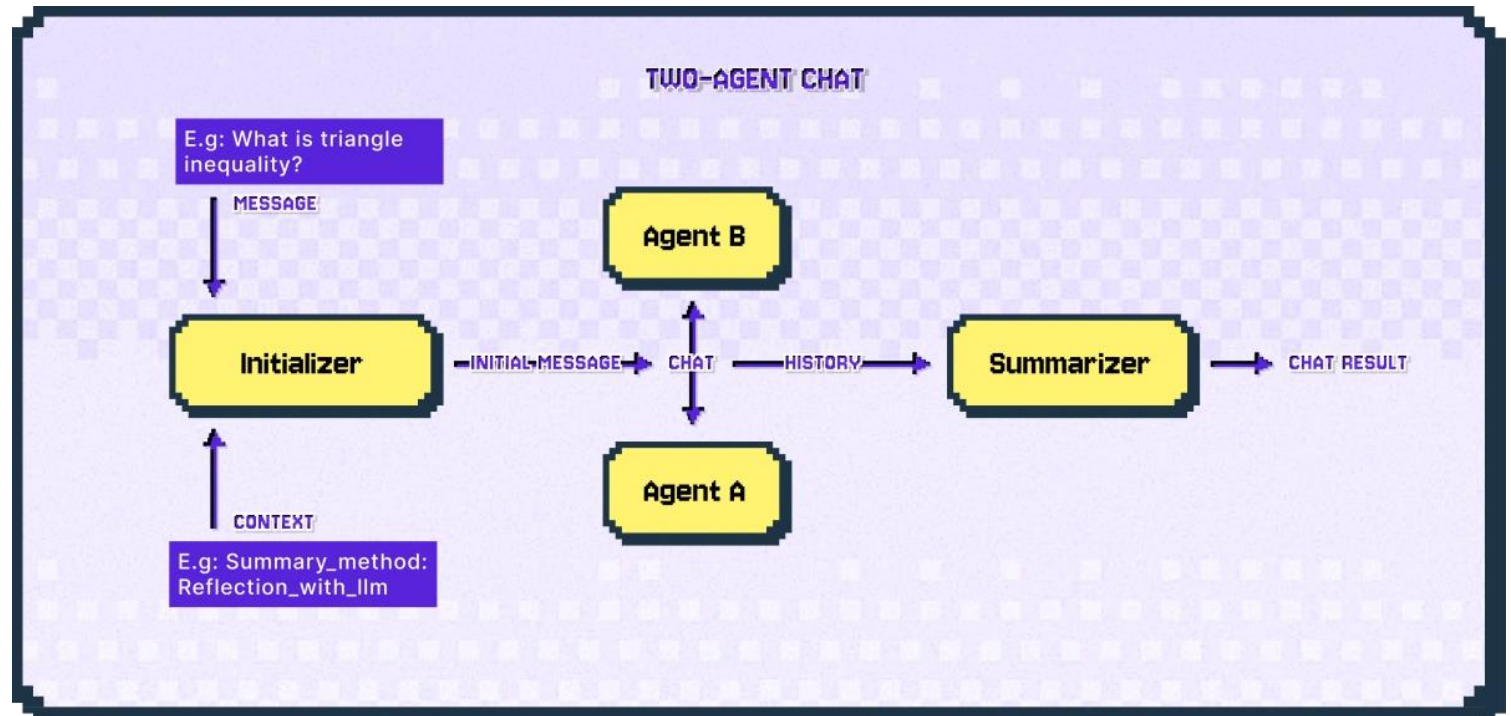
- **Two-agent chat:** The simplest form of conversation pattern where two agents chat back-and-forth with each other.
- **Sequential chat:** A sequence of chats, each between two agents, chained together by a carryover mechanism (which brings the summary of the previous chat to the context of the next chat). Useful for simple sequential workflows.
- **Group chat:** A chat between more than two agents with options on how agents are selected.

