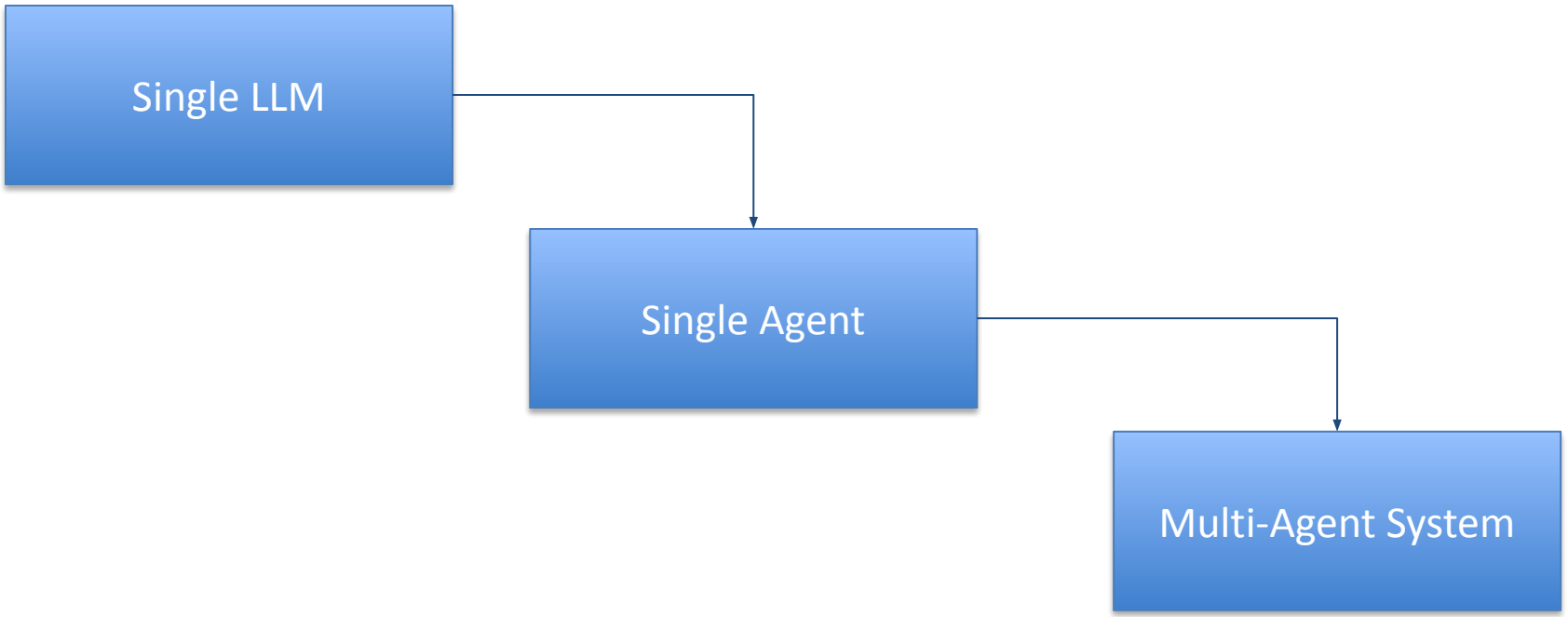
Building Multi-Agent Systems with AutoGen

