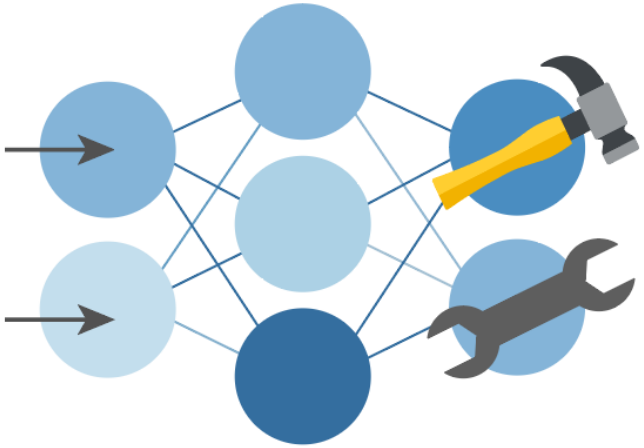
Python + Agents



🛠️ Feb 24: Building your first agent in Python
🧠 Feb 25: Adding context and memory to agents
📊 Feb 26: Monitoring and evaluating agents
🔀 Mar 3: Building your first AI-driven workflows
🕸️ Mar 4: Orchestrating advanced multi-agent workflows
💪 Mar 5: Adding a human-in-the-loop to workflows

🔗 **Register at aka.ms/PythonAgents/series**

Microsoft

Python + Agents

Adding context and memory to agents

aka.ms/pythonagents/slides/contextmemory

Pamela Fox

Python Cloud Advocate

www.pamelafox.org

Today we'll cover...

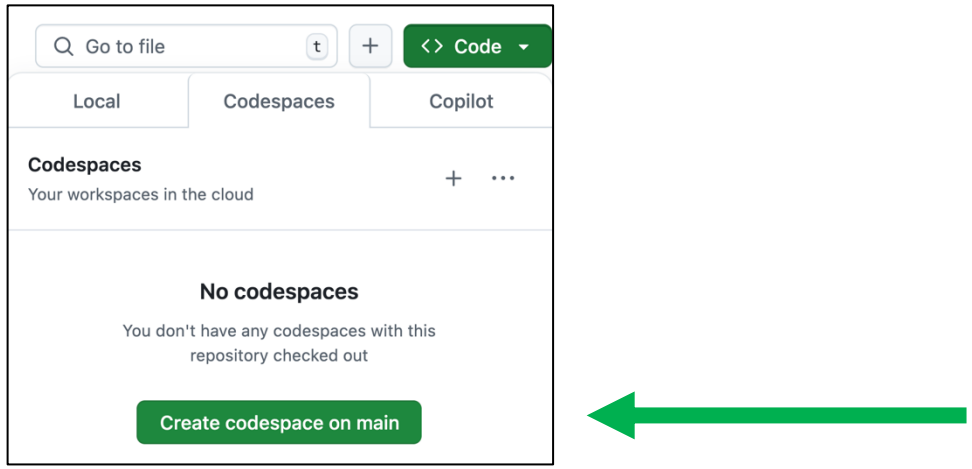
- What is context?
- Knowledge context providers
- What is memory?
- Threads
- Chat history
- Dynamic memory
- Context management

Want to follow along?

1. Open this GitHub repository:

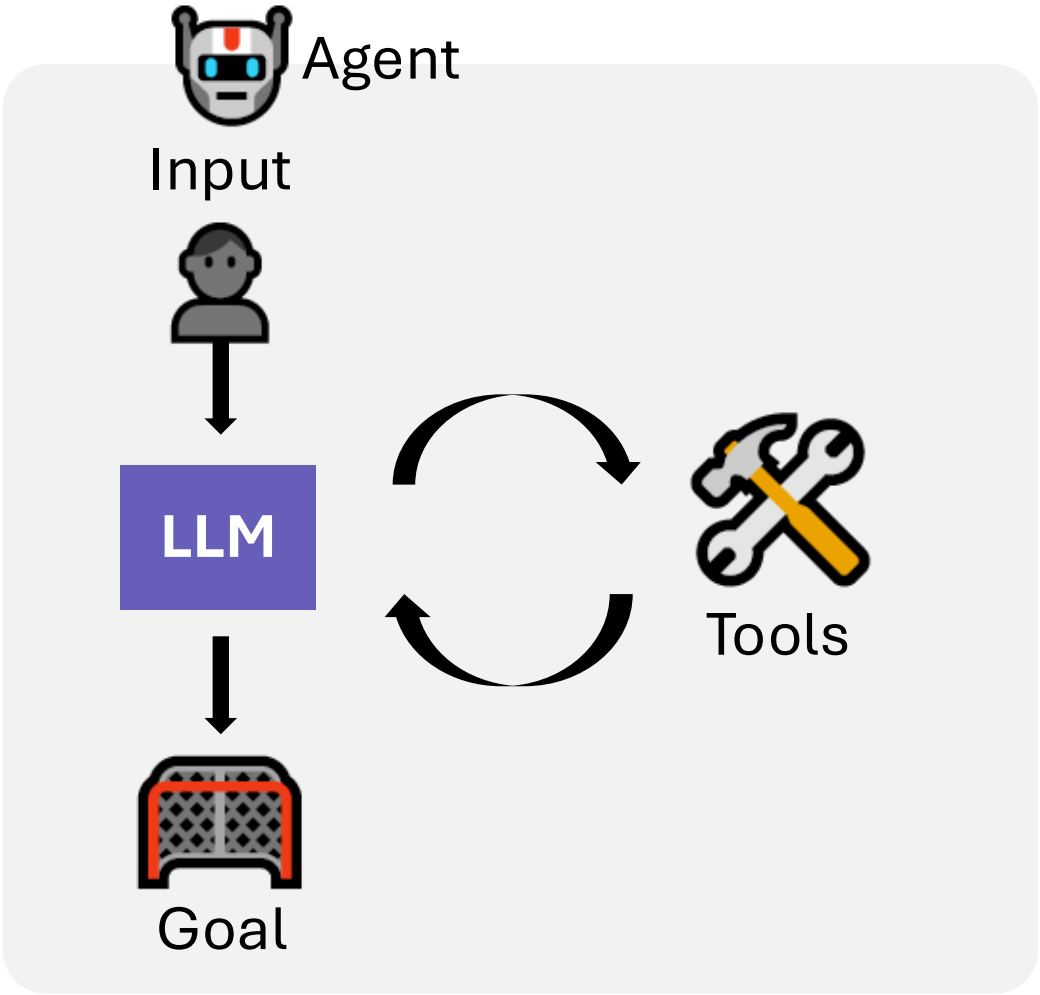
aka.ms/python-agentframework-demos

2. Use "Code" button to create a GitHub Codespace:



3. Wait a few minutes for Codespace to start up ⏱️

Recap: What's an agent?



