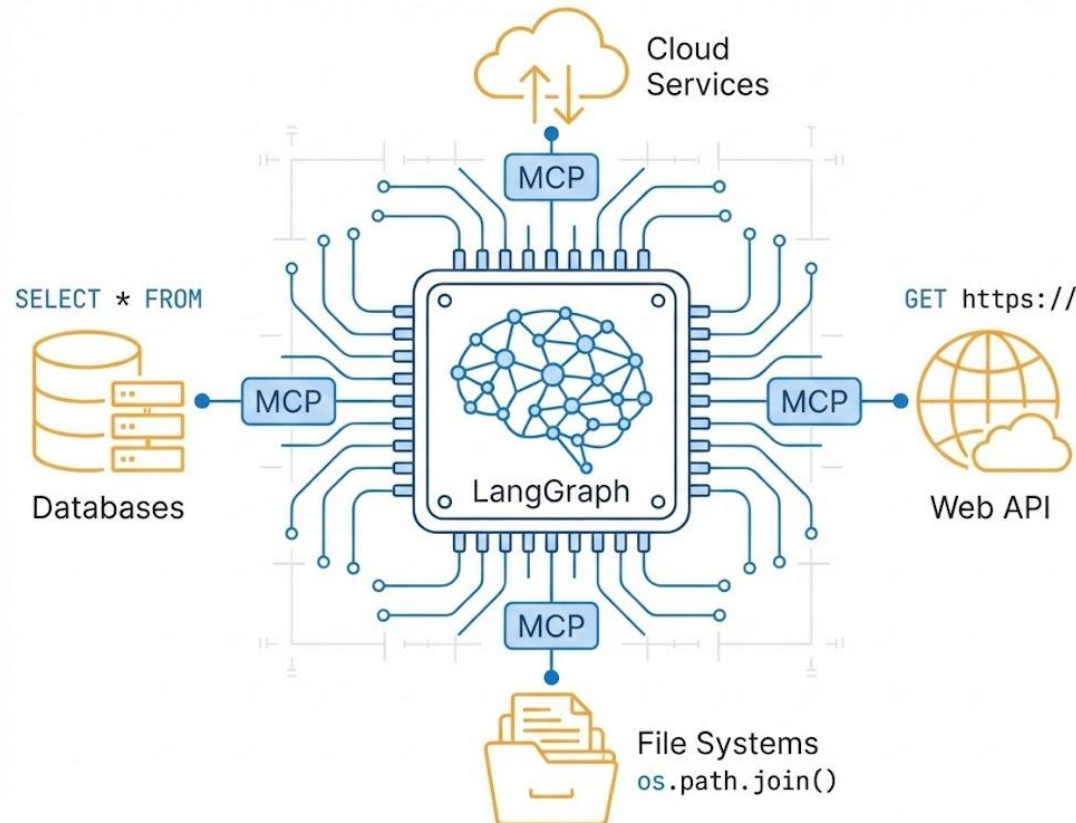


# NeuroSymbolic Artificial Intelligence at Scale

## Parte: 6.1 (2025-26) From RAG LLM to Agentic LLM (Part 2)

Paolo Nesi, [paolo.nesi@unifi.it](mailto:paolo.nesi@unifi.it)

<https://www.disit.org/>



Gianni Pantaleo

[gianni.pantaleo@unifi.it](mailto:gianni.pantaleo@unifi.it)

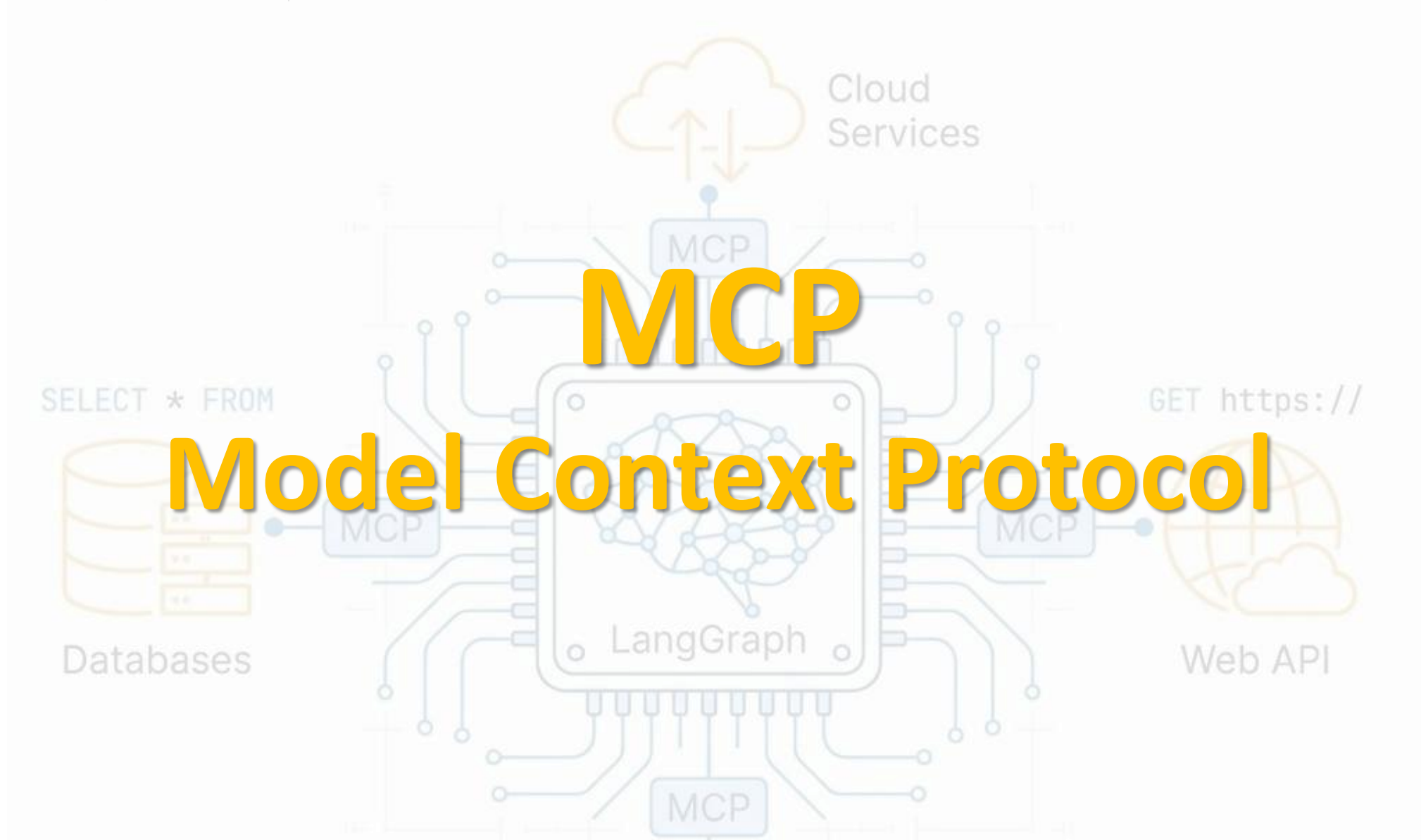
**DISIT LAB**

Google Scholar

<https://scholar.google.com/citations?user=ohyLOOUAAAAJ&hl=it>

# Outline

- AI Agents: Definitions & Motivations
- The Roadmap to AI Agents
  - ❖ *Reasoning*
  - ❖ *Acting: Tool calling*
  - ❖ *Reasoning + Acting: the ReAct framework*
- Taxonomies and Design Patterns of AI Agents
- **The Model Context Protocol: MCP** ←
- Orchestration Frameworks: LangGraph



