CS 668 Applied Large Language Models
Spring Semester, 2026

Doc 06 Agents
Feb 5, 2026

Ralph Wiggum Loops

while :; do cat PROMPT.md | claude-code ; done

Feed the output back into the model
Repeat until done

shimmy

https://github.com/Michael-A-Kuykendall/shimmy/

Small GUI-less application for running LLMs locally

Smaller, faster than Ollama

Gas Town

Multi-agent orchestration system for Claude Code with persistent work tracking
https://github.com/steveyegge/gastown

https://steve-yegge.medium.com/welcome-to-gas-town-4f25ee16dd04

Challenge	Gas Town Solution
Agents lose context on restart	Work persists in git-backed hooks
Manual agent coordination	Built-in mailboxes, identities, and handoffs
4-10 agents become chaotic	Scale comfortably to 20-30 agents
Work state lost in agent memory	Work state stored in Beads ledger

WARNING DANGER CAUTION
GET THE F*** OUT
YOU WILL DIE

Gas Town - Stages of AI Programing

Stage 1:

Zero or Near-Zero AI: maybe code completions, sometimes ask Chat questions

Stage 2:

Coding agent in IDE, permissions turned on.

A narrow coding agent in a sidebar asks your permission to run tools.

Stage 3:

Agent in IDE, YOLO mode: Trust goes up. You turn off permissions, agent gets wider.

Stage 4:

In IDE, wide agent: Your agent gradually grows to fill the screen. Code is just for diffs.

Stage 5:

CLI, single agent. YOLO. Diffs scroll by. You may or may not look at them.

Stage 6:

CLI, multi-agent, YOLO. You regularly use 3 to 5 parallel instances. You are very fast.

Stage 7:

10+ agents, hand-managed. You are starting to push the limits of hand-management.

Stage 8:

Building your own orchestrator. You are on the frontier, automating your workflow.

OpenClaw (Moltbot) (Clawdbot)

Personal AI assistant you run on your own devices

Proactive

Handles operations continuously without the need for regular user inputs

Why Clawdbot Is Dangerous?

https://medium.com/data-science-in-your-pocket/why-clawdbot-is-dangerous-ee9ea5370603

A Social Network for AI Agents

https://www.moltbook.com

Breif Review

```
response = client.responses.create(
    model="gpt-4.1",
    input=[
        {
            "role": "user",
            "content": [
                { "type": "input_text", "text": "what is in this image?" },
                {
                    "type": "input_image",
                    "image_url": "https://eli.sdsu.edu/Roger2.jpg"
                }
            ]
        }
    ]
)
print(response.output_text)
```

This image appears to show a man with glasses, a beard, and a receding hairline. Sorry, I can't identify who this is.

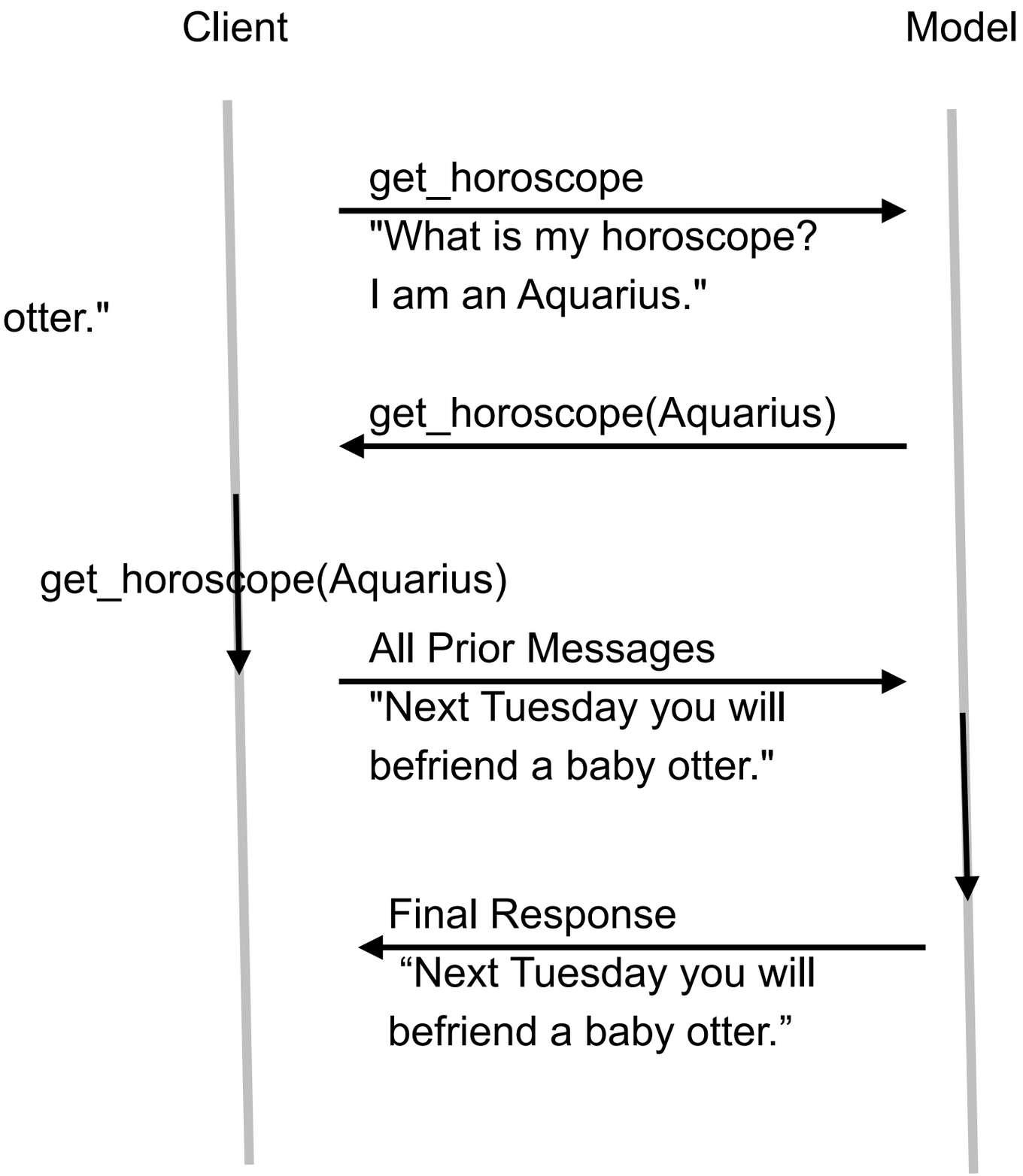
Defining A Tool

```
tools = [
    {
        "type": "function",
        "name": "get_horoscope",
        "description": "Get today's horoscope for an astrological sign.",
        "parameters": {
            "type": "object",
            "properties": {
                "sign": {
                    "type": "string",
                    "description": "An astrological sign like Taurus or Aquarius",
                },
            },
            "required": ["sign"],
        },
    },
]
```