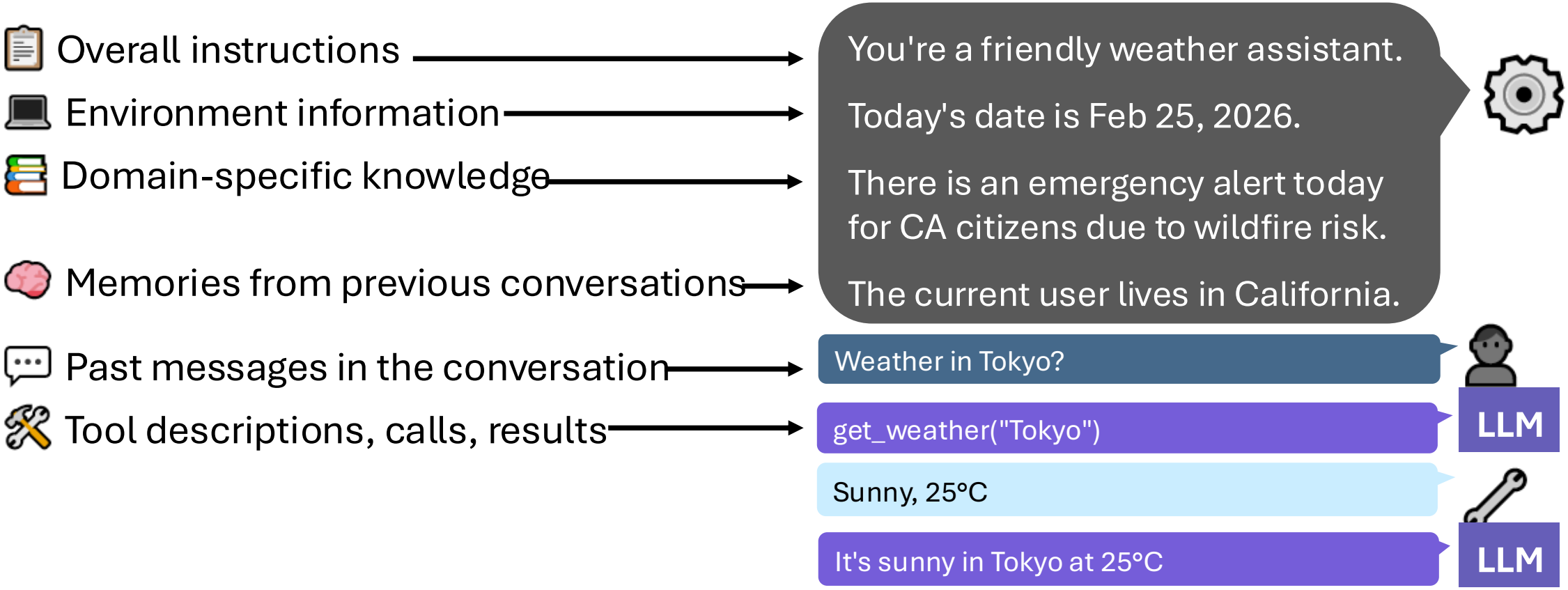
An **AI agent** uses an **LLM** to run **tools** in a loop to achieve a **goal**.

Agents are improved by better **context**:

- Knowledge
- Memory
- Humans

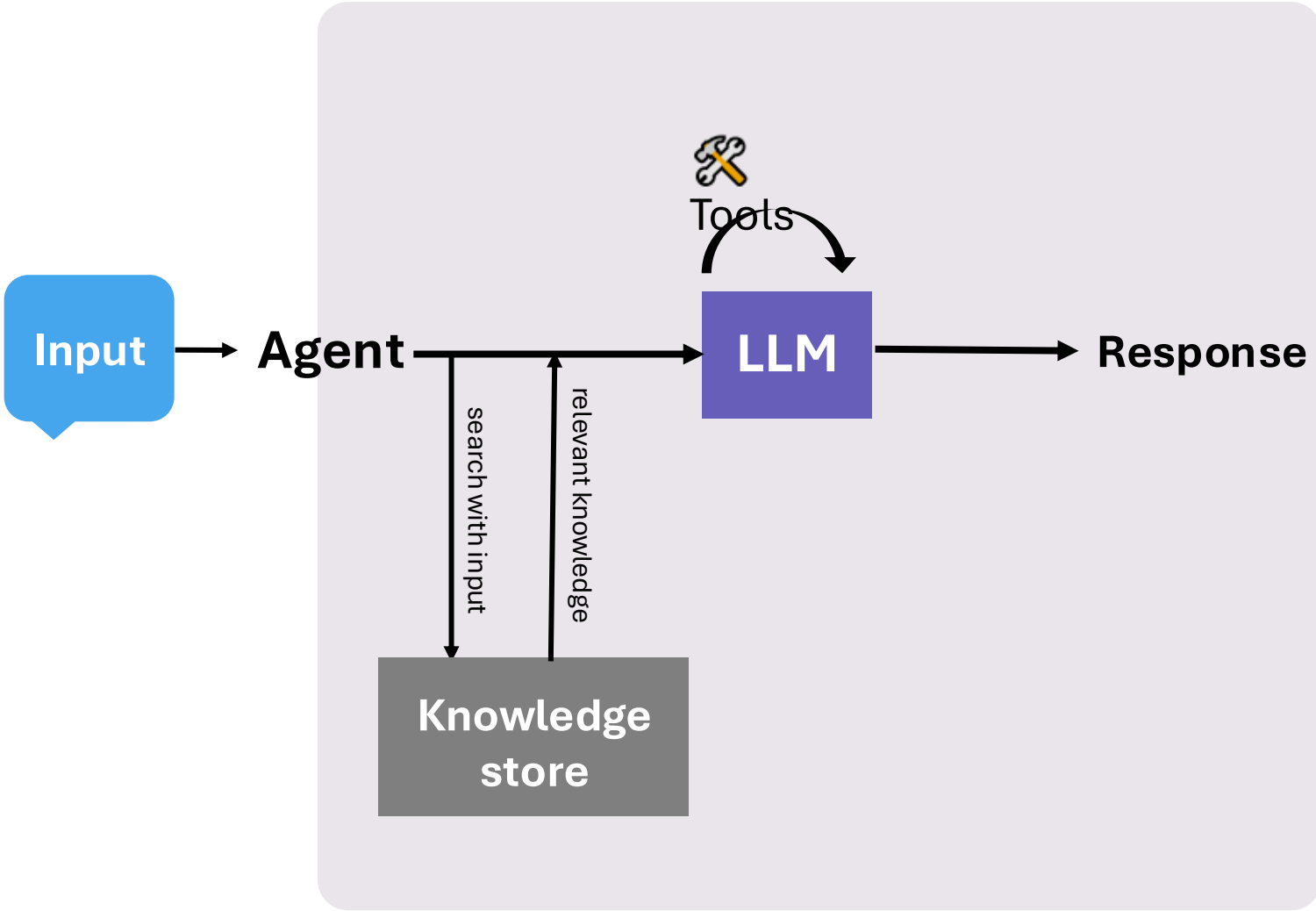
What is context?

The context to an agent is all the inputs that help the agent decide what action it will take next and how it will respond.



Knowledge

Knowledge retrieval



The agent retrieves knowledge *before* asking the LLM to call tools.

Why not just use a tool to retrieve knowledge?

In many scenarios, the agent *always* needs domain-specific knowledge to ground its response, so it's unnecessary to ask an LLM to decide to use the tool.

agent-framework

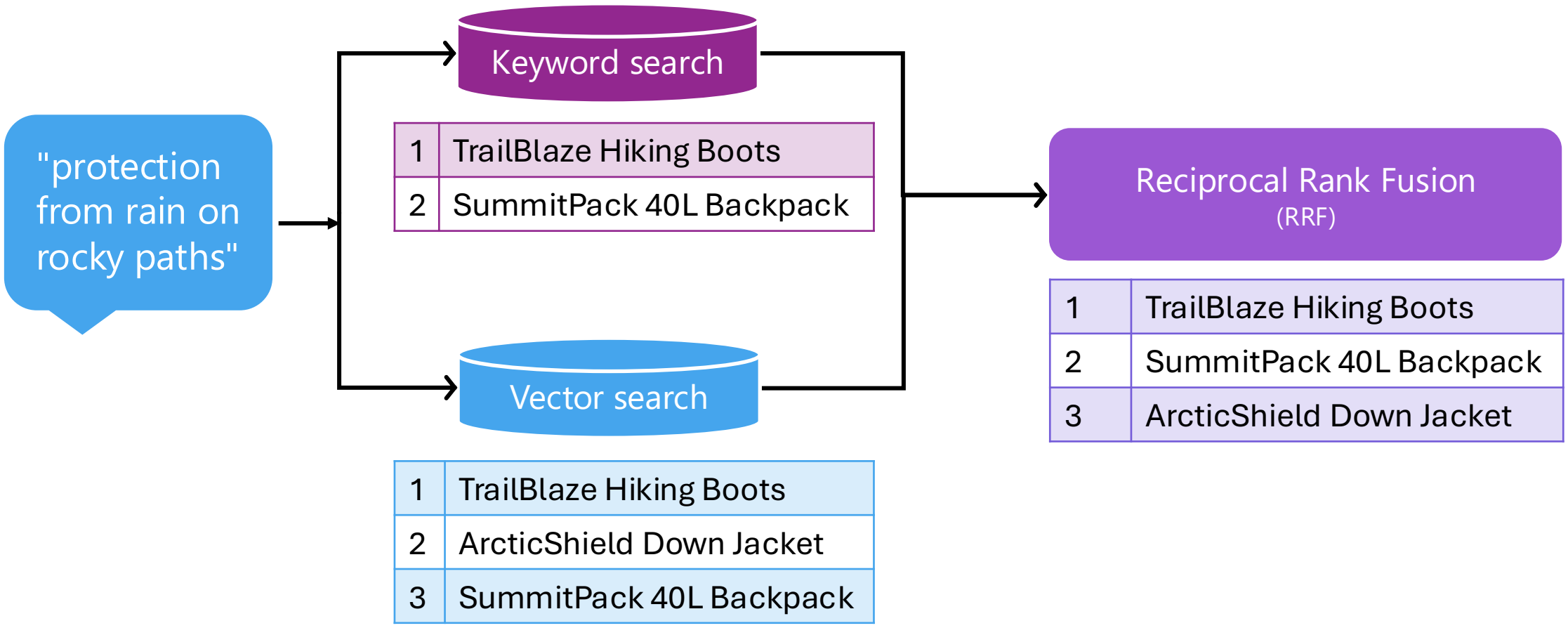
Knowledge retrieval from SQLite database