-Annu Sachan
Principal Manager-AI/ML

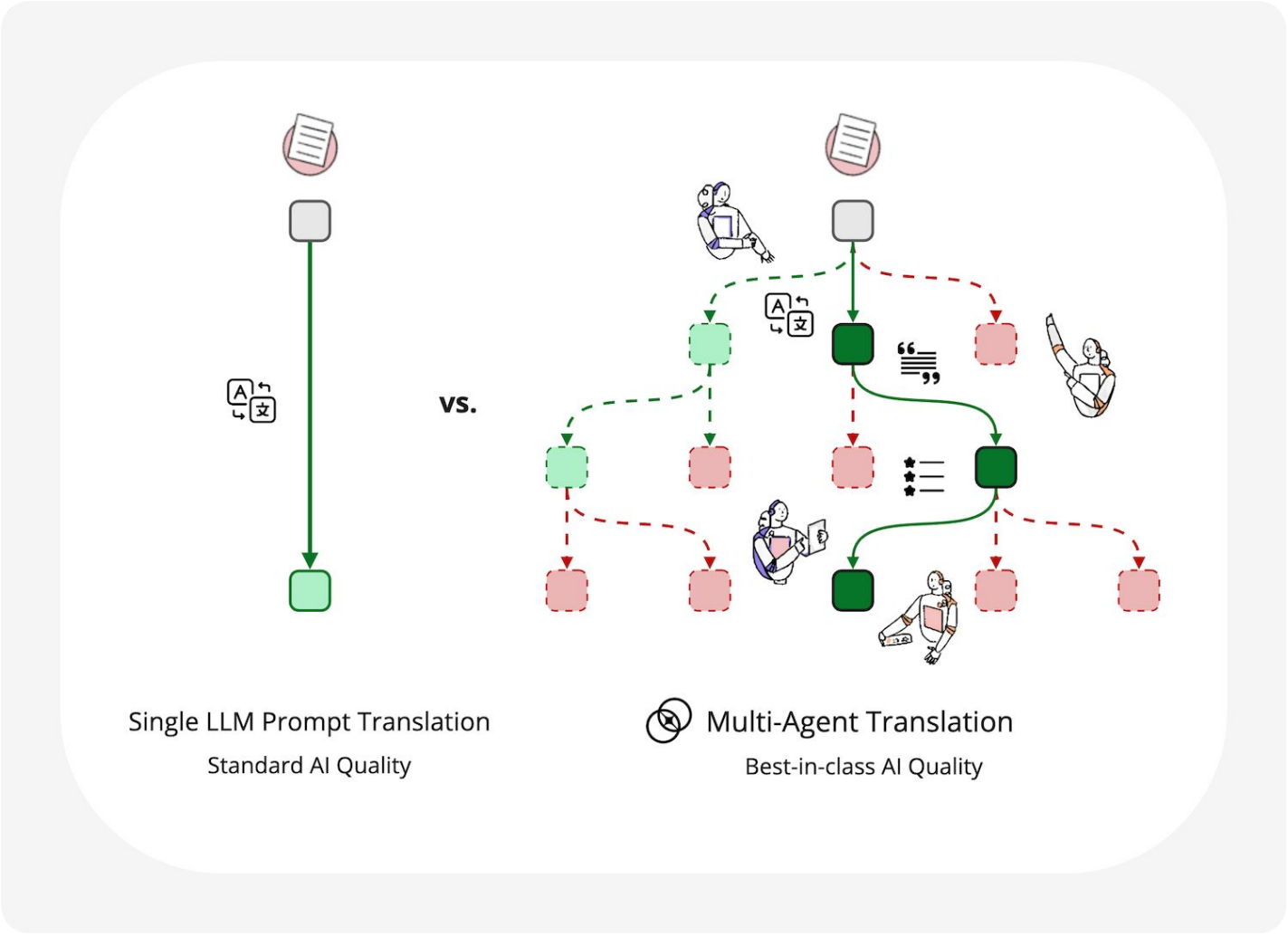
Conceptual Progression



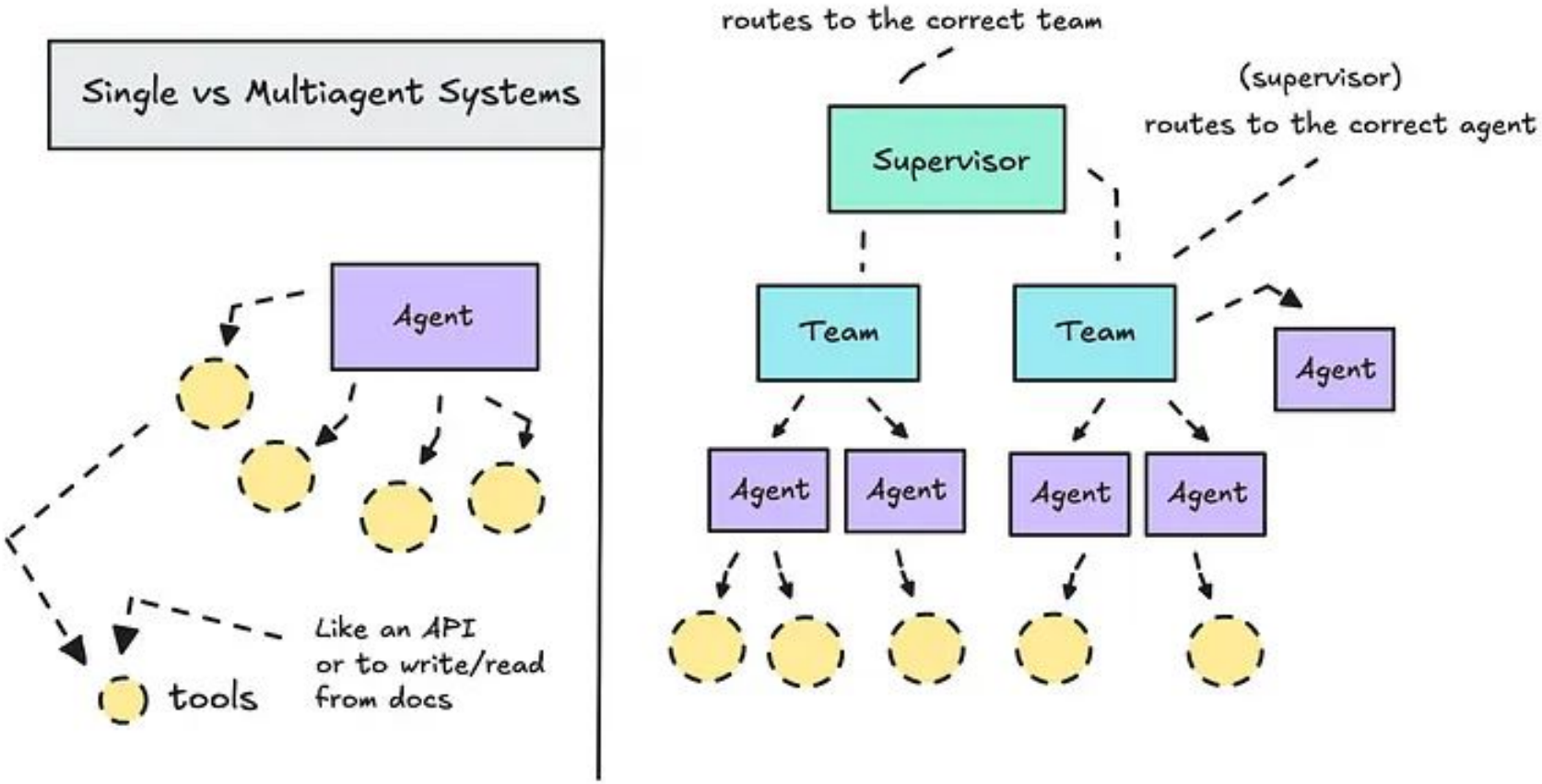
Why Multi-Agent Systems?

- Single LLM limitations
- Role-based agents
- Coordination and specialization
- Scalable reasoning

Single LLM vs Multi Agent



Single Agent vs Multi-Agent



Agent Framework Landscape

- AutoGen – conversation-first
- LangGraph – graph-based workflows
- CrewAI – role orchestration
- Google ADK – Governed enterprise agents
- OpenAI Swarm – Lightweight coordination

Langgraph

Best for: Deterministic, stateful agent workflows

- DAG / state-machine based
- Explicit state persistence
- Strong retry & recovery semantics

⚠️ Less “free-form” agent chat

✅ Excellent for **production reliability**

CrewAI

Best for: Role-based task execution (manager → workers)

- Very intuitive mental model
- Fast to prototype
- Clean “crew / role / task” abstraction

⚠️ Less control over deep reasoning
✅ Great for PoCs and product teams

Google ADK

Best for: Enterprise-grade, deterministic agent workflows

- Gemini-first, plan-and-tool–driven agents
- Strictly typed tools and explicit execution flows
- Built-in observability, safety, and governance

⚠️ Less emergent agent behavior

✅ Excellent for regulated, production systems

OpenAI Swarm

Best for: Lightweight coordination

- Minimal abstraction
- Simple handoffs between agents
- Very hackable

⚠️ No orchestration primitives

✅ Good for experiments and teaching

Autogen

Best for: Research-grade, tool-heavy, human-in-the-loop systems

- Explicit agent conversations
- Strong tool orchestration
- Event-based logs (what you just saw)
- Supports group chats, selectors, supervisors