# Scaling Tool Integration: The $M \times N$ Problem

- In real-world scenarios, many different AI applications may have to interact with many different external services.
- Assuming we have:
  - **M** distinct AI agents or applications (e.g., a customer support agent, a research agent, a coding assistant, a data analysis agent);
  - **N** external services (e.g., programmatic functions, databases, REST APIs, cloud storage etc.).
- Every (*application, service*) pair usually requires a dedicated custom adapter that handles the specific authentication protocol, data serialization format / schema / payload, error handling logic, etc.
- The total number of adapters is therefore:

$$M \times N$$

# Scaling Tool Integration: The $M \times N$ Problem

- This is known as the  $M \times N$  Problem.
- The  $M \times N$  integration matrix has significant drawbacks and issues:
  - **Fragility**: a change in any API service or external tool requires to modify each adapter connected to it (up to  $M$  adapters per service)
  - **Redundancy**: identical integration logic is re-implemented independently for each agent, with no possibility of reuse
  - **Poor scaling**: adding a new agent requires writing  $N$  new adapters from scratch; adding a new service requires updating  $M$  existing ones
- This is defined as the  $M \times N$  problem, which represents a fundamental scalability barrier for agentic systems from growing beyond small, hand-crafted tool sets.

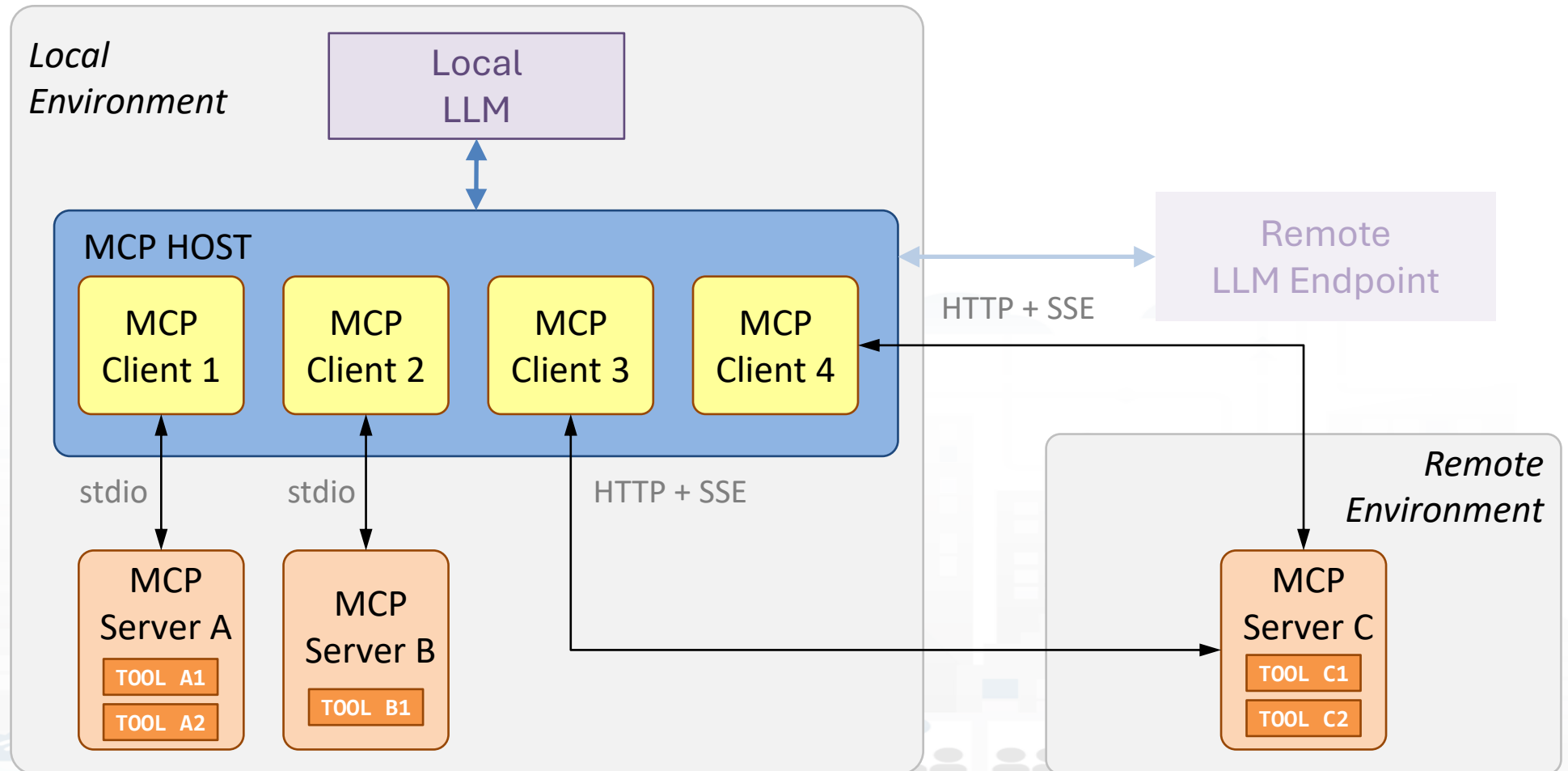
# The Model Context Protocol (MCP)

- The Model Context Protocol (MCP) is an open standard published by Anthropic [Anthropic, 2024b] to resolve the  $M \times N$  problem.
- MCP defines a single universal interface that any data source or tool can expose, and any AI agent can consume, without specialized or custom “glue code” for integration.
  - Each **tool or service exposes** a standardized interface descriptor, defining its name, capabilities, input schema, and expected output format.
  - Each **AI application or agent consumes** these interface descriptors uniformly, invoking any compliant tool through the same protocol, regardless of its underlying implementation
- MCP therefore specifies a common contract for capability discovery, tool invocation, and result handling, so that any compliant AI model can interact with any compliant tool.
- Historical parallel: before USB, every peripheral and device had its own proprietary connector. *MCP is like the USB-C in the domain of AI applications.*