Create a subclass of `BaseContextProvider` to retrieve data for the given input:

```
class SQLiteKnowledgeProvider(BaseContextProvider):
  async def before_run(self, *, agent, session, context, state):
    user_text = context.input_messages[-1].text
    results = self._search(user_text)
    if results:
      context.extend_messages(self.source_id,
        [Message(role="user", text=self._format(results))])

agent = Agent(client=client,
  instructions=("You are an outdoor-gear shopping assistant."),
  context_providers=[knowledge_provider])

response = await agent.run("Do you have anything for kayaking?")
```

Full example: agent_knowledge_sqlite.py

Improve retrieval with hybrid search



agent-framework

Knowledge retrieval from PostgreSQL

PostgreSQL can do a hybrid search with an embedding model and SQL query:

```
class SQLiteKnowledgeProvider(BaseContextProvider):
  async def before_run(self, *, agent, session, context, state):
    user_text = context.input_messages[-1].text
    results = self._search(user_text)
    if results:
      context.extend_messages(self.source_id,
        [Message(role="user", text=self._format(results))])

agent = Agent(client=client,
  instructions=("You are an outdoor-gear shopping assistant."),
  context_providers=[knowledge_provider])

response = await agent.run("Do you have anything for kayaking?")
```

Full example: agent_knowledge_sqlite.py

More ways to improve knowledge retrieval

- Before the search step:
  - Query rewriting ("what do you have for kayaking?" → "kayaking gear")
  - Multiple knowledge sources with source selection ("kayaking" → "outdoor_products_db")
- After the results return:
  - Re-ranking model (["paddle", "boat"] → ["boat", "paddle"])
  - Reflection and iterative search ("results not good enough?" → "search again!")

You can also implement many techniques implementing retrieval as a tool!

Learn more:

- Python+AI RAG stream: aka.ms/pythonai/resources
- Ignite talk on Agentic RAG with Azure AI Search: aka.ms/agenticrag/ignite/video

Memory

Memory

Can LLM remember past messages?

Can app remember past conversations?

Can LLM remember facts from past conversations?

Threads

- Within conversation, just messages and tool calls
- Can be stored in-memory OR persisted to a database to restore conversation later
- May require truncation or summarization

Chat history

- Must be stored in a persistent database
- Across conversations, typically tied to user ID

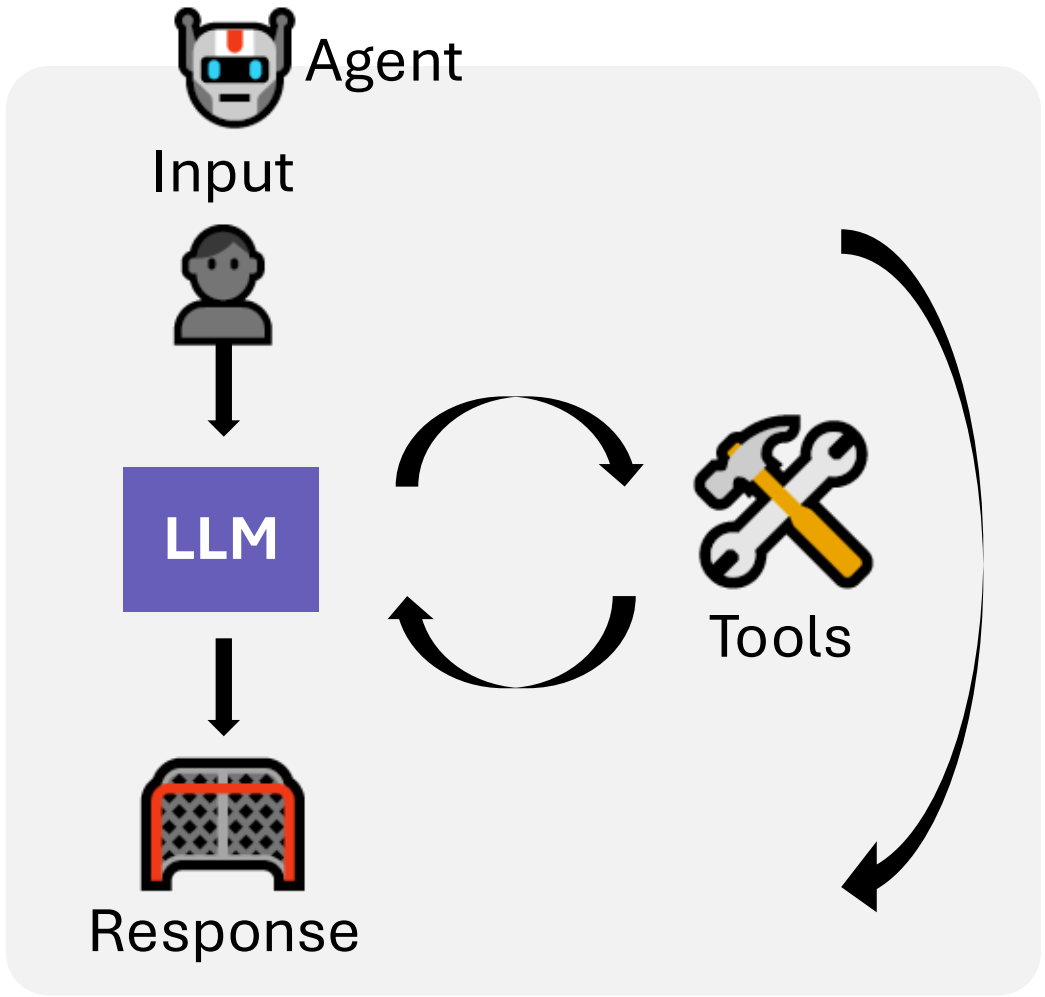
Dynamic memory

- Must be stored in a persistent database
- Retrieve memories, optionally using search to find most relevant

https://learn.microsoft.com/agent-framework/user-guide/agents/agent-memory

Conversation threads

Conversation threads



The agent should remember past messages:

agent-framework

Conversation thread (In-memory)

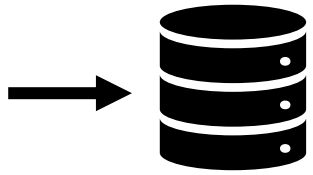
By default, the conversation messages are stored in the thread in memory:

```
from agent_framework import ChatAgent
from agent_framework.openai import OpenAIChatClient

agent = ChatAgent(
  chat_client=OpenAIChatClient(),
  instructions="You are a helpful assistant.")

thread = agent.get_new_thread()

response = await agent.run("Hello, my name is Alice", thread=thread)
```

Full example: agent_thread.py

Persistent chat history

Persistent chat history



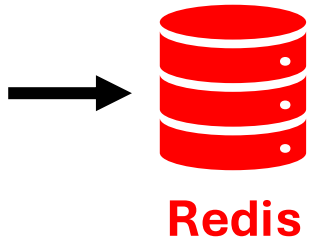
Database

id	messages
thread123	system: You're a weather assistant. user: Weather in Tokyo? ...
thread456	system: You're a weather assistant. user: Will it rain in CA? ....

Implementation details:

- Will entire threads fit in one row? If not, how to handle?
- How will threads be linked to users?
- Will the threads be searchable?