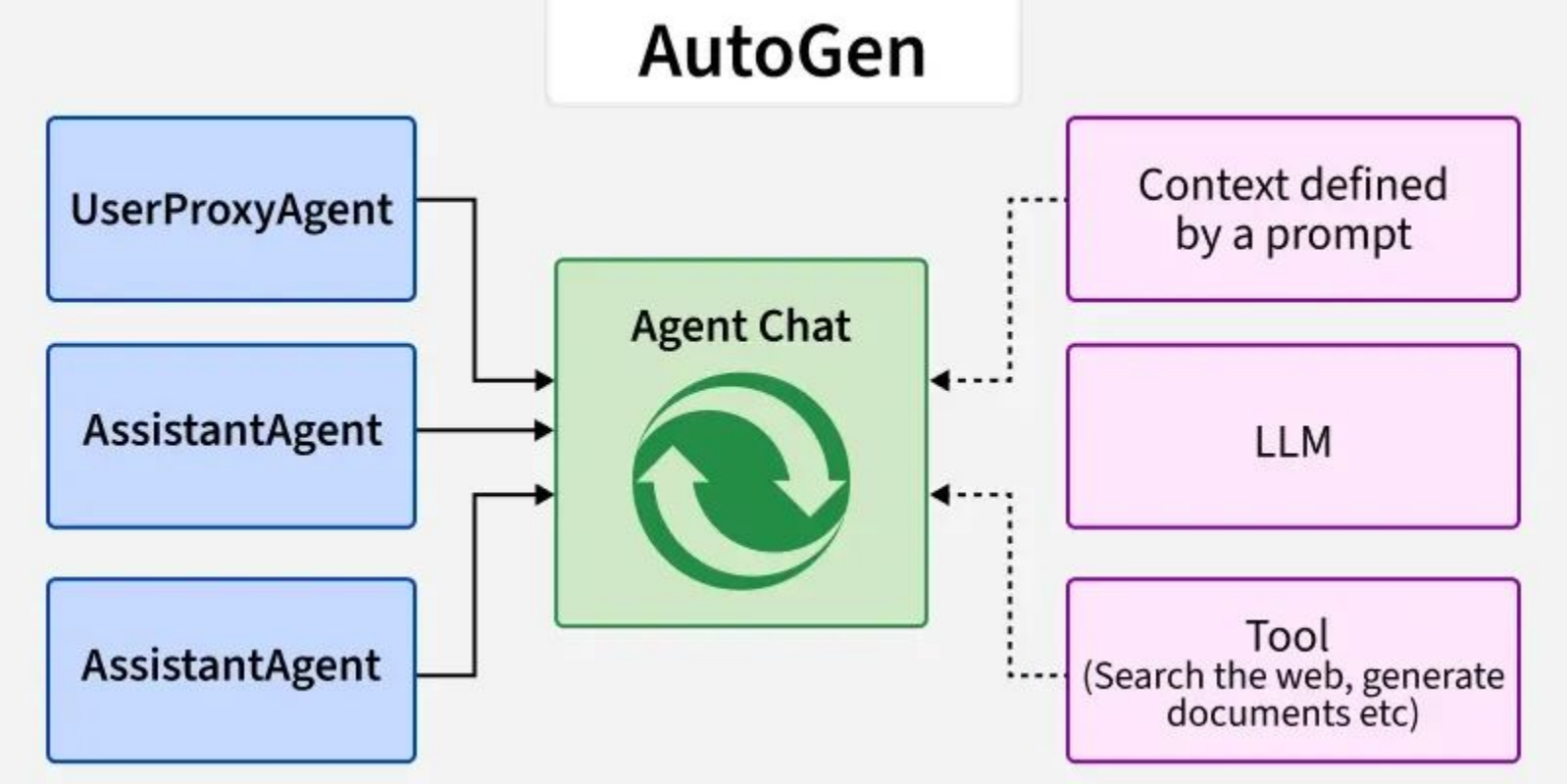
⚠️ Steeper learning curve

✅ Best for *enterprise* and *complex workflows*

Dimension	AutoGen	LangGraph	CrewAI	Google Agent Development Kit (ADK)	OpenAI Swarm
Primary goal	Emergent multi-agent reasoning	Deterministic orchestration	Fast role-based execution	Governed enterprise agents	Lightweight coordination
Agent interaction	Conversational	Graph transitions	Manager–worker	Plan & tool execution	Handoff-based
Control model	Emergent	State machine / DAG	Semi-structured	Explicit & strict	Minimal
Determinism	Medium	High	Medium	Very high	Low
State persistence	Optional / manual	First-class	Limited	Scoped & governed	None
Tool integration	Flexible	Structured	Flexible	Strongly typed	Minimal

Dimension	AutoGen	LangGraph	CrewAI	Google Agent Development Kit (ADK)	OpenAI Swarm
Observability	Event logs	Graph traces	Basic logs	Enterprise-grade tracing	Minimal
Failure handling	Prompt-driven	Retries & recovery	Limited	Policy-driven	Manual
Human-in-the-loop	Native	Possible	Limited	Controlled	Manual
Best LLM fit	Model-agnostic	Model-agnostic	Model-agnostic	Gemini-first	OpenAI-first
Setup complexity	High	Medium	Low	Medium–High	Very low
Production readiness	Medium–High	High	Medium	Very high	Low
Best use case	Complex agent collaboration	Reliable production workflows	Rapid PoCs & automation	Regulated enterprise systems	Teaching & demos

AutoGen Core Architecture



High-Level Architecture

AutoGen's architecture revolves around the concept of *conversational agents*. These agents communicate with each other through messages, forming collaborative workflows. The core components are:

- **Agents:** Represent individual entities with specific roles, responsibilities and capabilities.
- **Teams:** A team is a group of agents that work together to achieve a common goal.
- **Messages:** The communication medium between agents, carrying information and instructions.
- **Conversations:** The dynamic exchange of messages between agents to achieve a common goal.
- **GroupChat:** A multi-agent conversation environment managed by a group chat manager.