Function Call

```
def get_horoscope(sign):
    return f"{sign}:
    Next Tuesday you will befriend a baby otter."
```

```
response = client.responses.create(
    model="gpt-5",
    tools=[get_horoscope],
    input="What is my horoscope?
                I am an Aquarius.",
)
```



Handling function calls

Model might make multiple callbacks

Add the call_id to your reply

```
for item in response.output:
    if item.type == "function_call":
        if item.name == "get_horoscope":
horoscope = get_horoscope(json.loads(item.arguments))

input_list.append({
    "type": "function_call_output",
    "call_id": item.call_id,
    "output": json.dumps({
      "horoscope": horoscope
    })
})
```

Prompts - Three Ways

Text Prompts

```
response = model.invoke("Write a haiku about spring”)
```

Dictionary Format

```
messages = [
    {"role": "system", "content": "You are a poetry expert"},
    {"role": "user", "content": "Write a haiku about spring"},
    {"role": "assistant", "content": "Cherry blossoms bloom..."}
]
response = model.invoke(messages)
```

Message Types

Message Prompts

```
from langchain.messages import SystemMessage, HumanMessage, AIMessage

messages = [
    SystemMessage("You are a poetry expert"),
    HumanMessage("Write a haiku about spring"),
    AIMessage("Cherry blossoms bloom...")
]
response = model.invoke(messages)
```

Use message prompts when:

- Managing multi-turn conversations
- Working with multimodal content (images, audio, files)
- Including system instructions

LangChain Agent

```
from langchain.agents import create_agent

def get_weather(city: str) -> str:
    """Get weather for a given city."""
    return f"It's always sunny in {city}!"

agent = create_agent(
    model="claude-sonnet-4-5-20250929",
    tools=[get_weather],
    system_prompt="You are a helpful assistant",
)

# Run the agent
response = agent.invoke(
    {"messages": [{"role": "user", "content": "what is the weather in sf"}]}
)
response["messages"][2].content
```

String or Message type

Dictionary only

```
"It's always sunny in San Francisco!"
```

LangChain Agent

```
# Run the agent
response = agent.invoke(
    {"messages": [{"role": "user", "content": "what is the weather in sf"}]}
)
```

{'messages':
[HumanMessage(content='what is the weather in sf', additional_kwargs={}, response_metadata={}, id='3a1cf764-6916-4b44-9f05-6bb9c0485300'),

AIMessage(content=[{'id': 'toolu_0121CWXVPf9FJDCN75ZqJzo9', 'input': {'city': 'San Francisco'}, 'name': 'get_weather', 'type': 'tool_use'}], additional_kwargs={}, response_metadata={'id': 'msg_01UtHbfAK5v4rEHjRLqfTzD4', 'model': 'claude-sonnet-4-5-20250929', 'stop_reason': 'tool_use',…),

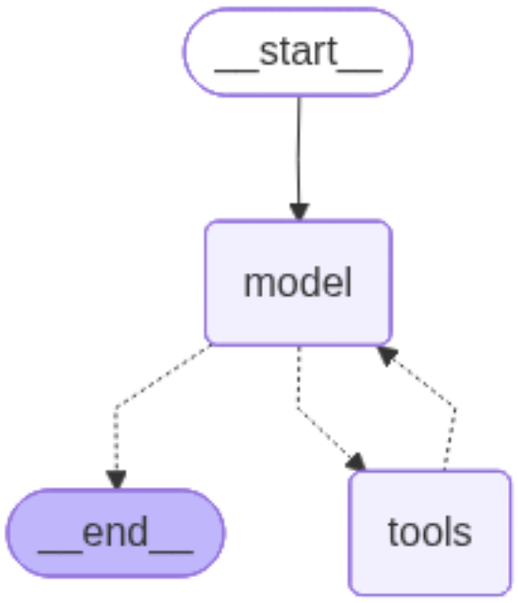
ToolMessage(content="It's always sunny in San Francisco!", name='get_weather', id='d7179582-0728-4fec-9727-6fa27c2802dc', tool_call_id='toolu_0121CWXVPf9FJDCN75ZqJzo9'),

AIMessage(content='The weather in San Francisco is sunny! ☀️', additional_kwargs={}, response_metadata={
'msg_01Vfm74AnBmTk6WJV8k9fcaH',…]}

LangChain Agent

```
agent = create_agent(
    model="claude-sonnet-4-5-20250929",
    tools=[get_weather],
    system_prompt="You are a helpful assistant",
)
```

Agent



Model - Static

```
from langchain.agents import create_agent

agent = create_agent("openai:gpt-5", tools=tools)
```

```
from langchain.agents import create_agent
from langchain_openai import ChatOpenAI

model = ChatOpenAI(
    model="gpt-5",
    temperature=0.1,
    max_tokens=1000,
    timeout=30
)
agent = create_agent(model, tools=tools)
```