## Two Agent Chat

- Two-agent chat is the simplest form of a conversation pattern.
- Start a two-agent chat using the `initiate_chat` of every `ConversableAgent` agent

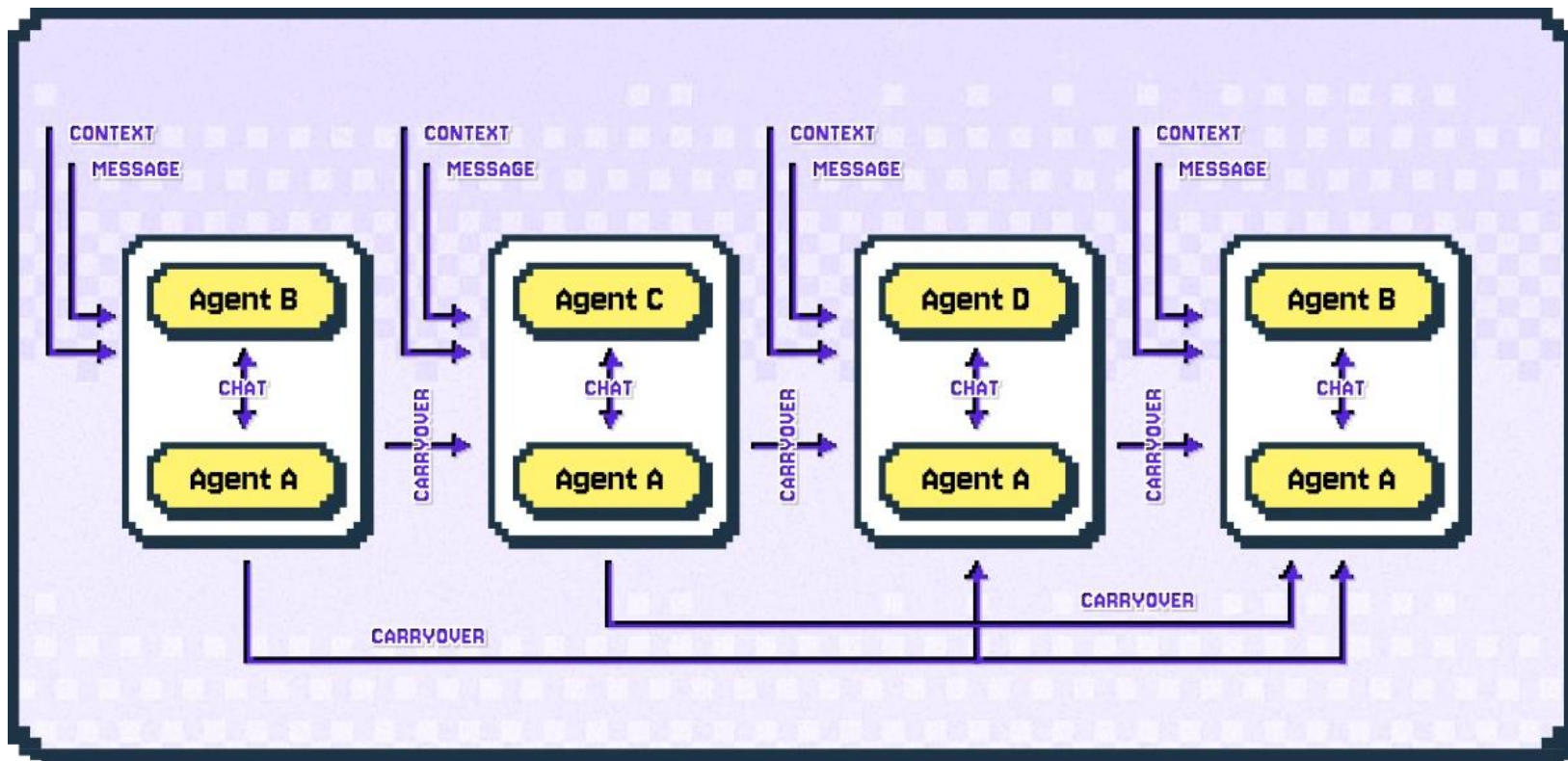
Notebook 3&4: Build a simple two-agent chat between two agents.