# The MCP Three-Layer Architecture & Specifications

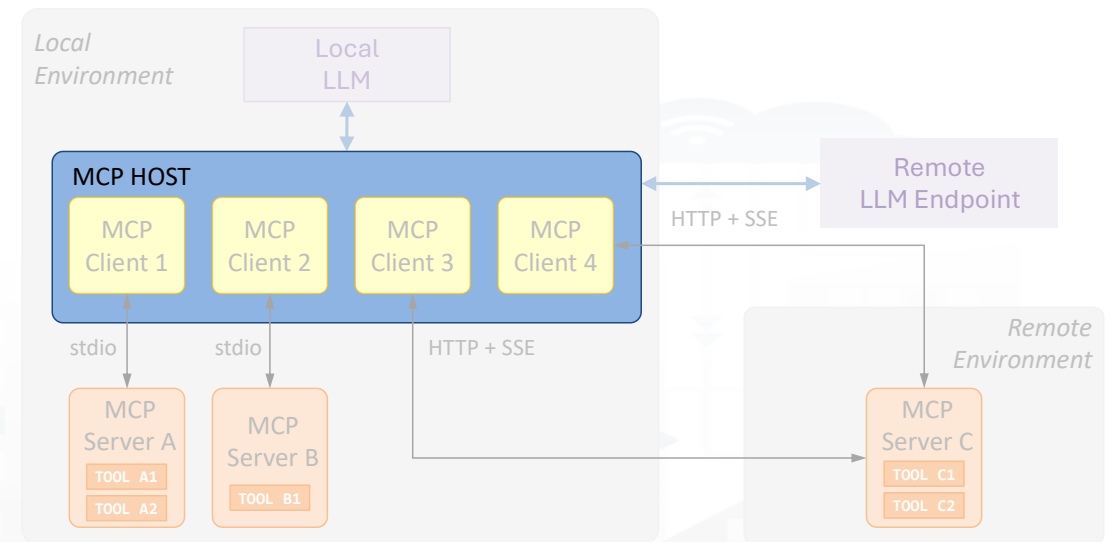
- MCP has a *three-layer* architecture: **Host**, **Client** and **Server**.
- MCP is an open protocol that standardizes how AI applications (Hosts / Clients) communicate with external data sources and tools (Servers).
- MCP uses JSON-RPC 2.0 as messaging protocol with several transport mechanisms:
  - stdio (Local)
  - Streamable HTTP (Remote)
- Primitives exposed by MCP servers:
  - Tools (actions)
  - Resources (data)
  - Prompts (templates)

# The MCP Architecture



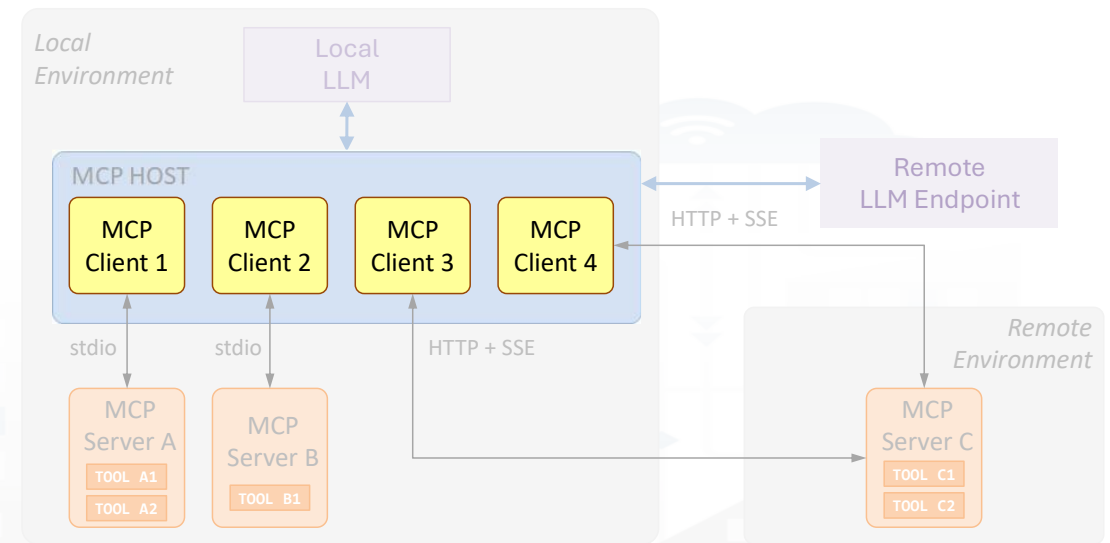
# MCP Host: The AI Application

- **MCP Host** is the AI application that needs to use external capabilities
  - The Host is the application module that serves as the entry point for user interactions (e.g. GUI or CLI).
  - The Host may embed the LLM directly or manage connections to an external LLM endpoint
  - The Host can manage one or more Client connections.
  - The Host acts as a control layer to filter which Server tools are exposed to the LLM.
  - The Host receives results from multiple Servers and aggregates them as context into coherent LLM prompts.
  - The Host enforces security policies and manages user authentication, authorization, and access control.
  - The Host may implement Human-In-The-Loop (HITL) by requiring user approval before executing sensitive tool calls.



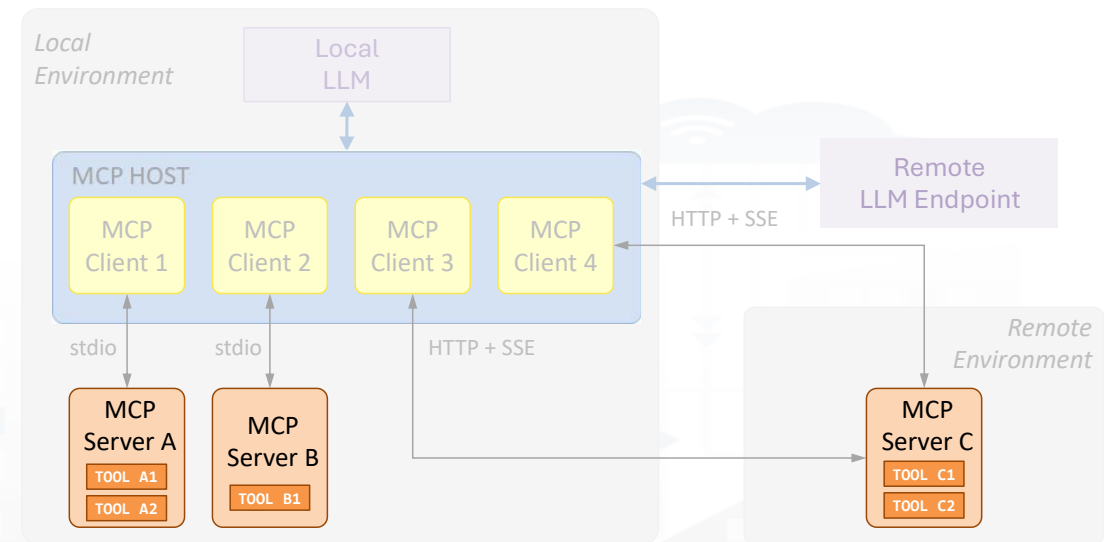
# MCP Client: The Protocol Connector

- **MCP Client** — A component embedded in the Host that is connected and handles communication with a specific MCP server.
  - Each Client manages the connection lifecycle with the Server:
    - Initialization → capability negotiation → operation → shutdown
  - Capability negotiation: after initialization, Client and Server exchange supported capabilities (tools, resources, prompts).
  - The Client translates requests from/to LLM into JSON-RPC messages:
    - when the LLM decides to invoke a tool, the Client translates such decision into a JSON-RPC message and forwards it to the server;
    - the Client also translate back the Server's JSON-RPC response into a format the LLM can reason over.