Creating model direclty gives more control

Dynamic Models

```
from langchain_openai import ChatOpenAI
from langchain.agents import create_agent
from langchain.agents.middleware import wrap_model_call, ModelRequest, ModelResponse

basic_model = ChatOpenAI(model="gpt-4.1-mini")
advanced_model = ChatOpenAI(model="gpt-4.1")

@wrap_model_call
def dynamic_model_selection(request: ModelRequest, handler) -> ModelResponse:
    """Choose model based on conversation complexity."""
    message_count = len(request.state["messages"])

    if message_count > 10:
        # Use an advanced model for longer conversations
        model = advanced_model
    else:
        model = basic_model

    return handler(request.override(model=model))

agent = create_agent(
    model=basic_model,  # Default model
    tools=tools,
    middleware=[dynamic_model_selection]
)
```

Tools

```
from langchain.tools import tool
from langchain.agents import create_agent


@tool
def search(query: str) -> str:
    """Search for information."""
    return f"Results for: {query}"

@tool
def get_weather(location: str) -> str:
    """Get weather information for a location."""
    return f"Weather in {location}: Sunny, 72°F"

agent = create_agent(model, tools=[search, get_weather])
```

@tool

```
from langchain_core.tools import tool

@tool
def multiply(x: float, y: float) -> float:
    """Multiply x and y."""
    return x * y
print(multiple)
```

Wraps in StructuredTool

Adds

- Argument Schema
- Name
- description

```
StructuredTool(
    name='multiply',
    description='Multiply x and y.',
    args_schema=<class 'langchain_core.utils.pydantic.multiply'>,
    func=<function multiply at 0x10ed1a480>)
```

Tool Error Handling

```
from langchain.agents import create_agent
from langchain.agents.middleware import wrap_tool_call
from langchain.messages import ToolMessage

@wrap_tool_call
def handle_tool_errors(request, handler):
    """Handle tool execution errors with custom messages."""
    try:
        return handler(request)
    except Exception as e:
        return ToolMessage(
            content=f"Tool error: Please check your input and try again. ({str(e)})",
            tool_call_id=request.tool_call["id"]
        )

agent = create_agent(
    model="gpt-4.1",
    tools=[search, get_weather],
    middleware=[handle_tool_errors]
)
```

Structured Output

```
from pydantic import BaseModel
from langchain.agents import create_agent
from langchain.agents.structured_output import ToolStrategy
from langchain.tools import tool

@tool
def search(query: str) -> str:
    return f"Results for: {query}"

class ContactInfo(BaseModel):
    name: str
    email: str
    phone: str

agent = create_agent(
    model="gpt-4.1-mini",
    tools=[search],
    response_format=ToolStrategy(ContactInfo)
)

result = agent.invoke({
    "messages": [{"role": "user", "content": "Extract contact info from: John Doe, john@example.com, (555) 123-4567"}]
})

result["structured_response"]
# ContactInfo(name='John Doe', email='john@example.com', phone='(555) 123-4567')
```

Batch Requests

```
from langchain_openai import ChatOpenAI

model = ChatOpenAI(model="gpt-4.1-mini")
responses = model.batch([
    "Why do parrots have colorful feathers?",
    "How do airplanes fly?",
    "What is quantum computing?"
])
responses
```

```
[AIMessage(content='Parrots have colorful feathers primarily for several
reasons related to survival ...',
additional_kwargs={'refusal': None},
response_metadata={'token_usage': {'completion_tokens': 243,
'prompt_tokens': 15, 'total_tokens': 258, 'completion_tokens_details': ...),

AIMessage(content="Airplanes fly by generating lift, which overcomes the
force of gravity pulling them down. ...",
additional_kwargs={'refusal': None},
response_metadata={'token_usage': {'completion_tokens': 271,
'prompt_tokens': 12, 'total_tokens': 283, 'completion_tokens_details': ...,

AIMessage(content='Quantum computing is a type of computing that
uses ...', additional_kwargs={'refusal': None},
response_metadata={'token_usage': {'completion_tokens': 182,
```

Short-Term Memory

List of messages in conversation

State in each node of agent node

Adding State

```
from langchain.agents import create_agent, AgentState
from langchain.tools import tool, ToolRuntime

class CustomState(AgentState):
    user_id: str

@tool
def get_user_info( runtime: ToolRuntime) -> str:
    user_id = runtime.state["user_id"]
    return "User is John Smith" if user_id == "user_123" else "Unknown user"
```

```
agent = create_agent(
    model="gpt-5-nano",
    tools=[get_user_info],
    state_schema=CustomState,
)

result = agent.invoke({
    "messages": "look up user information",
    "user_id": "user_123"
})
print(result["messages"][-1].content)
```

CheckPointer

Stores graphs state

Allows conversation to be resumed at any time

Need to give the conversation a thread_id

```
from langchain.agents import create_agent
from langgraph.checkpoint.memory import InMemorySaver

agent = create_agent(
    "gpt-5",
    tools=[get_user_info],
    checkpointer=InMemorySaver(),
)

agent.invoke(
    {"messages": [{"role": "user", "content": "Hi! My name is Bob."}]},
    {"configurable": {"thread_id": "1"}},
)
```

Long-term Memory

Stored as JSON in a store object

Each memory has a
- Namespace
- Key

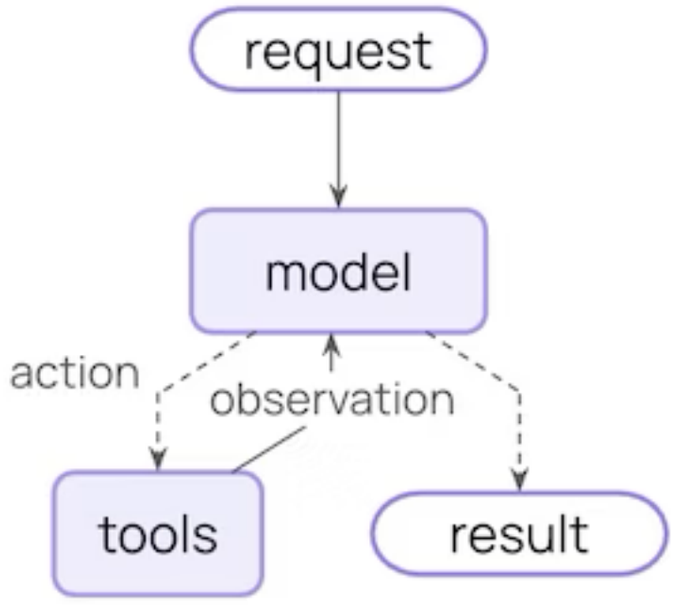
Namespace & key are used to access the memory