We learn how the `summay_method` parameter of the `initiate_chat` works in summarizing the history of the chat.



## Sequential Chat

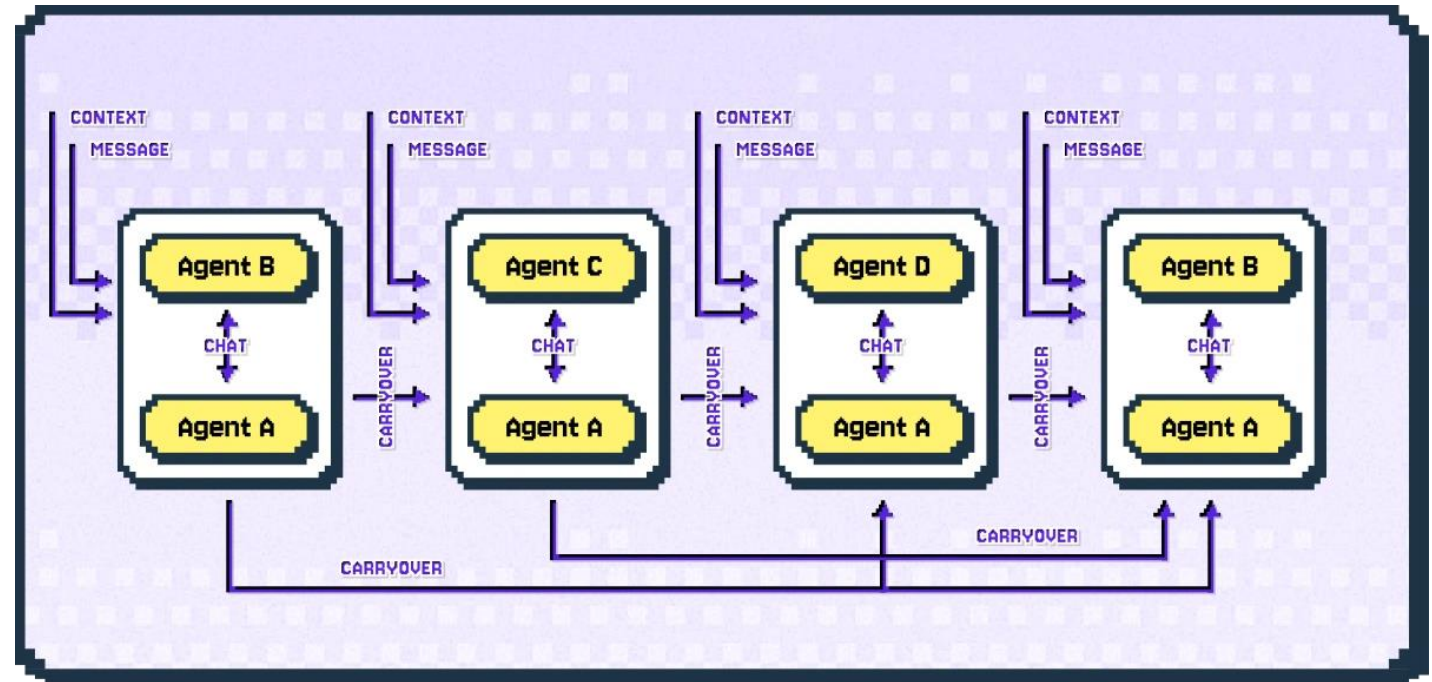
- A sequence of chats between two agents
- Useful for complex tasks that can be broken down into interdependent sub-tasks.
- The chats are chained together by a mechanism called *carryover*.
  - *The summary of the conversation between two agents becomes a carryover for the next two-agent chat*



# Sequential Chat

- Notes:
  - Carryover accumulates as the conversation moves forward, so each subsequent chat starts with all the carryovers from previous chats.