# MCP Server: The Capability Provider

- **MCP Server** — A standalone process that exposes a specific set of capabilities to any compliant Client.
  - A server can be a **local** process communicating over stdio, as well as a **remote** service over Streamable HTTP or HTTP+SSE (Server-Sent Events).
  - Server lifecycle: receives initialize request from Client → declare capabilities → handle requests → shutdown
  - Each Server wraps one or more external tools, data sources, or services, exposing them through the MCP protocol that any compliant Client can discover and invoke.
  - Servers are composable: prefer several small servers over monolithic ones
    - Allows separation of concerns and simplifies debugging
    - Improves tool selection accuracy and scaling capabilities
  - MCP Servers do NOT make decisions, they expose capabilities. The LLM (via Host) decides when and how to use them

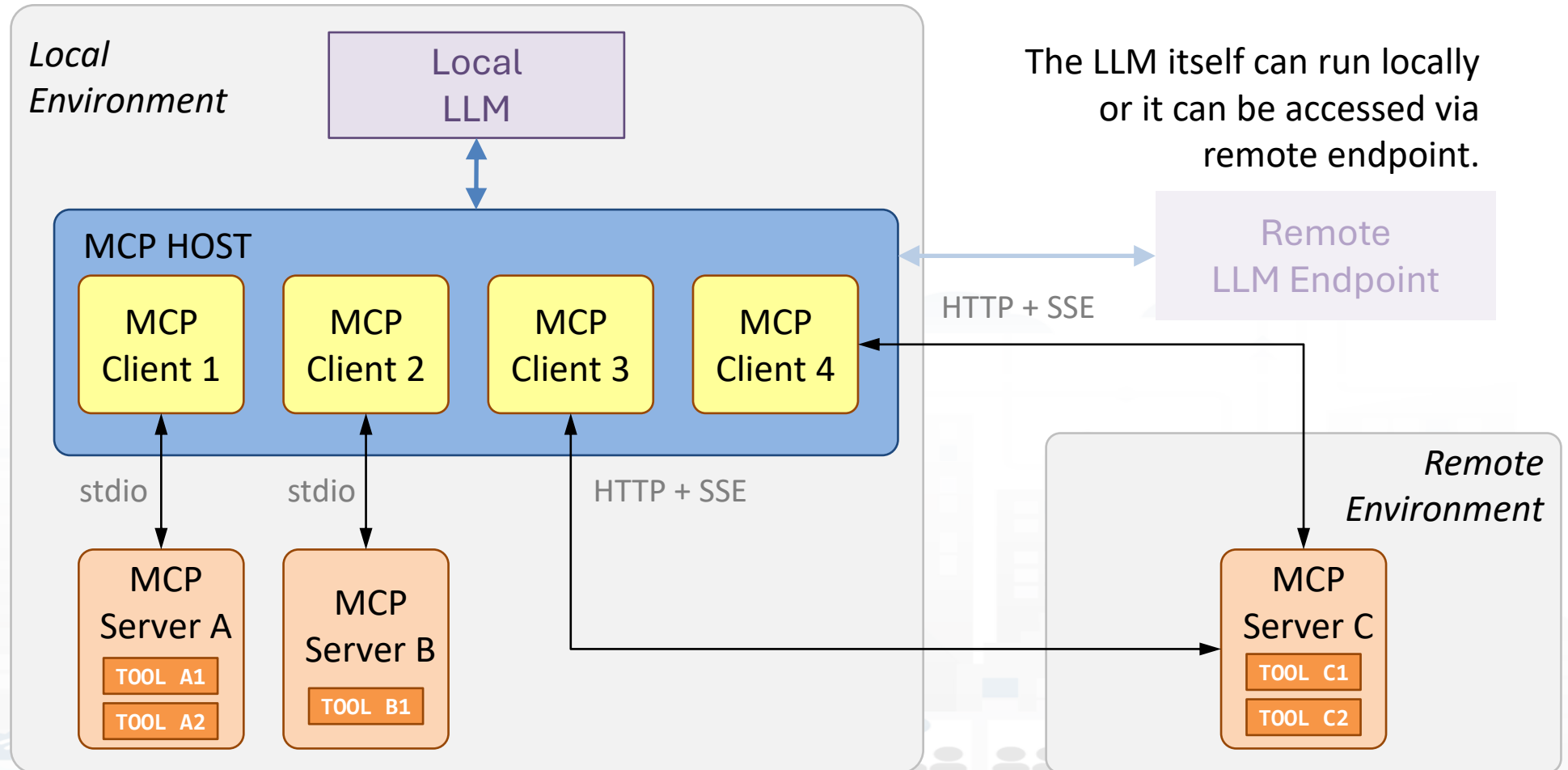


# MCP Sampling

- MCP enables Servers to request LLM completions or additional generations from the LLM or additional input Host via clients, enabling Human In The Loop (HITL) patterns where the server waits for human input before proceeding.
- Typically, the flow is:  
LLM / Host → invokes tool → MCP Server responds.
- With Sampling, the flow is reversed:  
MCP Server → requires LLM completion / Host interaction → LLM/ Host responds.
- This enables patterns where the server, while running a tool, may need contextual input from the model or the interactions from the user.
- Clients that support sampling must declare the sampling capability during initialization with the server.