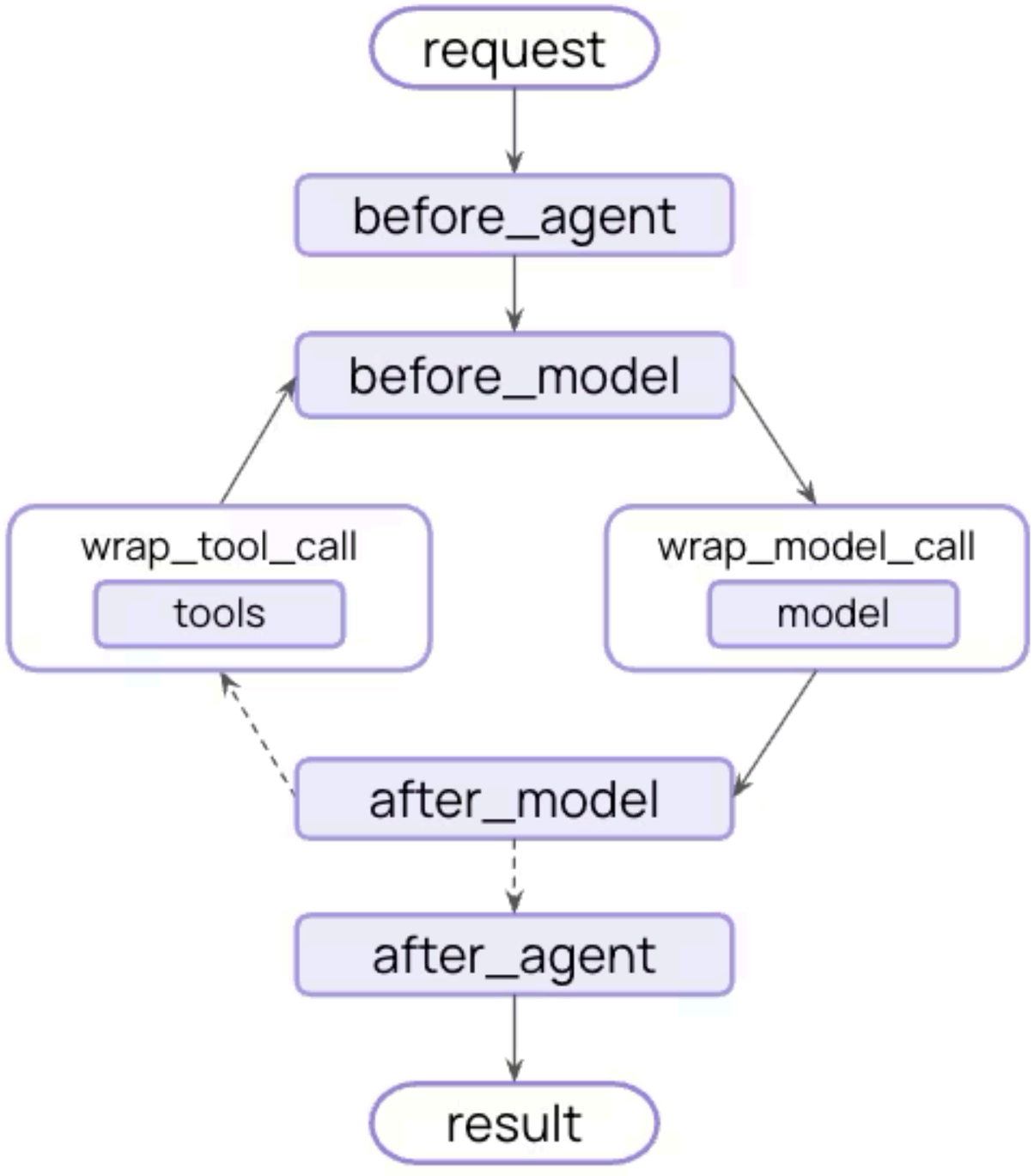
```
from langgraph.store.memory import InMemoryStore

def embed(texts: list[str]) -> list[list[float]]:  return [[1.0, 2.0] * len(texts)]

# InMemoryStore saves data to an in-memory dictionary. Use a DB-backed store in production use.
store = InMemoryStore(index={"embed": embed, "dims": 2})
user_id = "my-user"
application_context = "chitchat"
namespace = (user_id, application_context)
store.put(
    namespace,
    "a-memory",
    {
        "rules": [
            "User likes short, direct language",
            "User only speaks English & python",
        ],
        "my-key": "my-value",
    },
)
# get the "memory" by ID
item = store.get(namespace, "a-memory")
# search for "memories" within this namespace, filtering on content equivalence, sorted by vector similarity
items = store.search(
    namespace, filter={"my-key": "my-value"}, query="language preferences"
)
```

Middleware


request
model
action
observation
tools
result


request
before_agent
before_model
wrap_tool_call
tools
wrap_model_call
model
after_model
after_agent
result

Built-in Middleware

Summarization	Automatically summarize conversation history when approaching token limits.
Human-in-the-loop	Pause execution for human approval of tool calls.
Model call limit	Limit the number of model calls to prevent excessive costs.
Tool call limit	Control tool execution by limiting call counts.
Model fallback	Automatically fallback to alternative models when primary fails.
PII detection	Detect and handle Personally Identifiable Information (PII).
To-do list	Equip agents with task planning and tracking capabilities.
LLM tool selector	Use an LLM to select relevant tools before calling main model.
Tool retry	Automatically retry failed tool calls with exponential backoff.
Model retry	Automatically retry failed model calls with exponential backoff.
LLM tool emulator	Emulate tool execution using an LLM for testing purposes.
Context editing	Manage conversation context by trimming or clearing tool uses.
Shell tool	Expose a persistent shell session to agents for command execution.
File search	Provide Glob and Grep search tools over filesystem files.