- **Tool Calling:** The mechanism by which agents can invoke external functions or APIs.

Agents

AutoGen AgentChat provides a set of preset Agents, each with variations in how an agent might respond to messages. All agents share the following attributes and methods:

- **name**: The unique name of the agent.
- **description**: The description of the agent in text.
- **run**: The method that runs the agent given a task as a string or a list of messages, and returns a **TaskResult**. Agents are expected to be stateful and this method is expected to be called with new messages, not complete history.
- **run_stream**: Same as **run()** but returns an iterator of messages that subclass **BaseAgentEvent** or **BaseChatMessage** followed by a **TaskResult** as the last item

Agent Type

AutoGen supports different types of agents, each with specific characteristics and roles:

- **AssistantAgent:** Designed to assist with specific tasks, leveraging LLMs for reasoning and problem-solving. They typically have pre-defined system prompts to guide their behavior.
- **UserProxyAgent:** Acts as a proxy for a human user, relaying messages and providing feedback. They can execute code and interact with the environment.

Termination

We explored how to define agents, and organize them into teams that can solve tasks. However, a run can go on forever, and in many cases, we need to know *when* to stop them. This is the role of the termination condition.

AgentChat supports several termination condition by providing a base **TerminationCondition** class and several implementations that inherit from it.

Some examples of Built-In Termination Conditions:

1. **MaxMessageTermination**: Stops after a specified number of messages have been produced, including both agent and task messages.
2. **TextMentionTermination**: Stops when specific text or string is mentioned in a message (e.g., “TERMINATE”).

For more refer to this link:
https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/tutorial/termination.html