Persistent chat history in Redis



key	value [LIST]
thread123	{"role": "system", "content": "You're a weather assistant."} {"role": "user", "content": "Weather in Tokyo?"} {"role": "assistant", "tool_call": "get_weather"}

Implementation details:

- Keys are thread IDs
- The values are lists of messages that can be truncated to the most recent N

agent-framework

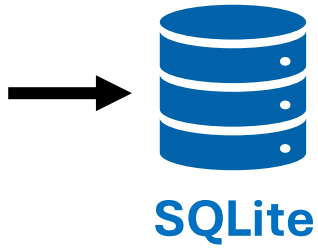
Persistent chat history

Use a built-in `ChatMessageStore` (like `RedisChatMessageStore`) or write a subclass:

```
from agent_framework.redis import RedisChatMessageStore

thread_id = str(uuid.uuid4())

store = RedisChatMessageStore(
    redis_url="redis://localhost:6379",
    thread_id=thread_id)

agent = ChatAgent(
  chat_client=OpenAIChatClient(),
  instructions="You are a helpful weather agent.",
  tools=[get_weather])
```

Full example: agent_thread_redis.py

Persistent chat history in SQLite



id	thread_id	message
1	thread123	{"role": "system", "content": "You're a weather assistant."}
2	thread123	{"role": "user", "content": "Weather in Tokyo?"}
3	thread123	{"role": "assistant", "tool_call": "get_weather"}
4	thread456	{"role": "system", "content": "You're a weather assistant."}

Implementation details:

- Rows are single messages, retrieved based off incrementing ID order
- No limits to # of messages in thread

agent-framework

Persistent chat history with custom storage

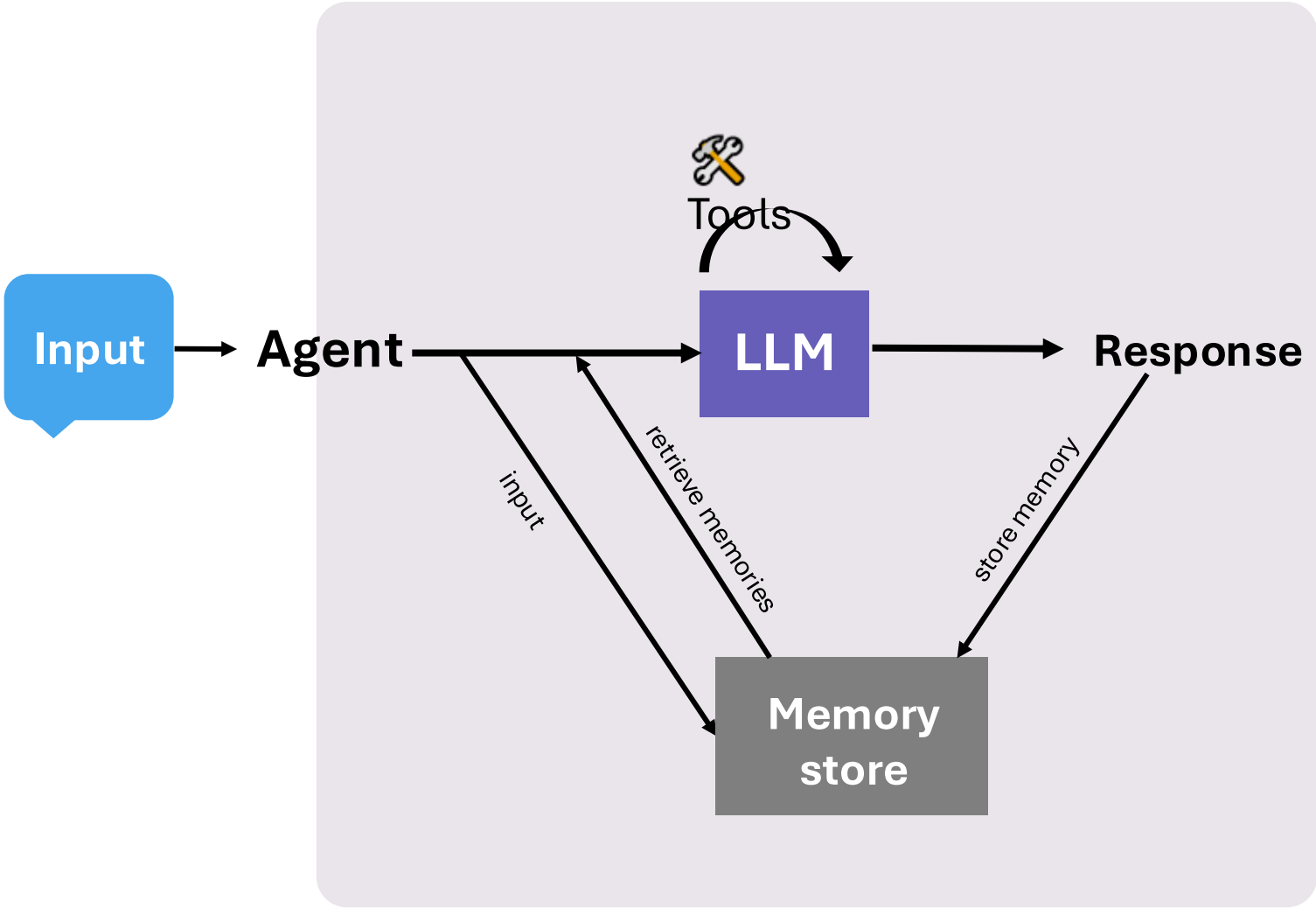
Write a subclass that implements `ChatMessageStoreProtocol`:

```python
class SQLiteChatMessageStore(ChatMessageStoreProtocol):
  def __init__(self, db_path: str, thread_id: str | None = None):

  async def add_messages(self, messages: Sequence[ChatMessage]):

  async def list_messages(self) -> list[ChatMessage]:

  async def serialize(self, **kwargs: Any) -> dict[str, Any]:

  @classmethod
  async def deserialize(cls, state: MutableMapping[str, Any]) -> SQLiteChatMessageStore:

  async def update_from_state(self, state: MutableMapping[str, Any]) -> None:

  def close(self) -> None:
```

Full example: agent_thread_sqlite.py

Dynamic memory

Dynamic memory



Implementation matters!

Retrieving memories:

- All or only most relevant?

Storing memories:

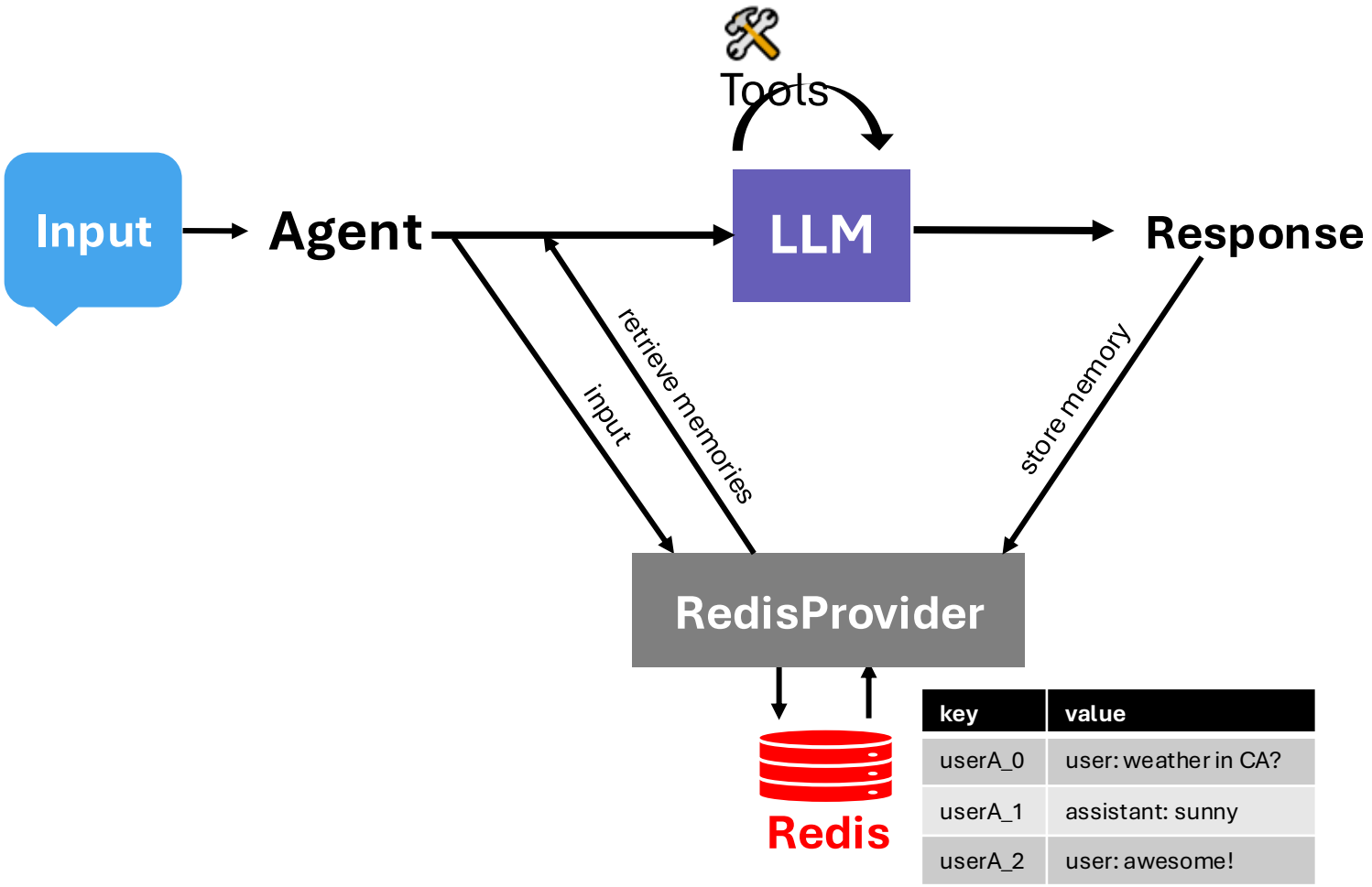
- Whole message?
- LLM-synthesized memory?