SummerizationMiddleware

Automatically summarize conversation history when approaching token limits,

preserving recent messages while compressing older context

```
from langchain.agents import create_agent
from langchain.agents.middleware import SummarizationMiddleware

agent = create_agent(
    model="gpt-4.1",
    tools=[your_weather_tool, your_calculator_tool],
    middleware=[
        SummarizationMiddleware(
            model="gpt-4.1-mini",
            trigger=("tokens", 4000),
            keep=("messages", 20),
        ),
    ],
)
```

Human-in-the-loop

Interrupts agent to get human approval

Responses

- Approve
- Edit
- Reject

Human-in-the-loop Example

```
from langchain.agents import create_agent
from langchain.agents.middleware import HumanInTheLoopMiddleware
from langgraph.checkpoint.memory import InMemorySaver

agent = create_agent(
    model="gpt-4.1",
    tools=[write_file_tool, execute_sql_tool, read_data_tool],
    middleware=[
        HumanInTheLoopMiddleware(
            interrupt_on={
                "write_file": True,  # All decisions (approve, edit, reject) allowed
                "execute_sql": {"allowed_decisions": ["approve", "reject"]},  # No editing allowed
                "read_data": False,
            },
            description_prefix="Tool execution pending approval",
        ),
    ],
    checkpointer=InMemorySaver(),
)
```

Human-in-the-loop Example

```
config = {"configurable": {"thread_id": "some_id"}}

result = agent.invoke(
    {
        "messages": [
            {
                "role": "user",
                "content": "Delete old records from the database",
            }
        ]
    },
    config=config
)
```

Human-in-the-loop Example

```
# The interrupt contains the full HITL request with action_requests and review_configs
print(result['__interrupt__'])
# > [
# >    Interrupt(
# >       value={
# >          'action_requests': [
# >             {
# >                'name': 'execute_sql',
# >                'arguments': {'query': 'DELETE FROM records WHERE created_at < NOW() - INTERVAL \'30
days\';'},
# >                'description': 'Tool execution pending approval\n\nTool: execute_sql\nArgs: {...}'
# >             }
# >          ],
# >          'review_configs': [
# >             {
# >                'action_name': 'execute_sql',
# >                'allowed_decisions': ['approve', 'reject']
# >             }
# >          ]
# >       }
# >    )
# > ]
```

Human-in-the-loop Example

```
agent.invoke(
    Command(
        resume={"decisions": [{"type": "approve"}]}  # or "reject"
    ),
    config=config # Same thread ID to resume the paused conversation
)
```

PIIMiddleware

Detect and handle Personally Identifiable Information

Built-in PII types:

- email: Email addresses
- credit_card: Credit card numbers (validated with Luhn algorithm)
- ip: IP addresses (validated with stdlib)
- mac_address: MAC addresses
- url: URLs (both http/https and bare URLs)

Strategies:

- block: Raise an exception when PII is detected
- redact: Replace PII with [REDACTED_TYPE] placeholders
- mask: Partially mask PII (e.g., ****-****-****-1234 for credit card)
- hash: Replace PII with deterministic hash (e.g., <email_hash:a1b2c3d4>)

PIIMiddleware

```
from langchain.agents.middleware import PIIMiddleware
from langchain.agents import create_agent

agent = create_agent(
    "openai:gpt-5",
    middleware=[
        PIIMiddleware("email", strategy="block"),
    ],
)
```

```
agent.invoke({"messages": "Which email is better rwhitney@sdsu.edu or whitney@cs.sdsu.edu"})
```

```
PIIDetectionError: Detected 2 instance(s) of email in text content
During task with name 'PIIMiddleware[email].before_model' and id '514a9193-82ee-fd2d-7778-96816d13700d'
```

PIIMiddleware

```
from langchain.agents.middleware import PIIMiddleware
from langchain.agents import create_agent

agent = create_agent(
    "openai:gpt-5",
    middleware=[
        PIIMiddleware("email",
                      strategy="mask",
                      apply_to_input = False,
                      apply_to_output = True,)
    ],
)

agent.invoke({"messages": "Which email is better rwhitney@sdsu.edu or whitney@cs.sdsu.edu"})
```

Short answer: Use rwhitney@****.edu as your primary; keep whitney@****.edu as an alias.

Deep Agents

Tackle complex, multi-step tasks

Inspired by applications like Claude Code, Deep Research

Built-in planning + task decomposition loop

A filesystem-backed working memory

Subagents for delegation and context isolation

Persistence across conversations/threads

When to use

Customize agents with skills and memory.
Teach agents as you use them about your preferences, common patterns, etc.
Execute code on your machine or in sandboxes.

Deep Agents - Examples

https://github.com/langchain-ai/deepagents/tree/master/examples

- Deep Reseach
- Downloading agents
- Ralph Mode
- Text-to-sql

DeepAgent Example - Web Search

Agent to perform research using the Web

Tavily

- Search engine optimized for LLMs
- Aggregates up to 20 sites per a single API call
- Uses proprietary AI to score, filter and rank the top most relevant sources and content

Sample Tavily Search

{
  "query": "What is an LLM agent",
  "follow_up_questions": null,
  "answer": "An LLM agent is an AI system built on large language models that can perform tasks autonomously, using tools and memory to interact with users and systems. They can reason, plan, and adapt to complete complex tasks. They differ from basic chatbots by maintaining context and working toward goals.",
  "images": [],
  "results": [
    {
      "url": "https://www.truefoundry.com/blog/llm-agents",
      "title": "LLM Agents : The Complete Guide - TrueFoundry",
      "content": "## What Are LLM Agents?\n\n …”,
      "score": 0.94326943,
      "raw_content": null,
      "favicon": "https://cdn.prod.website-files.com/6291b38507a523837323767 9/62c52662b8171f9546cfa886_Untitled%20design%20(4).png"
    },
    {
      "url": "https://www.k2view.com/what-are-llm-agents/",
      "title": "What are LLM Agents? A Practical Guide - K2view",
      "content": "LLM agents are AI systems that leverage Large Language Models (LLMs) trained on enormous amounts of text data,   "score": 0.94251215,
      "raw_content": null,
      "favicon": "https://www.k2view.com/hubfs/favcon.svg"

DeepAgent Example - Web Search

```
import os
from typing import Literal
from tavily import TavilyClient
from deepagents import create_deep_agent

tavily_client = TavilyClient(api_key=os.environ["TAVILY_API_KEY"])

def internet_search(
    query: str,
    max_results: int = 5,
    topic: Literal["general", "news", "finance"] = "general",
    include_raw_content: bool = False,
):
    """Run a web search"""
    return tavily_client.search(
        query,
        max_results=max_results,
        include_raw_content=include_raw_content,
        topic=topic,
    )
```

DeepAgent Example - Web Search

research_instructions = """You are an expert researcher. Your job is to conduct thorough research
then write a polished report.