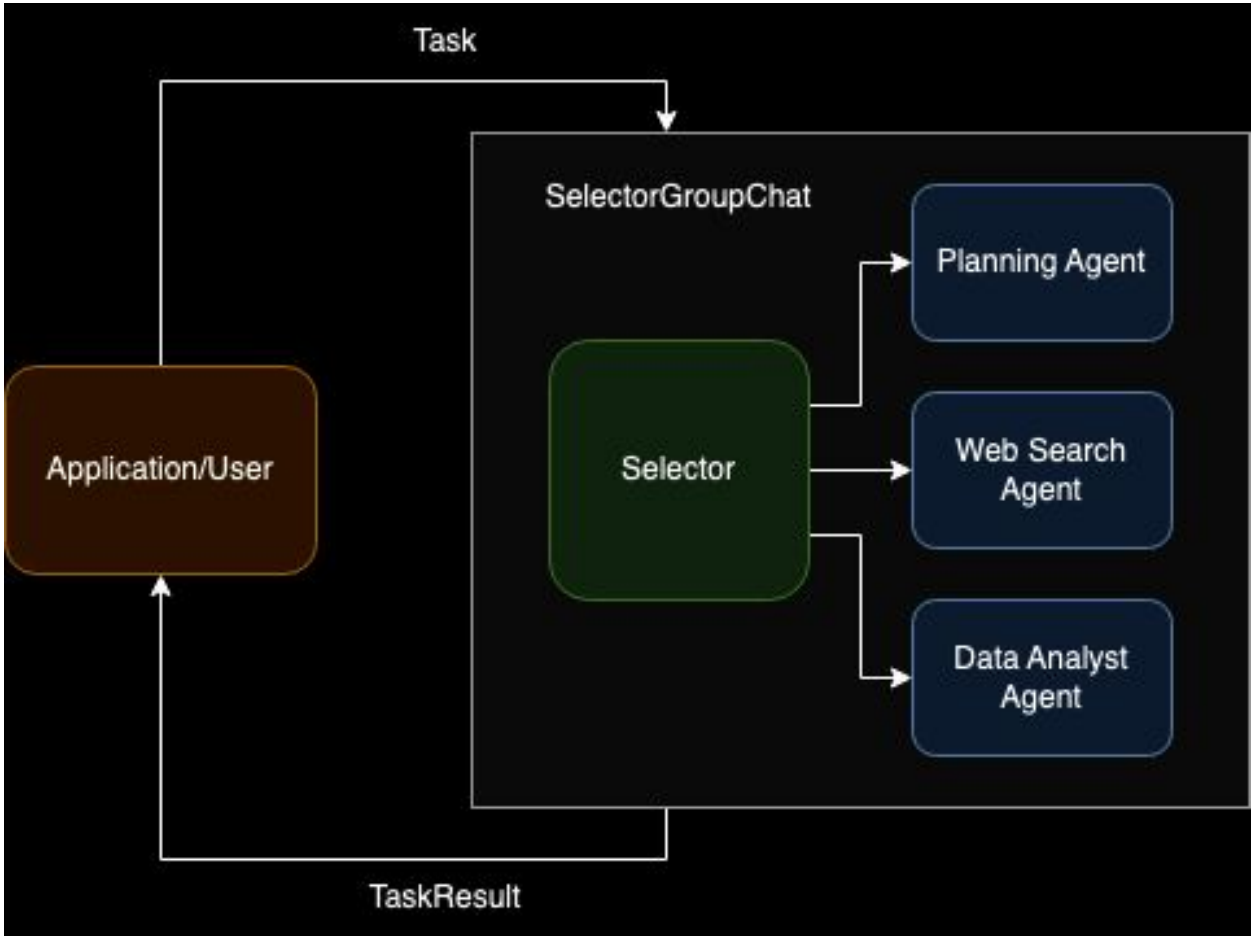
Teams

A team is a group of agents that work together to achieve a common goal.