Forgetting memories:

- Oldest? Obsolete?

agent-framework

Dynamic memory with Redis



`RedisProvider:`

Retrieving memories:

- Searches with full-text, vector, or hybrid search

Storing memories:

- Entire user and assistant messages (no tool calls)

Forgetting memories:

- No mechanism- memories remain forever

agent-framework

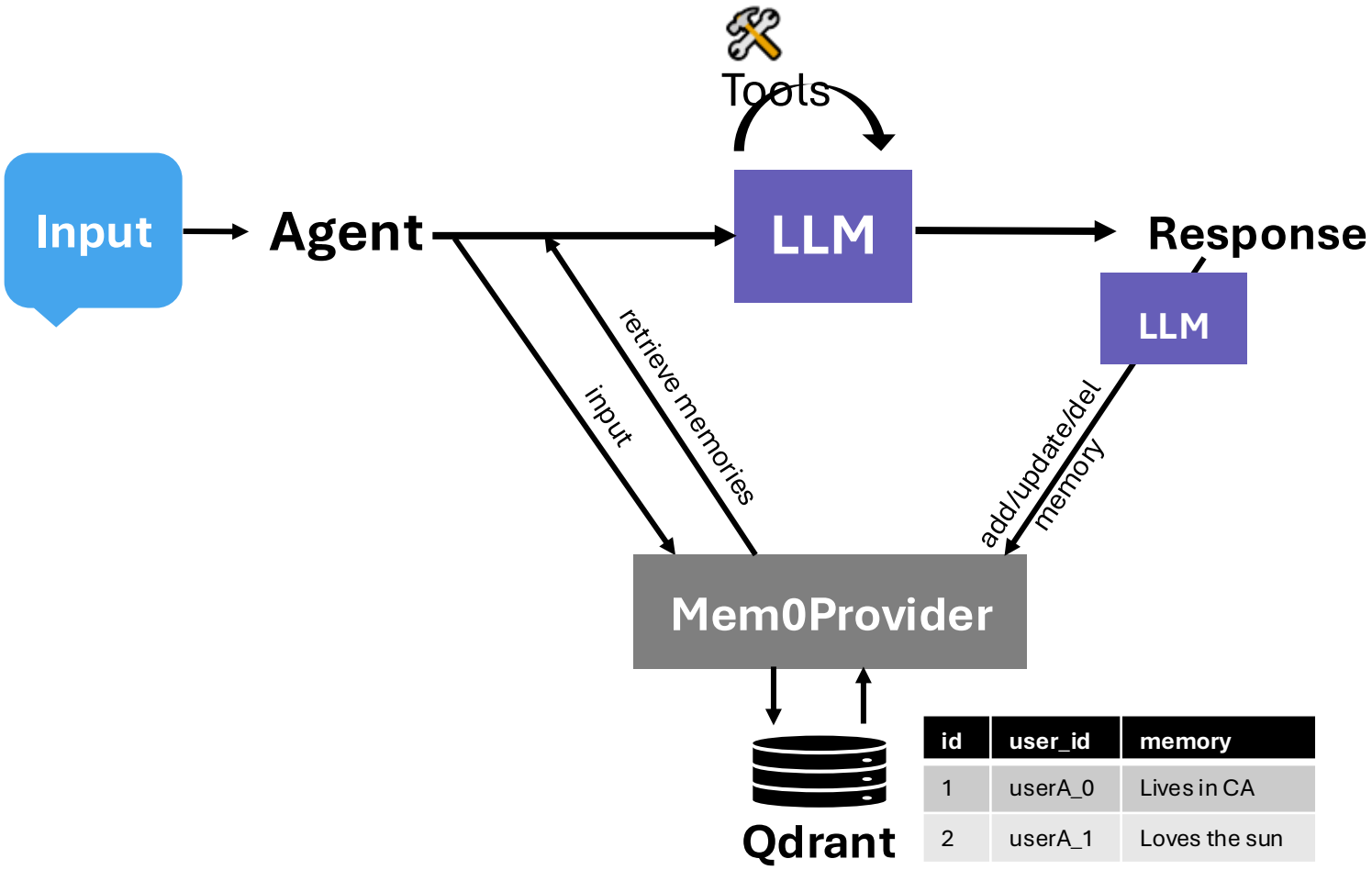
Dynamic memory with Redis

```python
from agent_framework.redis import RedisProvider

user_id = str(uuid.uuid4())
memory_provider = RedisProvider(
  redis_url=REDIS_URL,
  index_name="agent_memory_demo",
  overwrite_index=True,
  prefix="memory_demo",
  application_id="weather_app",
  agent_id="weather_agent",
  user_id=user_id,
)
agent = ChatAgent(
  chat_client=OpenAIChatClient(),
  instructions="You are a helpful weather assistant.",
  context_providers=memory_provider,
  tools=[get_weather])
```

Full example: agent_memory_redis.py

agent-framework

Dynamic memory with Mem0



`Mem0Provider:`

Retrieving memories:

- Searches the associated database (Qdrant default)