Notebook 5: Build a sequential chat



## Group chat

---

- Specialized expertise: Each agent can focus on what it does best, creating a team of specialists
- Dynamic collaboration: Agents can interact with each other and respond to new information as it emerges
- Flexible handoffs: Control can move between agents based on context, conversation state, or explicit directions
- Shared context: All agents work within the same conversation context, maintaining continuity
- Scalable complexity: Easily add new capabilities by introducing specialized agents rather than making existing agents more complex



## Group chat

---

Implementing a group chat in AG2 is a simple three-step process

- First, build the **agents**
- Then, **create a pattern** that defines how agents will interact
- Finally, **initialize the group chat** using the pattern

# Patterns

---

The pattern defines the orchestration logic - which agents are involved, who speaks first, and how to transition between agents.

AG2 provides several pre-defined patterns to choose from:

- **DefaultPattern:** A minimal pattern for simple agent interactions where the handoffs and transitions needs to be explicitly defined
- **AutoPattern:** Automatically selects the next speaker based on conversation context
- **RoundRobinPattern:** Agents speak in a defined sequence
- **RandomPattern:** Randomly selects the next speaker
- **ManualPattern:** Allows human selection of the next speaker

The easiest pattern to get started with is the **AutoPattern**, where a group manager agent automatically selects agents to speak by evaluating the messages in the chat and the descriptions of the agents.

# Pattern implementation

---

```
from autogen import ConversableAgent, LLMConfig
from autogen.agentchat import initiate_group_chat
from autogen.agentchat.group.patterns import AutoPattern

# Create your specialized agents
with llm_config:
    agent_1 = ConversableAgent(name="agent_1", system_message="...")
    agent_2 = ConversableAgent(name="agent_2", system_message="...")

# Create human agent if needed
human = ConversableAgent(name="human", human_input_mode="ALWAYS")

# Set up the pattern for orchestration
pattern = AutoPattern(
    initial_agent=agent_1,          # Agent that starts the workflow
    agents=[agent_1, agent_2],      # All agents in the group chat
    user_agent=human,               # Human agent for interaction
    group_manager_args={"llm_config": llm_config} # Config for group manager
)

# Initialize the group chat
result, context_variables, last_agent = initiate_group_chat(
    pattern=pattern,
    messages="Initial request",     # Starting message
)
```

## Agent Orchestration: Coordinating Multiple Agents

---

In Notebook 6-9 we'll see how to build a multi-agent system with AutoPattern.

## Tools: Extending Agent Capabilities

---

- Tools provide a structured way for agents to interact with the outside world.
- Think of tools as specialized functions that agents can choose to use when appropriate.
- With tools we provide AI with ability to generate its own data or retrieve knowledge from different sources

Tool usage in AG2 follows a two-step process:

- **Selection:** An agent (driven by its LLM) decides which tool is appropriate based on the given task
- **Execution:** A separate executor agent invokes the tool and returns the results