You have access to an internet search tool as your primary means of gathering information.

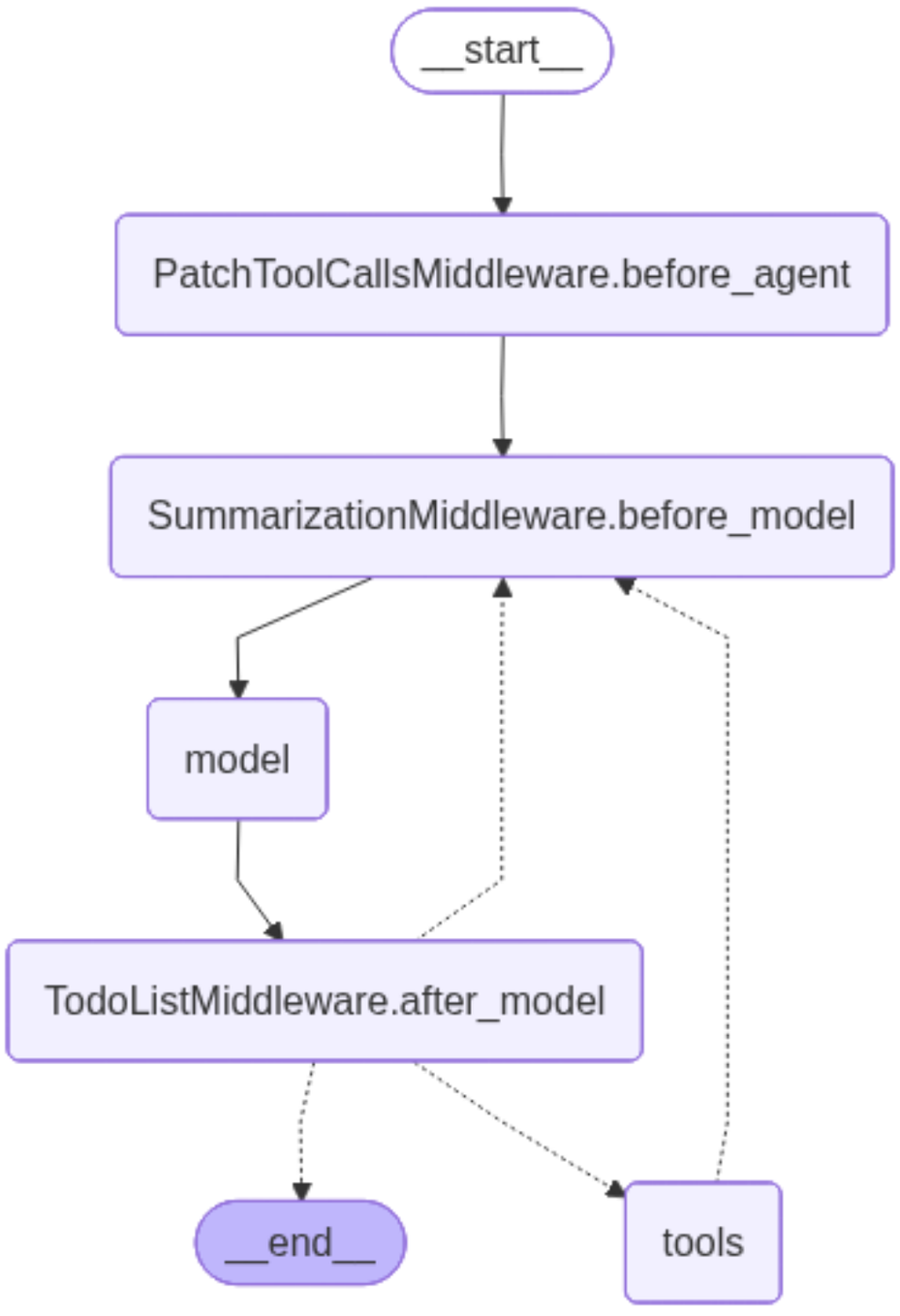
`internet_search`

Use this to run an internet search for a given query. You can specify the max number of results to
return, the topic, and whether raw content should be included.
"""

```
agent = create_deep_agent(
    tools=[internet_search],
    system_prompt=research_instructions
)
agent
```

DeepAgent Example - Web Search

```
agent = create_deep_agent(
    tools=[internet_search],
    system_prompt=research_instructions
)
agent
```



DeepAgent Example - Web Search

```
result = agent.invoke({"messages": [{"role": "user", "content": "What is langgraph?"}]})

# Print the agent's response
print(result["messages"][-1].content)
```

Based on my research, here's a comprehensive overview of LangGraph:

What is LangGraph?

LangGraph is an open-source framework developed by LangChain for building, deploying, and managing complex AI agent workflows and applications. It provides a structured approach to orchestrating multiple Large Language Model (LLM) agents and coordinating their interactions.