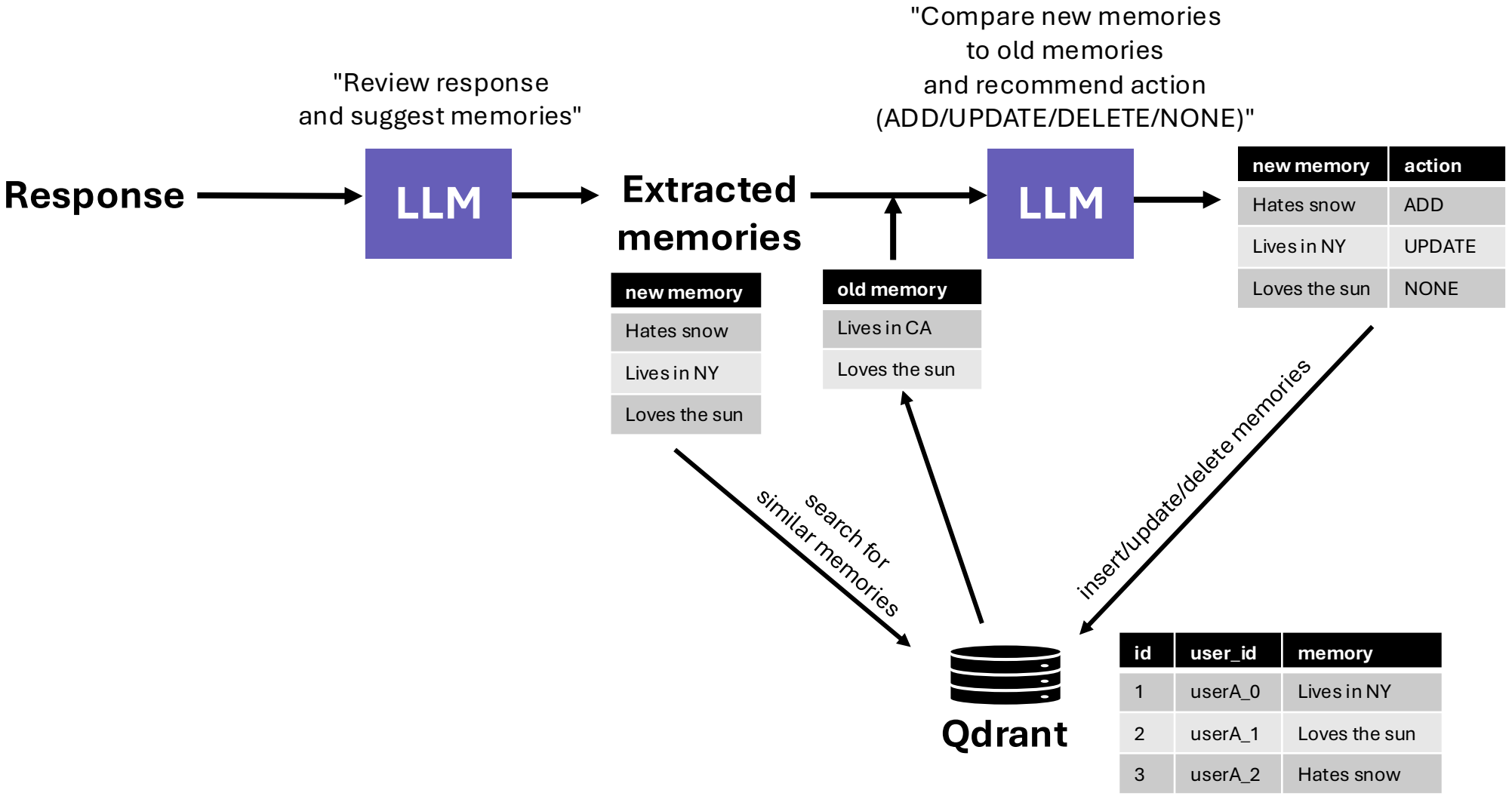
Storing memories:

- LLM suggests new memories

Forgetting memories:

- LLM suggests memory removal or updating

Mem0Provider: Memory update process



agent-framework

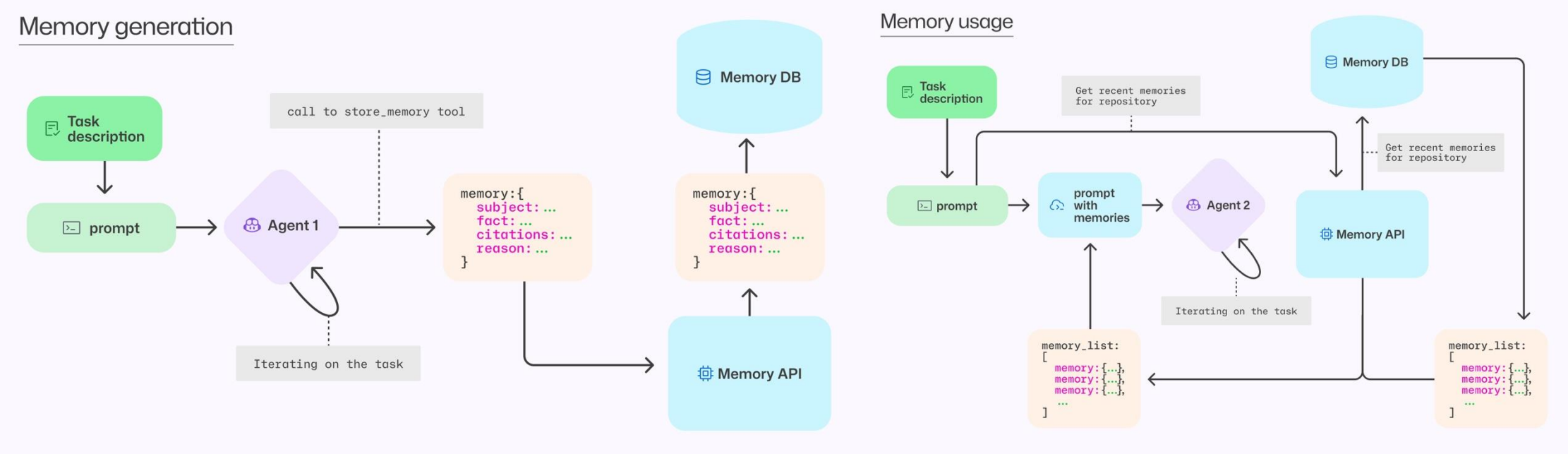
Dynamic memory with Mem0

Use a built-in memory provider (like `Mem0Provider`) or write your own context provider:

```
from agent_framework.mem0 import Mem0Provider
from mem0 import AsyncMemory

user_id = str(uuid.uuid4())

mem0_client = await AsyncMemory.from_config(mem0_config)

provider = Mem0Provider(user_id=user_id, mem0_client=mem0_client)

agent = ChatAgent(
  chat_client=OpenAIChatClient(),
  instructions="You are a helpful weather assistant.",
  context_providers=memory_provider,
  tools=[get_weather])
```

Full example: agent_thread_memory.py

Memory in production: GitHub Copilot



All memories include **citations**, which are verified for accuracy *before* memories are used. A memory which can no longer be verified is removed.

Learn more:
https://github.blog/ai-and-ml/github-copilot/building-an-agentic-memory-system-for-github-copilot/

Context management

Context window

The **context window** of an LLM is the amount of input tokens that an LLM can process at a given time.

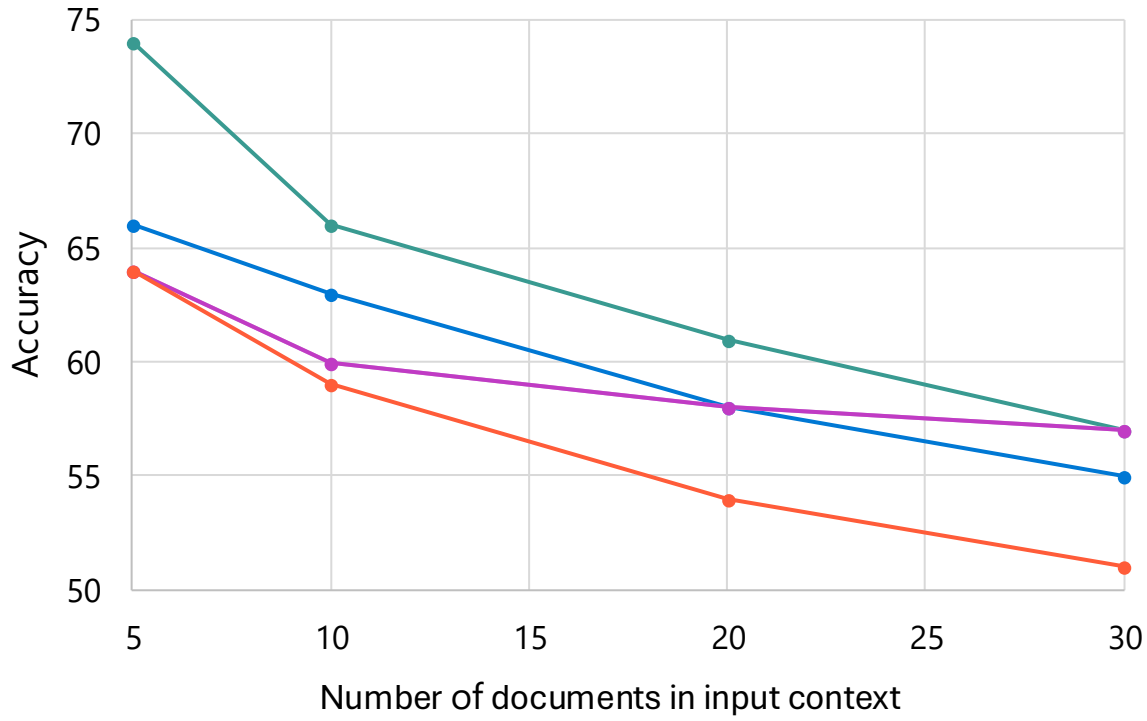
Context windows have been steadily increasing over time:

Model	Release date	Context window (tokens)
GPT-3	2020	2K
GPT-3.5 Turbo	Feb 2024	4K-16K
GPT-4	Mar 2023	8K-32K
GPT-4 Turbo	Dec 2023	128K
Claude 3 (Sonnet/Opus)	Feb 2025	200K
Claude Opus 4.6	Feb 2026	1M

Do we still need to worry about context size?