# Enhancing multi-agent system with tools

---

## Empowering a group chat in AG2 with tools

- Define the tools
- Then, build the **agents** and assign the **tools**
- Then, **create a pattern** that defines how agents will interact
- Finally, **initialize the group chat** using the pattern

```
# Define the tool (python function)
def tool_name(query: str): -> ReplyResult

    """Tool description. This prompt helps the agent understand the tool."""

    return ReplyResult(
        message = f"The output of this tool is ..."
    )

# Create LLM configuration first
llm_config = LLMConfig(api_type="openai", model="gpt-4o-mini")

# Create the agent using the context manager approach
my_agent = ConversableAgent(
    name="helpful_agent", # Give your agent a unique name
    system_message="You are a helpful AI assistant", # Define its personality and ,
    llm_config=llm_config # Pass the LLM configuration
    function=[tool_name] # List of tool names
)
```

```
# Create LLM configuration first
llm_config = LLMConfig(api_type="openai", model="gpt-4o-mini")

# Create the agent using the context manager approach
my_agent = ConversableAgent(
    name="helpful_agent", # Give your agent a unique name
    system_message="You are a helpful AI assistant", # Define its personality and ,
    llm_config=llm_config # Pass the LLM configuration
)

# Define the tool (python function)
def tool_name(query: str): -> ReplyResult

    """Tool description. This prompt helps the agent understand the tool."""

    return ReplyResult(
        message = f"The output of this tool is ..."
    )

my_agent.functions = [tool_name]
```

## Enhancing multi-agent system with tools

---

Each tool function can return a `ReplyResult`

```
def my_tool_function(param: str, context_variables: ContextVariables) -> ReplyResult:
    return ReplyResult(
        message="Tool result message",
        target=AgentTarget(next_agent)
        context_variables=context_variables # Updated context
    )
```

In Notebook 10 we first create a multi-agent system for tech support.

Then, in Notebook 11, we'll see how to add tools to our multi-agent system.

## Context Variables (overview)

---

**Context Variables** are a structured way to store and share information between agents in a group chat. They act as a persistent memory that:

- Maintains state throughout the entire conversation
- Is accessible to all agents in the group
- Can be read and updated by any agent or tool
- Stores data in a key-value format

In this section you will learn how to:

- Create and initialize context variables
- Read from and write to the shared context
- Pass information between agents
- Use context variables with tools to create more powerful workflows



## Context Variables (implementation)

---

Context variables in AG2 are implemented through the `ContextVariables` class, which provides a dictionary-like interface to store and retrieve values:

### Creating and Initializing Context Variables

```
from autogen.agentchat.group import ContextVariables

# Initialize context variables with initial data
context = ContextVariables(data={
    "user_name": "Alex",
    "issue_count": 0,
    "previous_issues": []
})

# Create pattern with the context variables
pattern = AutoPattern(
    initial_agent=coordinator_agent,
    agents = [coordinator_agent, tech_agent, general_agent],
    user_agent=user,
    context_variables=context,    # Pass context variables to the pattern
    group_manager_args={"llm_config": llm_config}
)
```

## Context Variables (implementation)

---

### Reading and Writing Context Values

```
# Reading values
user_name = context.get("user_name") # Returns "Alex"
non_existent = context.get("non_existent", "default") # Returns "default"

# Writing values
context.set("issue_count", 1) # Sets issue_count to 1
context.update({"last_login": "2023-05-01", "premium": True}) # Update multiple values

# Dictionary-like operations
user_name = context["user_name"] # Get a value
context["issue_count"] = 2 # Set a value
del context["temporary_value"] # Delete a value
if "premium" in context: # Check if a key exists
    print("Premium user")
```

## Context Variables (implementation)

---

### Reading and Writing Context Values

```
# Reading values
user_name = context.get("user_name") # Returns "Alex"
non_existent = context.get("non_existent", "default") # Returns "default"

# Writing values
context.set("issue_count", 1) # Sets issue_count to 1
context.update({"last_login": "2023-05-01", "premium": True}) # Update multiple values

# Dictionary-like operations
user_name = context["user_name"] # Get a value
context["issue_count"] = 2 # Set a value
del context["temporary_value"] # Delete a value
if "premium" in context: # Check if a key exists
    print("Premium user")
```

## Context Variables (implementation)

---

### Persistent across agent transitions

One of the most powerful features of context variables is their persistence across agent transitions. When control passes from one agent to another, the context variable go with it, maintaining the shared state.

```
def route_to_tech_support(issue: str, context_variables: ContextVariables) -> ReplyResult:
    """Route an issue to texhnical support."""
    # Update the context with the current issue
    context_variables["current_issue"] = issue
    context_variables["issue_count"] += 1
    context_variables["previous_issues"].append(issue)

    # Return control to the tech agent with the updated context
    return ReplyResult(
        message="Routing to technical support...",
        target=AgentTarget(tech_agent),
        context_variables=context_variables, # Update the shared context
```

Note:

- You should always define a tool to update the *context variables*.