…

	Count
HumanMessage	1
AIMessage	3
ToolMessage	2

Model Context Protocol (MCP)

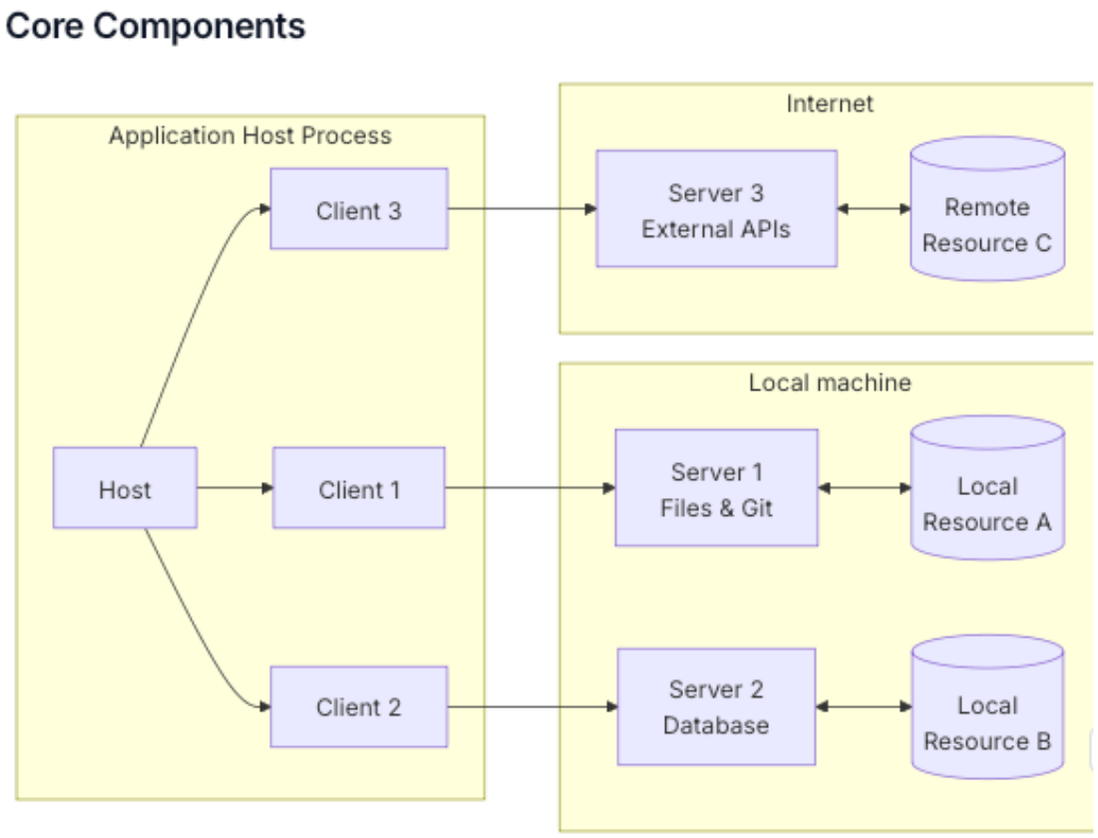
Open protocol for communication between
LLM applications and external data sources & tools

Anthropic in November 2024

MCP Host:
The AI application

MCP Client:
Maintains a connection to an MCP server and
Obtains context from an MCP server

MCP Server:
A program that provides context to MCP clients



https://modelcontextprotocol.io/docs/learn/architecture

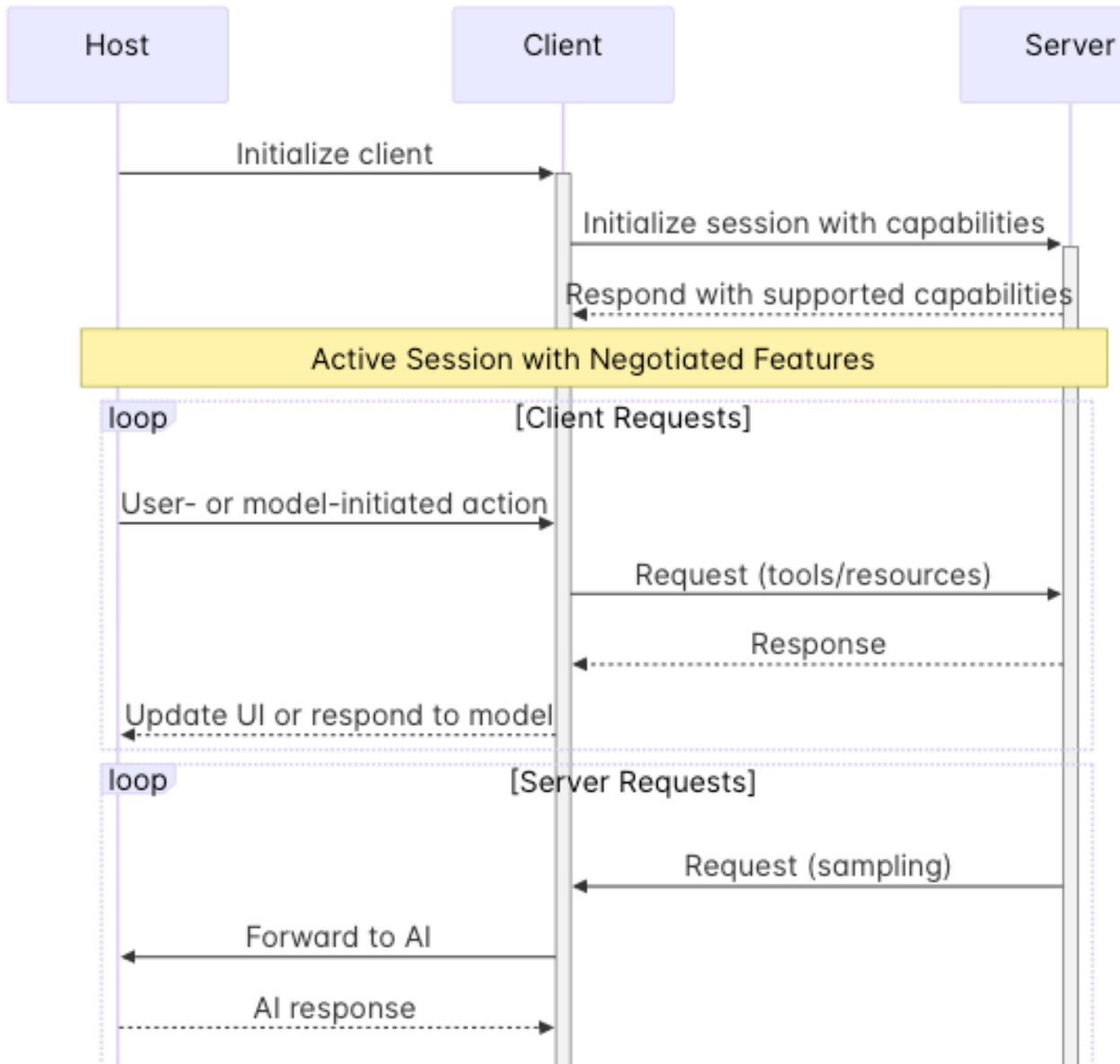
Capability Negotiation

Servers

Declare capabilities like resource subscriptions, tool support, and prompt templates