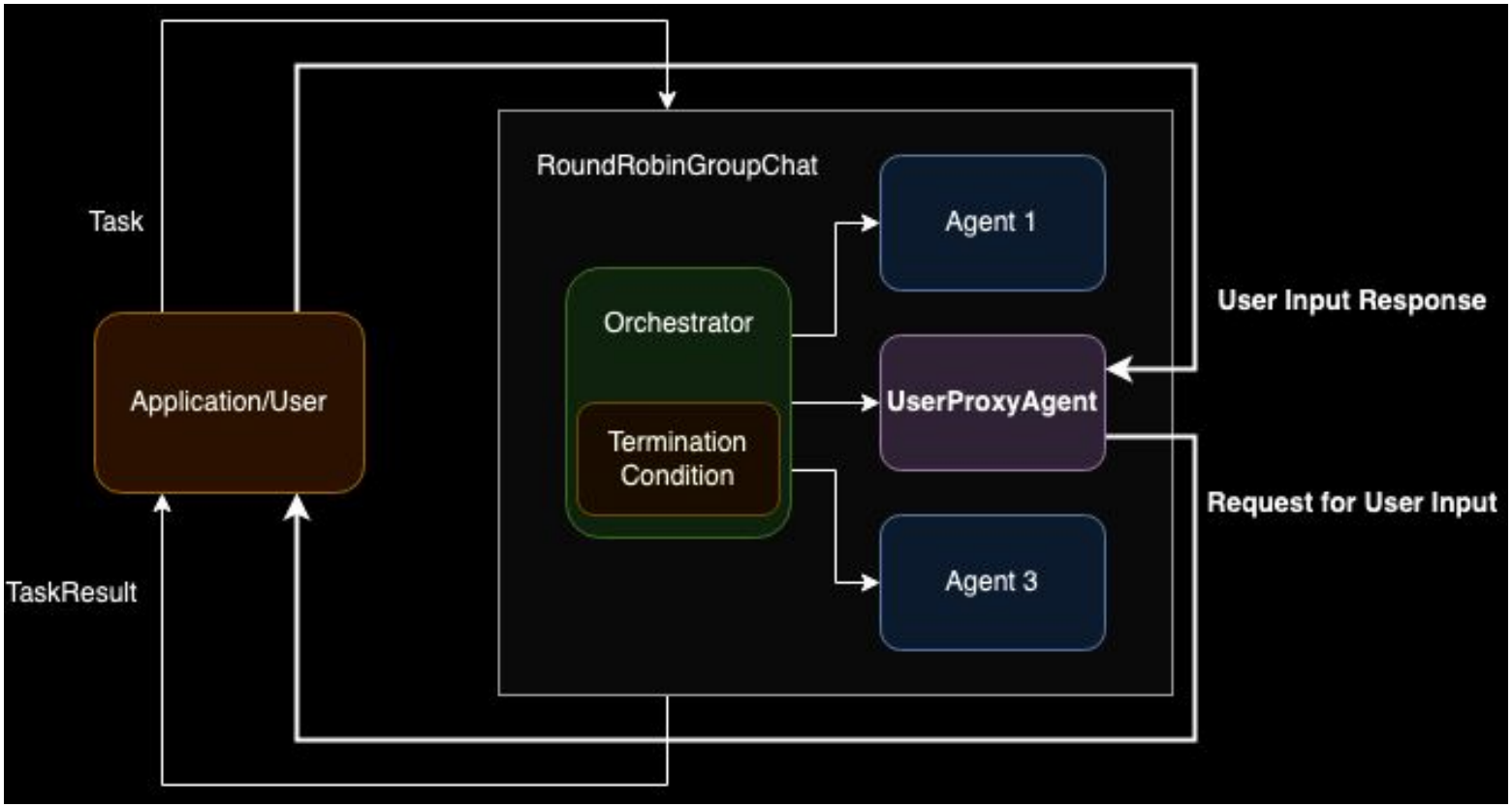
AgentChat supports several team presets:

- **RoundRobinGroupChat**: A team that runs a group chat with participants taking turns in a round-robin fashion (covered on this page). Tutorial
- **SelectorGroupChat**: A team that selects the next speaker using a ChatCompletion model after each message. Tutorial
- **MagenticOneGroupChat**: A generalist multi-agent system for solving open-ended web and file-based tasks across a variety of domains. Tutorial
- **Swarm**: A team that uses **HandoffMessage** to signal transitions between agents.

Selector Group Chat



Human-in-loop



Human + Agent Collaboration

State Management

- Local
- Mongo
- or any other database

For **AssistantAgent**, its state consists of the model_context. If you write your own custom agent, consider overriding the **save_state()** and **load_state()** methods to customize the behavior. The default implementations save and load an empty state.

Persistence & Memory

- Stateless vs Stateful agents
- Short-term vs Long-term memory
- Persist decisions, not noise

Short-Term Memory (Working / Context Memory)

What it is

- The information inside the **current context window**
- Recent messages, tool outputs, system instructions

Characteristics

- Volatile (gone after the conversation or context reset)
- Token-limited
- Fastest access

Used for

- Multi-step reasoning
- Tool chaining
- “What did I just ask / do?”