Absolutely!

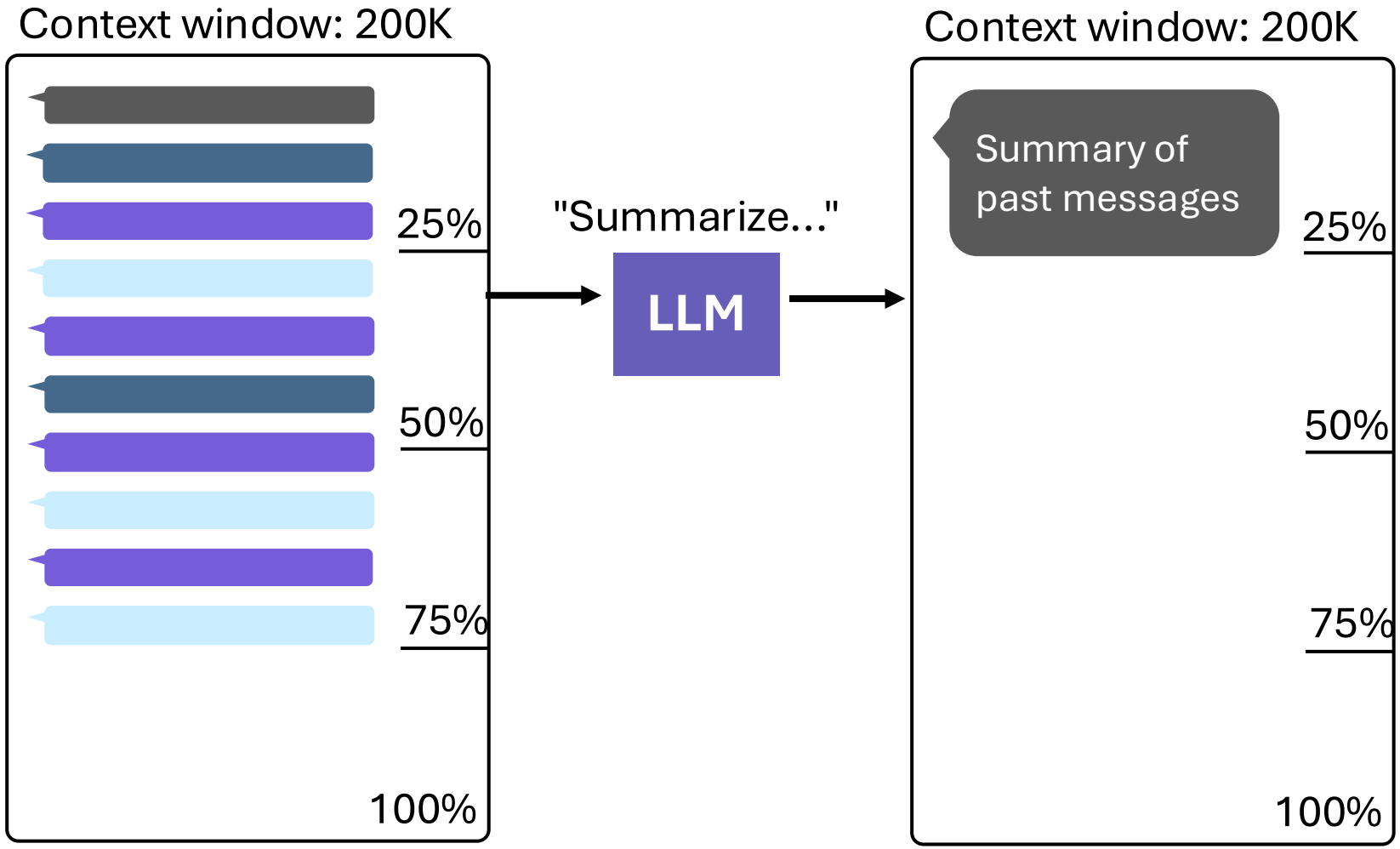
1. The context windows are finite (128K-2M)
2. When an LLM receives too much information, it can get easily distracted by irrelevant details.
3. The more tokens you send, the higher the cost, the slower the response.



Source: Lost in the Middle

See also: Context rot research from Chroma

Context compaction with summarization



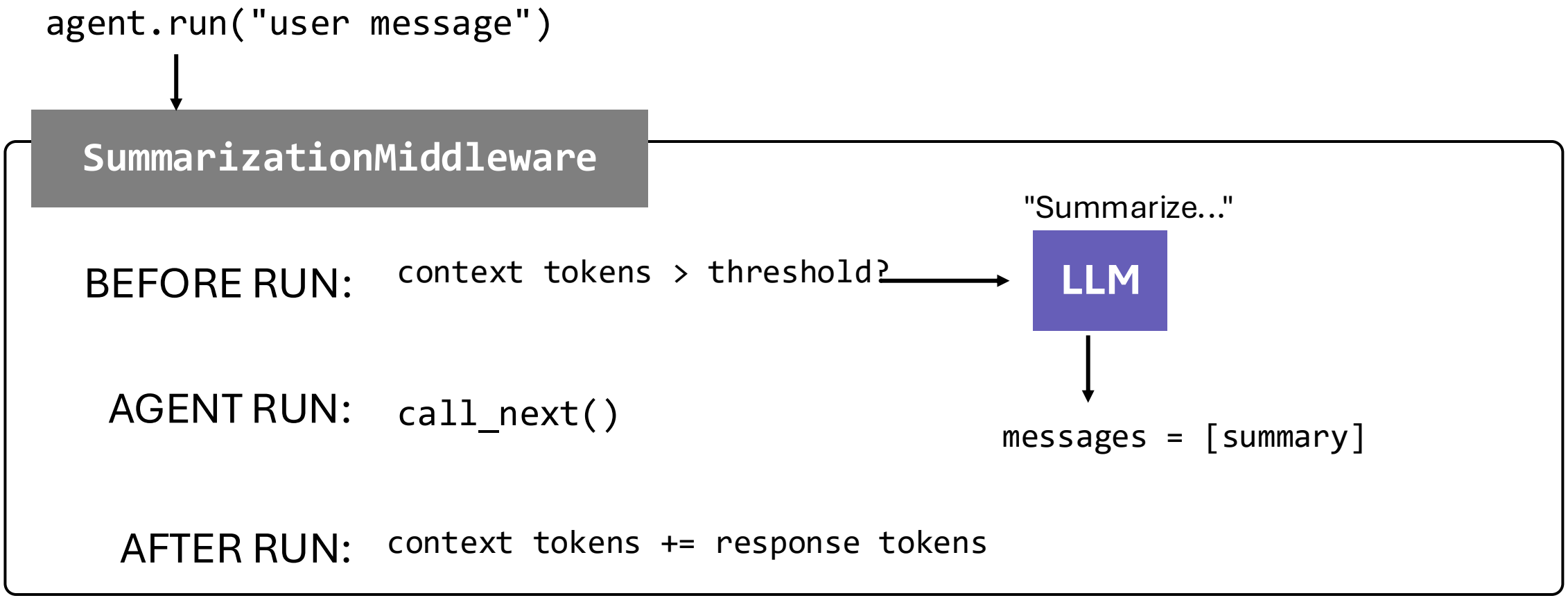
Implementation details:

- How often to summarize?
- What's most important to summarize?
- Do you store original messages for later?

agent-framework

Summarization middleware

We can subclass `AgentMiddleware` to summarize messages and replace them.