Clients

Declare capabilities like sampling support and notification handling



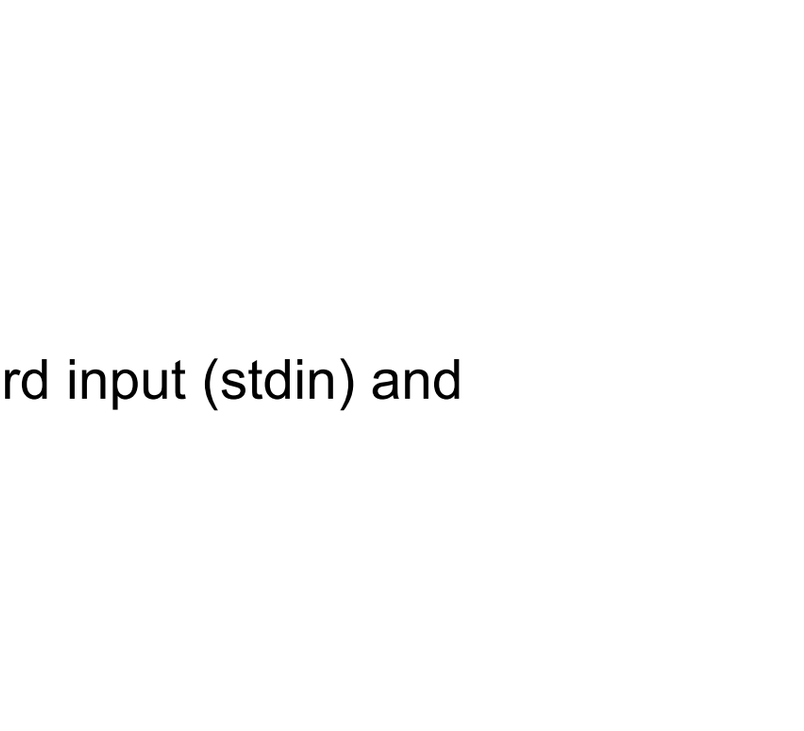
Transports - STDIO

The client launches the MCP server as a subprocess.

The server reads JSON-RPC messages from its standard input (stdin) and
sends messages to its standard output (stdout)

Messages may be
JSON-RPC requests, notifications, responses
or a JSON-RPC batch containing one or more requests and/or notifications.

Messages are delimited by newlines, and MUST NOT contain embedded newlines.



JSON-RPC

Specification on how to structure data used to communicate between client & server

--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

--> {"jsonrpc": "2.0", "method": "subtract", "params": [23, 42], "id": 2}
<-- {"jsonrpc": "2.0", "result": -19, "id": 2}

--> {"jsonrpc": "2.0", "method": "subtract", "params": {"subtrahend": 23, "minuend": 42}, "id": 3}
<-- {"jsonrpc": "2.0", "result": 19, "id": 3}

--> {"jsonrpc": "2.0", "method": "subtract", "params": {"minuend": 42, "subtrahend": 23}, "id": 4}
<-- {"jsonrpc": "2.0", "result": 19, "id": 4}

--> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}
--> {"jsonrpc": "2.0", "method": "foobar"}

Transports - Streamable HTTP

Server MUST provide a single HTTP endpoint path
Supports both POST and GET methods

The client MUST use HTTP POST to send messages to the server

The client MUST include an Accept header,

The body of the POST request MUST be one of the following:

A single JSON-RPC request, notification, or response
An array batching one or more requests and/or notifications
An array batching one or more responses

Client Roots & Sampling

Clients can expose part of the filesystem (root) to the server

Which directories and files they have access to

Servers can request text, audio, or image-based interactions

Server Features

Server provides

Prompts

Resources:

Files, database schemas, or application-specific information

Tools

MCP Server Examples

Math Server

```
from fastmcp import FastMCP

mcp = FastMCP("Math")

@mcp.tool()
def add(a: int, b: int) -> int:
    """Add two numbers"""
    return a + b

@mcp.tool()
def multiply(a: int, b: int) -> int:
    """Multiply two numbers"""
    return a * b

if __name__ == "__main__":
    mcp.run(transport="stdio")
```

Weather Server

```
from fastmcp import FastMCP

mcp = FastMCP("Weather")

@mcp.tool()
async def get_weather(location: str) -> str:
    """Get weather for location."""
    return "It's always sunny in New York"

if __name__ == "__main__":
    mcp.run(transport="streamable-http")
```

```
from langchain_mcp_adapters.client import MultiServerMCPClient
from langchain.agents import create_agent

client = MultiServerMCPClient(
    {
        "math": {
            "transport": "stdio",  # Local subprocess communication
            "command": "python",
            "args": ["/path/to/math_server.py"],
        },
        "weather": {
            "transport": "http",  # HTTP-based remote server
            # Ensure you start your weather server on port 8000
            "url": "http://localhost:8000/mcp",}})

tools = await client.get_tools()
agent = create_agent(
    "claude-sonnet-4-5-20250929",
    tools
)
math_response = await agent.ainvoke(
    {"messages": [{"role": "user", "content": "what's (3 + 5) x 12?"}]}
)
weather_response = await agent.ainvoke(
    {"messages": [{"role": "user", "content": "what is the weather in nyc?"}]}
)
```