Example

Agent remembers that it already asked for *supplier name* earlier in the same flow.

⚠️ **Common pitfall**: People think this is “memory”. It’s not — it’s just context.

Episodic Memory (Conversation / Event Memory)

What it is

- Logged past interactions or “episodes”
- Usually stored as conversation summaries or message chunks

Characteristics

- Medium persistence
- Often summarized or compressed
- Retrieved selectively

Used for

- Remembering past decisions
- Avoiding repeated questions
- Maintaining continuity across sessions

Example

“Last time, this user preferred Vendor A for cloud services.”

Implementation patterns

- Conversation summarization
- Vector DB over past chats
- Session-based memory store

Procedural Memory (Skills / How-To Memory)

What it is

- Knowledge of **how to do thing**
- Tool usage patterns, workflows, playbooks

Characteristics

- Often encoded in prompts or code
- Stable and reusable
- Not conversational

Used for

- "When task X → follow steps A → B → C"
- Tool invocation strategies
- Guardrails

Example

> "To validate an invoice: extract → verify → approve → notify."

Where it lives

- System prompts
- Agent policies
- State machines / graphs

Long-Term Personalized Memory (User / Entity Memory)

What it is

- Stable information about users, orgs, suppliers, projects

Characteristics

- Explicitly persisted
- Carefully scoped (privacy!)
- High business value

Used for

- Personalization
- Reduced friction
- Context-aware decisions

Example

"This supplier is inactive."
"User prefers email approvals."

⚠️ Needs **clear write rules** — agents shouldn't hallucinate memory updates.

Mem0

Mem0 is a **plug-and-play long-term memory layer for agents**.

You give it:

- Conversations
- User interactions
- Events

It gives you:

- Structured, persistent memory
- Automatic relevance filtering
- Safe read/write APIs

👉 Think **“managed memory service for agents”**.

Mem0

What Mem0 is *not*

- ❌ Not a reasoning engine
- ❌ Not a planner
- ❌ Not a graph workflow system

It's **memory infra**, not cognition.

Mem0

What problem it solves

Most agent systems struggle with:

- ❌ remembering *what actually matters*
- ❌ overwriting memory incorrectly
- ❌ hallucinating user preferences
- ❌ messy vector DB hacks

Mem0 solves this by acting as a **memory OS**.

Memory Type	Example
User Memory	“Prefers email approvals”
Entity Memory	“Vendor X is inactive”
Project Memory	“Budget cap is ₹5L”
Conversation Memory	Summarized chat
System Memory	Policies, constraints

Graphiti

Graphiti is a **temporal knowledge-graph–based memory framework** for agents.

It models memory as:

- Nodes (entities, facts, events)
- Edges (relationships)
- Time (when something was true)

👉 Think **“agent memory as a living graph”**.

Graphiti

Instead of storing memory as text chunks or embeddings:

Store reality as facts + relationships over time

Example:

```
(Vendor A) ——[is_active]——▸ (True) @ T1

(Vendor A) ——[is_active]——▸ (False) @ T2
```

Now the agent can reason:

- “What was true then?”
- “What changed?”
- “What conflicts?”

Graphiti

What Graphiti is *not*

- ❌ Not plug-and-play
- ❌ Not trivial to operate
- ❌ Not optimized for quick personalization

It's **powerful**, but **heavy**.

Graphiti

How Graphiti works (conceptually)

```
Event → Entity Extraction
      → Relation Detection
      → Graph Update (with time)
      → Query via graph traversal
```

Agents can ask:

- “What do we know about Supplier X *now*?”
- “What facts support this conclusion?”
- “Who wrote this memory?”

Persistence Architecture

Memory

AgentChat provides a **Memory** protocol that can be extended to provide this functionality. The key methods are query, update_context, add, clear, and close.

- add: add new entries to the memory store
- query: retrieve relevant information from the memory store
- update_context: mutate an agent's internal model_context by adding the retrieved information (used in the **AssistantAgent** class)
- clear: clear all entries from the memory store
- close: clean up any resources used by the memory store

Chat UI Integration

- Streamlit
- Clear agent attribution

Wrap-Up & Next Steps

- Production considerations
- Evaluation & monitoring
- Deployment patterns