agent-framework

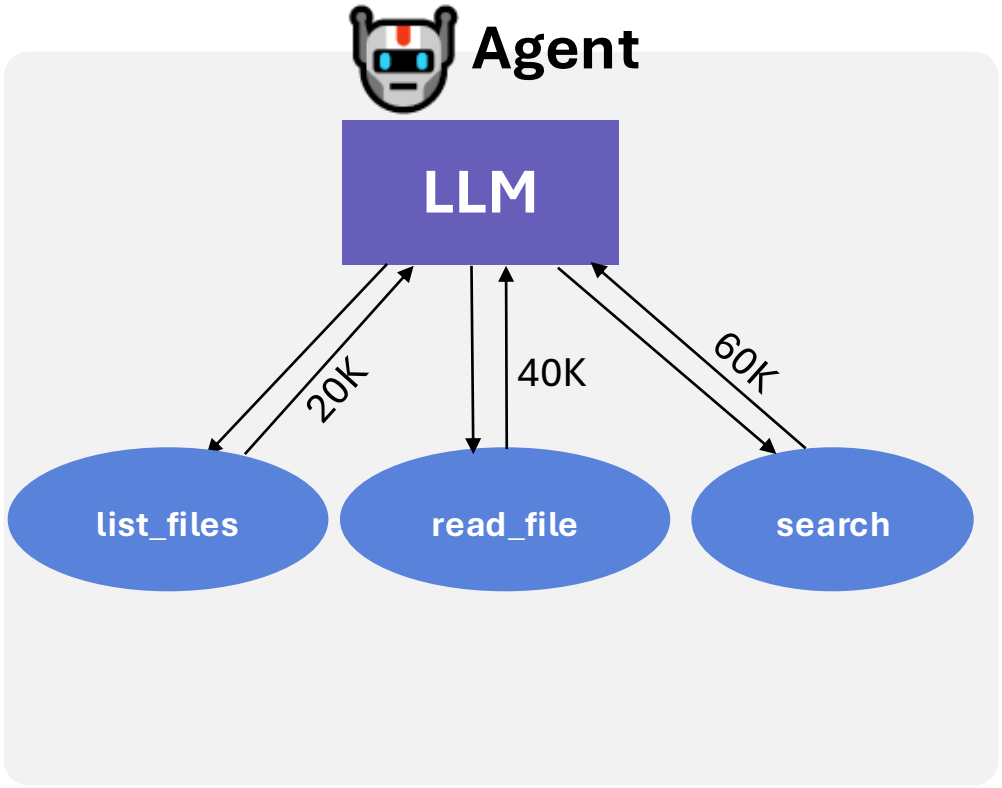
Summarization middleware

```python
class SummarizationMiddleware(AgentMiddleware):

  async def process(self, context, call_next):
    # Before the agent runs
    if context.session and self.context_tokens > self.token_threshold:
      history = context.session.state["memory"]["messages"]
      summary = await self._summarize(history) # LLM call
      context.session.state["memory"]["messages"] = [
       [Message(role="assistant", text=f"[Summary]\n{summary}")]]
      self.context_tokens = 0

    await call_next() # Runs the standard agent loop

    # After the agent runs
    if context.result and context.result.usage_details:
      self.context_tokens += context.result.usage_details["total_token_count"]
```

Full example: agent_summarization.py

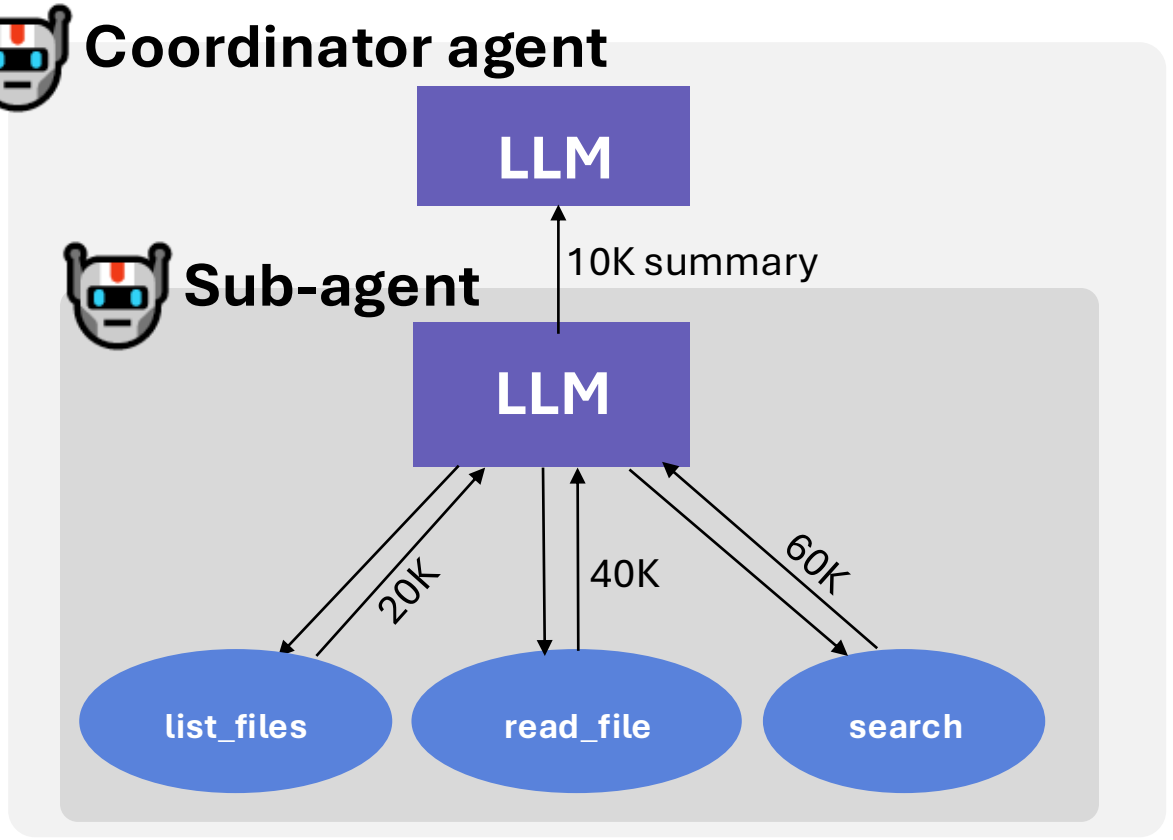
Context reduction with subagents

A single agent is risky...



😳 Context can explode due to large tool call results growing over time

Better: **Coordinator agent** calls **sub-agents**



Sub-agent returns a *summary* of tool call results.

agent-framework

Coordinator agent with sub-agents

```python
# Sub-agent: researches codebase in its own isolated context
research_agent = Agent(client=client,
  instructions="Read and search files, return a concise summary under 200 words.",
  tools=[list_project_files, read_project_file, search_project_files],
)

@tool
async def research_codebase(question: str) -> str:
  """Delegate research to the sub-agent. Returns only a summary."""
  response = await research_agent.run(question)
  return response.text

# Coordinator agent: only sees summaries, never raw file contents
coordinator = Agent(client=client,
  instructions="Answer questions about code. Use research_codebase to investigate.",
  tools=[research_codebase])
```

Full example: agent_with_subagent.py

Context reduction with subagents: Token usage

The coordinator's context window stays small, despite sub-agent using more tokens. In multi-turn conversations, this difference would compound with each turn.

	Single agent	vs.	Coordinator agent	Sub-agent
Input tokens	6,714		623	8,494
Output tokens	1,598		514	580
Total tokens	8,312		1,137	9,074

The final answer returned was a high-quality answer in *both* scenarios.

When to use sub-agents?

✅ **Use sub-agents when...**

- Tools return large outputs
- Task requires multiple chained tool calls
- Main agent has multiple responsibilities
- Long-running, multi-turn conversations
- Sub-tasks need specialized instructions

❌ **Skip sub-agents when...**

- Tool responses are small and simple
- Single tool call gets the answer
- Main agent needs to see raw tool output
- Low latency is critical (extra LLM round-trip)

Next steps

Register:
https://aka.ms/PythonAgents/series

Watch past recordings:
aka.ms/pythonagents/resources

Join office hours after each session in Discord:
aka.ms/pythonai/oh

🛠️ Feb 24: Building your first agent in Python
🧠 Feb 25: Adding context and memory to agents
📊 Feb 26: Monitoring and evaluating agents
🔀 Mar 3: Building your first AI-driven workflows
🕸️ Mar 4: Orchestrating advanced multi-agent workflows
💪 Mar 5: Adding a human-in-the-loop to workflows