## Context Variables (implementation)

---

In Notebook 12 we'll see how to enhance our tech support system with `ContextVariables`.

## Handoffs (overview)

---

**Handoffs** are the essential mechanism that controls **how agents interact and pass control to each** other in a group chat. If tools represent what agents can do and context variables represent what they know, **handoffs determine where they go next**.

In multi-agent systems, you need to carefully **control which agent speaks** when and handoffs provide you this control.

In this section you will learn how to:

- Define explicit handoff conditions between agents
- Use context variables to drive dynamic transitions
- Implement complex workflows with conditional branching
- Combine tools, context variables, and handoffs for sophisticated agent orchestration

## Handoffs (concepts)

---

Each agent in AG2 has a `handoffs` attribute that manages transitions from that agent.

**The `handoffs` attribute is an instance of the `Handoffs` class.**

When defining a handoff, you specify a transition target – the destination where control should go. AG2 provides several types of transition targets:

### Transition Targets:

- `AgentNameTarget`: Transfer control to an agent by name
- `AgentTarget`: Transfer control to a specific agent instance
- `AskUserTarget`: Ask the user to select the next speaker
- `GroupManagerTarget`: Transfer control to the group manager who will select the next speaker
- `RandomAgentTarget`: Randomly select from a list of agents
- `RevertToUserTarget`: Return control to the user agent
- `StayTarget`: Keep control with the current agent
- `TerminateTarget`: End the conversation

## Handoffs (implementation)

---

```
from autogen.agentchat.group import AgentTarget, RevertToUserTarget, TerminateTarget

# Transition to a specific agent
target = AgentTarget(tech_agent)

# Return to the user
target = RevertToUserTarget()

# End the conversation
target = TerminateTarget()
```

```
def my_tool_function(param: str, context_variables: ContextVariables) -> ReplyResult:
    return ReplyResult(
        message="Tool result message",
        target=AgentTarget(next_agent)
        context_variables=context_variables # Updated context
    )
```



## Handoffs (implementation)

---

AG2 offers four main ways to define handoffs:

- **Explicit handoffs from tools:** Direct transitions specified by tool return values
- **LLM-based conditions:** Transitions based on language model's analysis of messages
- **Context-based conditions:** Transitions based on values in context variables
- **After-work behavior:** Default transition when no LLM or context conditions are met and no tools are called

## Handoffs (implementation)

---

### Explicit Handoffs from tools

Tools can specify transitions directly in their return values

```
def classify_query(query: str, context_variables: ContextVariables) -> ReplyResult:
    if is_technical:
        return ReplyResult(
            message="This is a technical problem,",
            target=AgentTarget(tech_agent),
            context_variables=context_variables
        )
    else:
        return ReplyResult(
            message="This is a general question.",
            target=AgentTarget(general_agent),
            context_variables=context_variables
        )
```

## Handoffs (implementation)

---

### LLM-Based Conditions

LLM-based conditions use the LLM to determine when a transition should occur. This is useful when the decision depends on understanding the meaning of messages

```
from autogen.agentchat.group import OnCondition, StringLLMCondition

# Set up LLM-based handoffs
coordinator_agent.handoffs.add_llm_conditions([
    OnCondition(
        target=AgentTarget(tech_agent),
        condition=StringLLMCondition(prompt="When the user query is related to technical issues!"),
    ),
    OnCondition(
        target=AgentTarget(general_agent),
        condition=StringLLMCondition(prompt="When the user query is related to general questions."),
    )
])
```