# The MCP Architecture

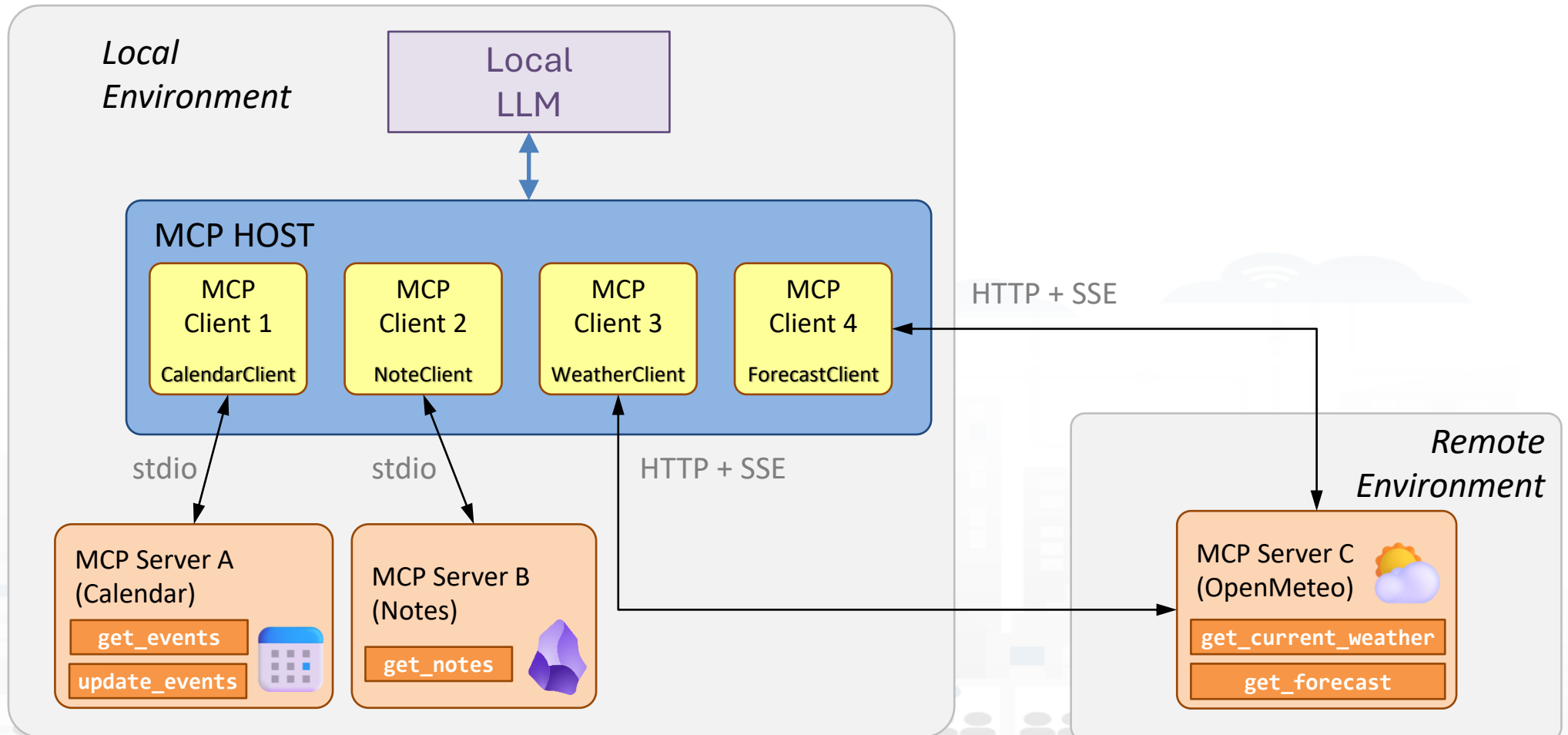
The MCP Host can run on the same machine as the local MCP server(s) (desktop application) or be deployed on a remote machine (cloud-hosted agent).



The LLM itself can run locally or it can be accessed via remote endpoint.

# Use Case Example: MCP Implementation for Agentic Applications

**Example Query:**  
“Check my upcoming calendar events and my notes, then update each event in my calendar with a recommendation based on the current weather or weather forecast.”

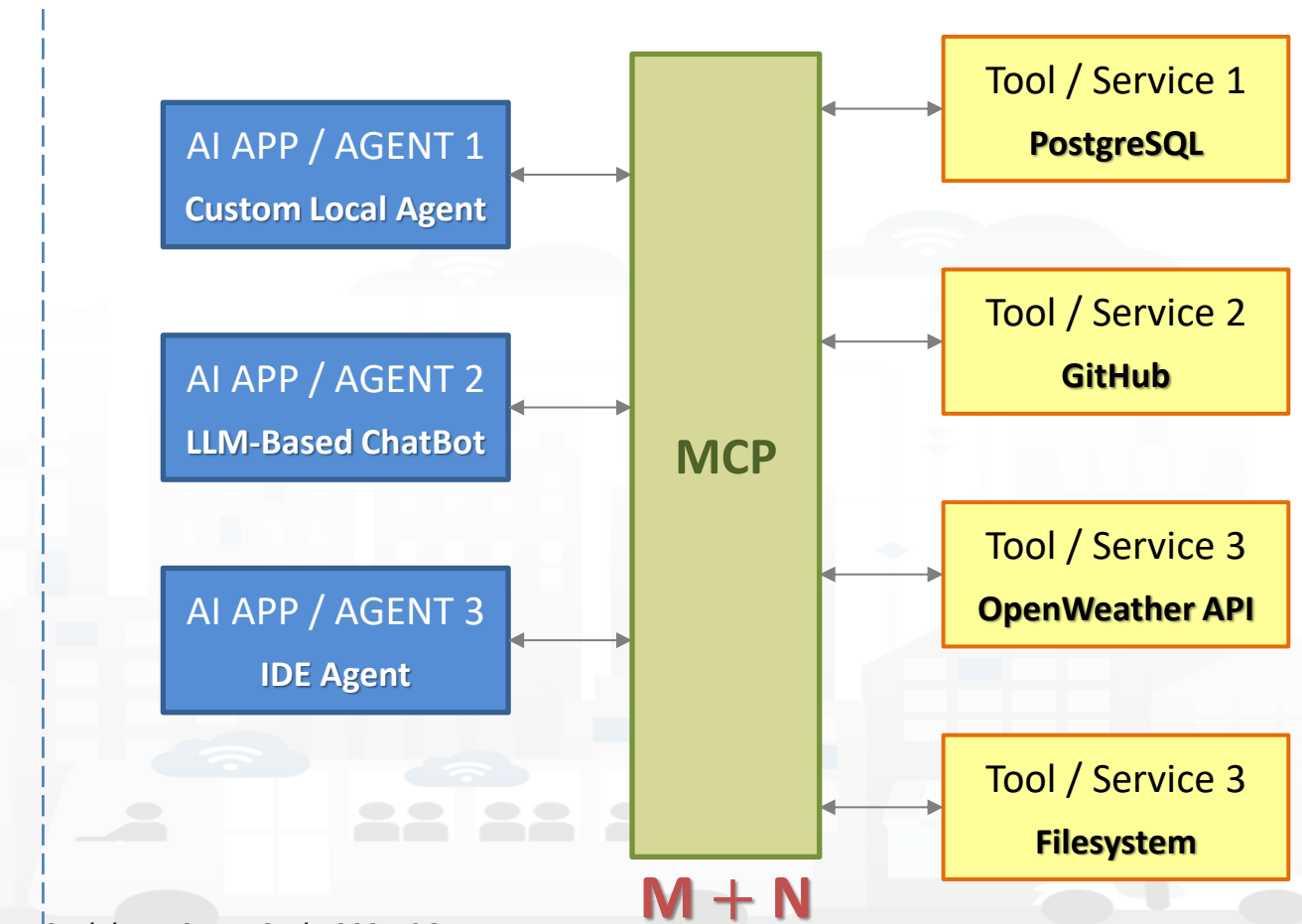
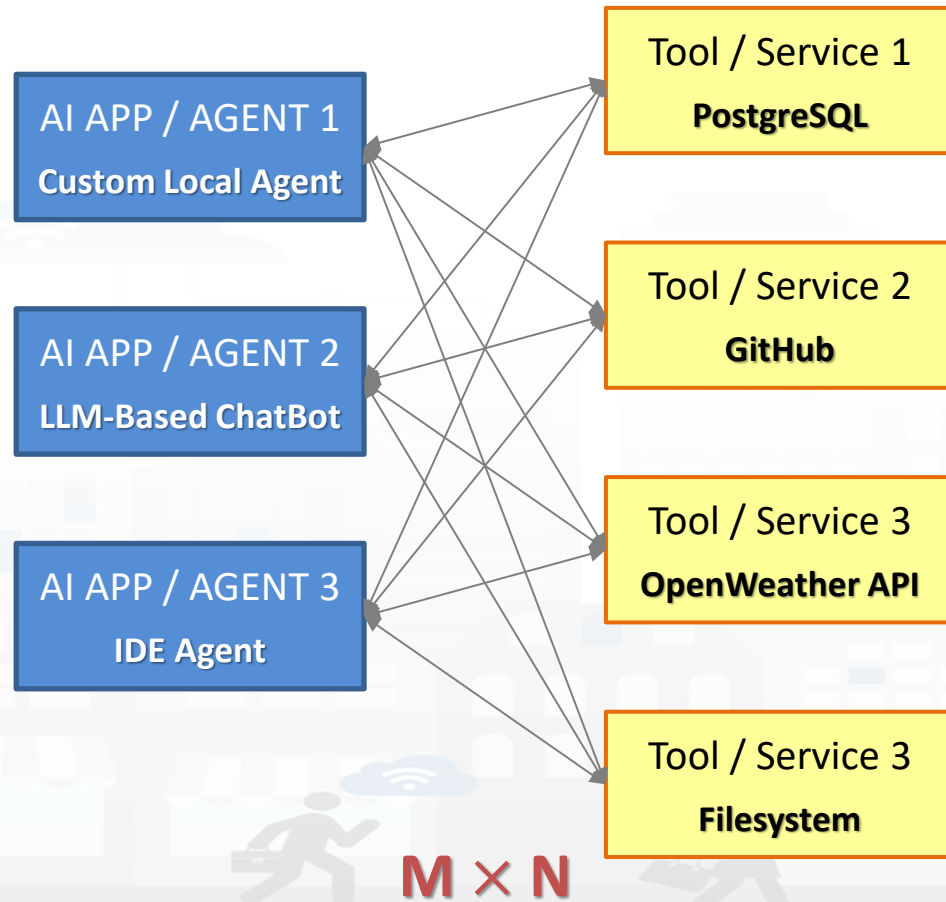