## Handoffs (implementation)

---

### Context-Based Conditions

Context-based conditions transition based on the values of context variables. This is ideal for decisions that depend on the system's state rather than message content.

```
from autogen.agentchat.group import OnContextCondition, ExpressionContextCondition, ContextExpression

# Set up context-based handoffs for the tech agent
tech_agent.handoffs.add_context_condition(
    OnContextCondition(
        target=AgentTarget(escalation_agent),
        condition=ExpressionContextCondition(
            expression=ContextExpression("${issue_severity} >= 8")
        )
    )
)
```

This example transitions to an escalation agent when the `issue_severity` context variable is 8 or higher.

## Handoffs (implementation)

---

### After-Work Behavior

After-work behavior defines the default transition that occurs after an agent completes its work and no specific condition is met.

```
# Set the default after-work transition  
tech_agent.handoffs.set_after_work(RevertToUserTarget())
```

In this example, control returns to the user after the tech agent finished speaking, unless another condition routes it elsewhere.

## Handoffs (implementation)

---

In notebook 13, we build a tech support system to demonstrate different types of handoffs and how they work together to create a seamless user experience.

## General Guidelines

---

### ◆ **Start with a Plan**

What steps are needed?

- How many agents (workers) would you assign?
- What should each one do?

### ◆ **Define the Right Tools**

- Need external data (web, APIs, files)? → Add **tool agents**.
- Need shared knowledge or memory? → Use **context variables**.

### ◆ **Use Handoffs for Complex Tasks**

- If the task involves multiple stages or roles, structure it with **agent-to-agent handoffs**.

### ◆ **Start Small, Then Scale**

- Begin with a simple setup to test core logic.
- Scale to more agents, tools, and coordination once it works.

### ◆ **Prompt Clearly and Generously**

- Don't hesitate to provide detailed instructions—LLMs thrive on well-structured prompts.
- Clarify roles, expectations, and outputs explicitly.

## Project ideas

---

<https://manus.im/>

- **AI Study Buddy:** An interactive AI tutor that helps students understand concepts, quizzes them, and explains answers in different ways until they understand.
- **AutoSyllabus:** A system that generates an entire course syllabus, lecture content, and slides based on a course title and target audience.
- **SiteForge Agents:** Multi-agent system to design and deploy a website from scratch—UI design, frontend/backend code, and deployment setup—based on user goals.
- **GrantWriter:** A team of agents that draft compelling research or nonprofit grant proposals from a project description, with agents focused on technical writing, budget, impact, and formatting.
- **EventPlanner AI:** Multi-agent assistant that plans events (e.g. conferences or weddings) including venue suggestions, guest list organization, budgeting, and personalized invites.
- **LLM Dungeon Master:** Agents act as game master, player, and narrator to generate a collaborative storytelling game (e.g. D&D-style text adventure).



## Project ideas

---

- **AutoJournalist:** A team of agents that collects news from APIs, summarizes articles, fact-checks, adds commentary, and formats it into a daily briefing or newsletter.
- **CareerCoachBot:** One agent collects the user's CV, another reviews it, a third generates job suggestions, and a fourth conducts mock interviews.
- **CodeRefactor Crew:** A system where one agent analyzes legacy code, another improves it, one adds documentation, and another tests for bugs or vulnerabilities.
- **Startup Simulator:** Agents play roles of CEO, CTO, marketing lead, and investor to simulate how a startup could pitch, plan, and iterate on a business idea.
- **LegalDoc Builder:** Agents generate custom contracts (e.g. NDA, lease, freelance agreements), with one focused on legality, one on clarity, one on formatting, and one on risk.
- **HealthCoach:** A multi-agent system where a symptom analyzer, diagnosis assistant, treatment planner, and compliance tracker work together to guide patients through symptom checking, care suggestions, and follow-up support.