# MCP: from $M \times N$ Problem to $M + N$ Problem

- Without MCP:
  - Every AI application must implement a custom integration for every tool: **M agentic AI applications for N tools =  $M \times N$  connectors** to build and maintain.
- With MCP:
  - Every external service exposes a single standardized MCP server interface: implemented once, usable by any MCP-compatible client;
  - Each MCP Server can expose multiple capabilities (i.e., tools, resources and prompts).
  - Every AI application implements a single MCP client, capable of connecting to any MCP server, regardless its underlying implementation;
  - Result: **M clients + N servers =  $M + N$  implementations.**

# MCP: from $M \times N$ Problem to $M + N$ Problem

- Therefore, the MCP reduces the  $M \times N$  connection integration to  $M + N$  connections.



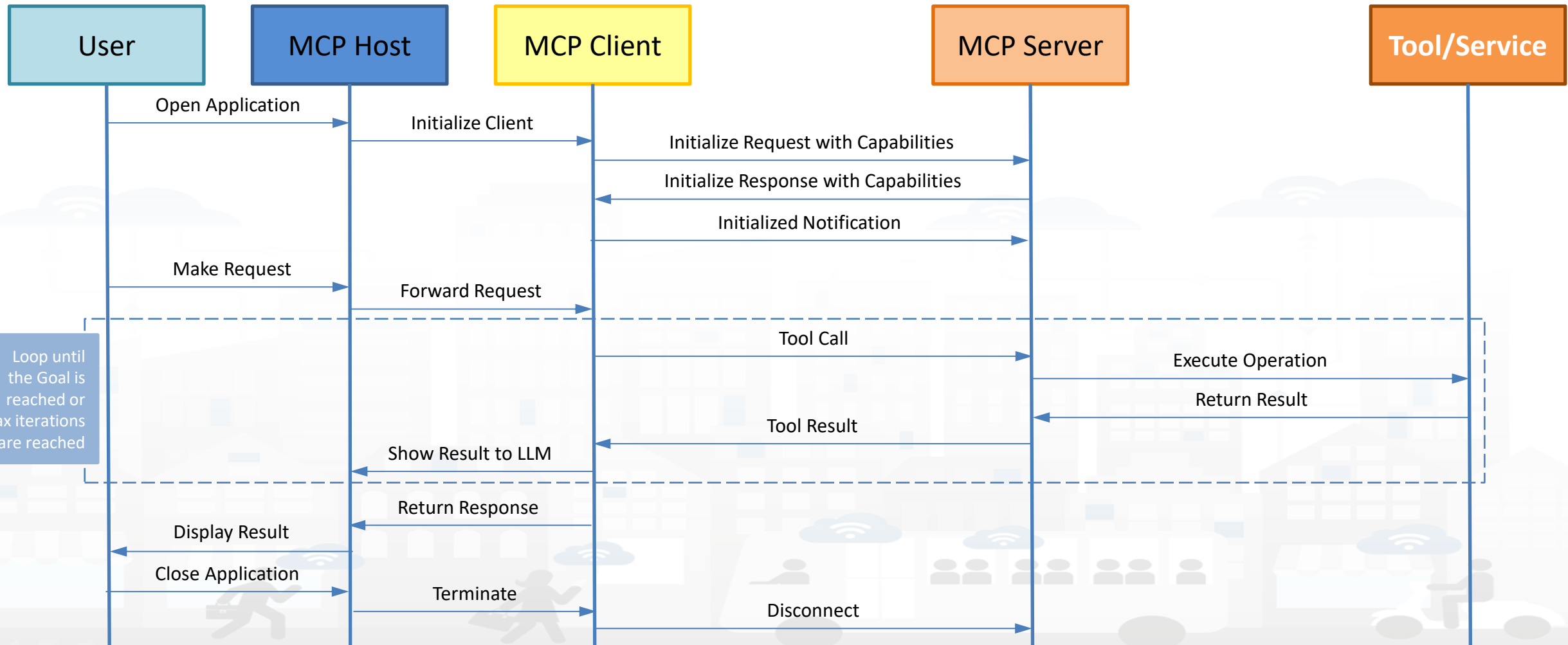
# MCP Transport Mechanisms

- **stdio** — The server runs as a local subprocess, communicating via stdin/stdout;
  - used for local servers spawned by the host;
  - one dedicated subprocess per Client (1:1 connection);
  - no network overhead: ideal for local tools (filesystem, local databases, shell commands)
  - Server lifecycle is tied to the Client session: starts when the Client connects, terminates when it disconnects
  - configuration: `{"command": "python", "args": ["server.py"]}`
- **HTTP (streamable-http)** — The server runs as a persistent HTTP process;
  - used for shared remote services, exploiting Server-Sent Events (SSE) for asynchronous communication: the Server directly pushes updates to the Client (no Client polling required)
  - supports multiple simultaneous clients (N:1);
  - requires network configuration and authentication, designed for remote services and cloud-deployed tools
  - Server lifecycle is independent: always running, ready to serve any incoming Client connection
  - configuration: `{"url": "http://host:port/mcp"}`

# MCP Primitives, i.e.: the “Capabilities”

- **Tools:** executable functions the LLM can invoke to perform actions or retrieve computed results (e.g., `get_weather`, `execute_code`, `send_email`).
  - Defined by name, docstring-derived description, JSON input schema specifying the expected parameters
  - Discovery: Client calls `tools/list` → Server returns available tools
- **Resources:** read-only data sources the host can access to provide context to the LLM (e.g., file contents, database records, live sensor readings).
  - Identified by URI: `file:///path`, `postgres://db/table`, `custom://...`
  - Unlike tools, resources are not invoked by the LLM, but they are fetched by the host and injected into the context
- **Prompts:** reusable, parameterized prompt templates defined server-side that the host can retrieve and instantiate at runtime (e.g., a standardized code review template, a domain-specific system prompt)
  - Each prompt has role (user/assistant) and content (text/image/resource)

# MCP Interactions for Tool Calling – Sequence Diagram



# MCP Example Messages

- **Transport layer: JSON-RPC 2.0** – every message is a request, response, or notification

```
{ "jsonrpc": "2.0", "id": 1, "method": "tools/call", "params": { ... } }  
{ "jsonrpc": "2.0", "id": 1, "result": { ... } }
```

- **Mandatory lifecycle:** initialize → notifications/initialized → work

```
// Client initialize request with capabilities
```

```
{ "method": "initialize", "params": {  
    "protocolVersion": "2024-11-05",  
    "clientInfo": { "name": "MyHost", "version": "1.0" },  
    "capabilities": { "roots": {}, "sampling": {} } }  
}
```

```
// Server response with capabilities
```

```
{ "jsonrpc": "2.0", "id": 0, "result": {  
    "protocolVersion": "2024-11-05",  
    "serverInfo": { "name": "CalendarServer", "version": "1.0" },  
    "capabilities": {  
        "tools": { "listChanged": true // server can notify when tool list changes },  
        "resources": { "subscribe": true, // client can subscribe to resource changes },  
        "prompts": { "listChanged": true }  
    }  
}}}
```

# MCP Example Messages

- **Tool discovery** via tools/list → invocation via tools/call

```
// Client → Server (request)
{ "jsonrpc": "2.0", "id": 1, "method": "tools/list" }
```

```
// Server → Client (response)
{ "jsonrpc": "2.0", "id": 1, "result": { "tools": [{ "name": "get_calendar_events", ... } ] } }
```

- **Tool calling**

```
// Client → Server (request)
{ "jsonrpc": "2.0", "id": 2, "method": "tools/call",
  "params": { "name": "get_calendar_events", "arguments": { "date_range": 7 } } }
```

```
// Server → Client (response)
{ "jsonrpc": "2.0", "id": 2, "result": {
  "content": [{ "type": "text", "text": "[{"date": "2026-04-28"...}]" } ],
  "isError": false
}
}
```

# FastMCP

- FastMCP (<https://gofastmcp.com/>) is a high-level Python framework for building and connecting MCP servers and clients.
- FastMCP wraps the official MCP Python SDK and automatically handles protocol lifecycle, schema generation and transport management.
  - Avoid manual JSON-RPC message handling, session management, and capability negotiation;
  - Auto-generates the MCP tool schema (name, description, parameter types) directly from function signatures and docstrings — no more “tools schemas” to maintain manually;
  - Single framework for both server (exposing tools) and client (consuming tools) sides
- Therefore, FastMCP mitigates the complexity of using official MCP SDK, allowing developers to focus on tool logic rather than protocol details.

# FastMCP

- FastMCP v1.0 has been incorporated into the official MCP SDK.
- FastMCP v2.x (maintained by Prefect) extends it with additional features such as HTTP transport and multi-server clients.
- FastMCP v3.0 (2025) introduces a more modular and composable architecture
- Installation (requires Python  $\geq 3.10$ ): `pip install fastmcp`
- The core element is the FastMCP class, which represents the MCP server, e.g.:  

```
mcp = FastMCP("CalendarServer")
```
- Act as the central registry for all tools, resources, and prompts exposed by the server, automatically generating their JSON schemas from function signatures and docstrings
- It provides the `run()` method to start the server with the transport method chosen at the design stage (`stdio` or `HTTP`).

# FastMCP: Core Server Decorators

- FastMCP uses decorators to register Python functions as MCP-compliant tools, resources, or prompts automatically
- **@mcp.tool()** — exposes a Python function as an MCP tool the LLM can invoke; schema is auto-generated from type hints (→ JSON schema properties) and docstring (→ description)
  - When FastMCP sees `@mcp.tool()`, it reads the function's type hints to generate the JSON input schema and the docstring to generate the tool description → *No manual schema writing required!*
- **@mcp.resource("uri://{param}")** exposes read-only data as an MCP resource; the LLM or host can fetch it to inject context without triggering side effects
- **@mcp.prompt()** exposes a reusable parameterized prompt template; the host retrieves and instantiates it at runtime
- All three decorators support sync and async functions transparently

# FastMCP: Core Server Decorators

```
import sys
from fastmcp import FastMCP
```

```
mcp = FastMCP("ServerName")
```

FastMCP Tool Decorator

```
@mcp.tool()
```

```
def my_tool(param: str) -> str:
```

```
    """Tool description used as MCP description."""
```

```
    return f"result: {param}"
```

```
if __name__ == "__main__":
```

```
    transport = sys.argv[1] if len(sys.argv) > 1 else ""
```

```
    if transport == "http":
```

```
        mcp.run(transport="http", host="0.0.0.0", port=8000) # HTTP transport
```

```
    elif transport == "stdio": # Explicit stdio transport
```

```
        mcp.run(transport="stdio")
```

```
    else:
```

```
        mcp.run() # Default transport: stdio
```

# FastMCP: Client Connecting to Server

```
from fastmcp import Client
```

```
# Single server
```

```
async with Client("server.py") as client:           # stdio  
async with Client("http://localhost:8000/mcp") as client: # HTTP
```

```
# Multi-server config (tools prefixed with client name)
```

```
async with Client({  
    "mcpServers": {  
        "CalendarClient": {"command": "python", "args": ["calendar_server.py"]},  
        "WeatherClient": {"url": "http://127.0.0.1:8000/mcp"},  
    }  
}) as client:
```

```
    tools = await client.list_tools()           # capability discovery
```

```
# In the agentic scenario, the following tool call is typically generated by the LLM –  
# not hardcoded. Shown here explicitly to illustrate the mechanics.
```

```
result = await client.call_tool("CalendarClient_get_calendar_events", {"date_range": 3})  
text = result.content[0].text
```

# FastMCP: Tools Schema Conversion

```
tools_schema = [  
  {  
    "type": "function",  
    "function": {  
      "name": t.name,  
      "description": t.description or "",  
      "parameters": t.inputSchema or {"type": "object", "properties": {}},  
    }  
  }  
  for t in await client.list_tools()  
]
```

FastMCP's `list_tools()` returns FastMCP tool objects; Ollama expects OpenAI-format dicts — conversion is manual.

**Multi-server tool name prefixing:** when using a multi-server config, FastMCP automatically prefixes tool names with the client key: `CalendarClient_get_calendar_events`, `WeatherClient_get_current_weather`.

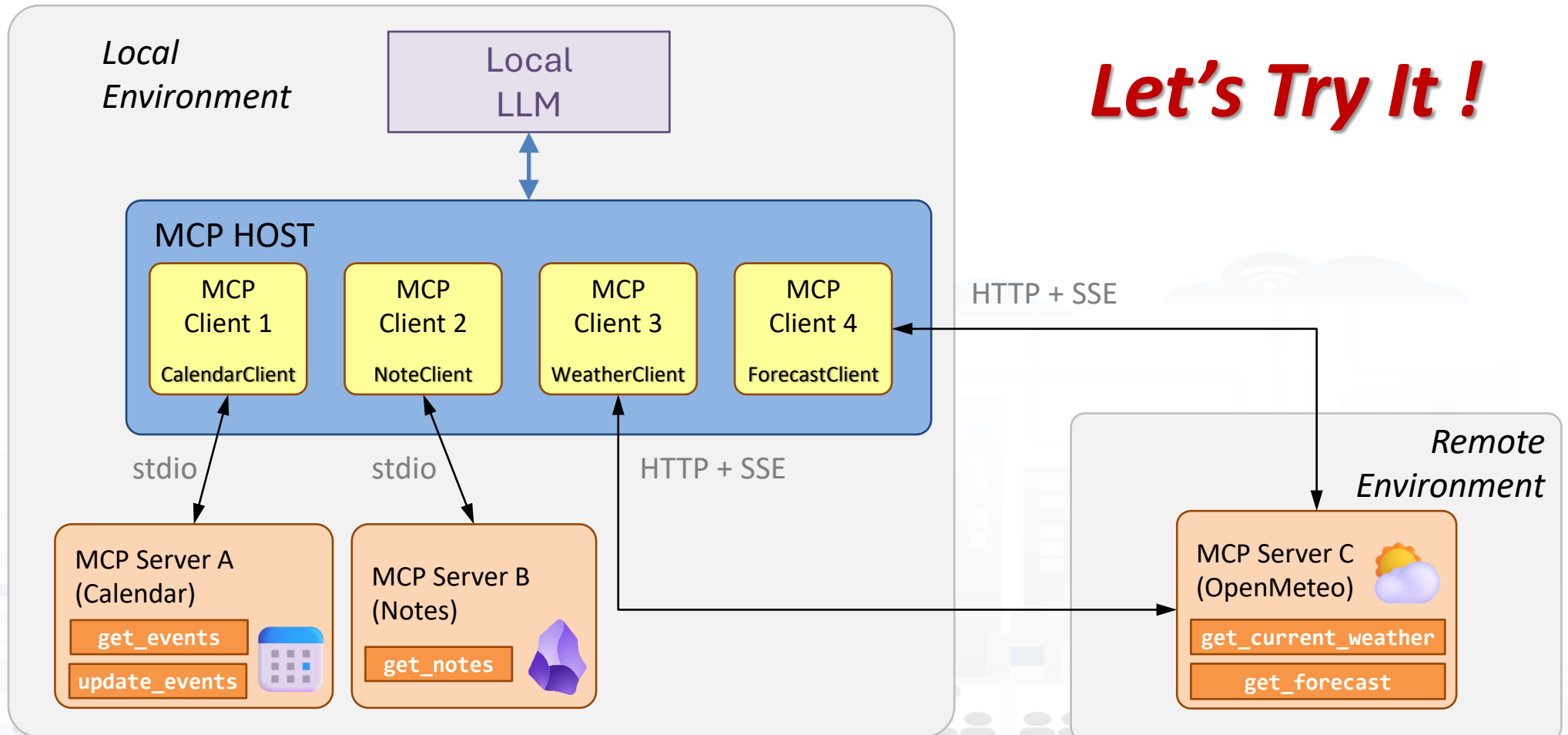
The LLM must use the prefixed name in tool calls; `client.call_tool()` resolves it to the correct server.

# MCP Primitives – Key Production Best Practices

- **Tool filtering** — expose only the tools relevant to the current task to reduce context token usage and improve tool selection accuracy in small models.
- **Anti-loop guard** — track `seen_calls` as a set of `name:args` signatures; if the model retries the same call, return a cached response instead of making another network request.
- **Single responsibility servers** — one server per domain (calendar, notes, weather) rather than one monolithic server; enables independent scaling, fault isolation, and selective tool exposure by the host.

# Use Case Example: MCP Implementation for Agentic Applications

**Example Query:**  
“Check my upcoming calendar events and my notes, then update each event in my calendar with a recommendation based on the current weather or weather forecast”



**Let's Try It !**

# Code Example: Building an MCP-based LLM Agentic System

- **Scenario:** Design a goal-oriented agentic system: the aim is to read the user's local calendar and notes, identify weather-dependent events, consult live weather data for current weather and/or weather forecast, and write an “outdoor/indoor” recommendation back into each event description in the calendar.
- Query example: *"Check my upcoming calendar events and my notes, then update each event in my calendar with a recommendation based on the current weather or weather forecast."*
- The agent should decide autonomously which tools to invoke based on what it discovers at runtime (no hardcoded sequence of steps)

# Code Example: Building an MCP-based LLM Agentic System

- The agent reads the user's **local calendar** (.ics file) and **local notes** (e.g.: Obsidian vault .md files) through two stdio MCP servers and retrieves **live weather data** through a third-party HTTP MCP server (e.g.; OpenMeteo).
- After reading the calendar, the agent discovers from the event descriptions that two events have an outdoor option: it should **autonomously decide** to check the weather for those locations before confirming the venue.
- For each weather-dependent event, the agent calls the appropriate weather tool.
- Finally, the agent writes the recommendation back into the calendar by calling a dedicated tool (`update_events`) with a real side effect on the filesystem.

# Code Example: Building an MCP-based LLM Agentic System

- Notes – Obsidian .md files:

## Florence event.md

### # Florence aperitivo

I have to book a venue for my aperitivo in Florence. Check the weather: if conditions are suitable, I will book a table at Terrazze Michelangelo. Indoor alternative: Bar Perseo in Piazza della Signoria.

## Milan dinner.md

### # Milan team dinner

For my team dinner in Milan, the outdoor rooftop of The Dome Restaurant is booked as first option. Check the weather: if rain is expected, switch to the indoor backup at Ristorante Berton, Via M. Bongiorno 13.

# Code Example: Building an MCP-based LLM Agentic System

- Calendar .ics file:

## calendar.ics

April 2026	●
APR	○
26	
<b>Aperitivo in Florence</b>	
Description: N/A	
Start: 26/04/2026, 18:00:00	Location: Florence, Italy
End: 26/04/2026, 20:00:00	Organizer: N/A
April 2026	●
APR	○
28	
<b>Dinner in Milan</b>	
Description: N/A	
Start: 28/04/2026, 20:00:00	Location: Milan, Italy
End: 28/04/2026, 22:30:00	Organizer: N/A

# Code Example: Building an MCP-based LLM Agentic System

- requirements.txt

```
fastmcp==2.13.3
```

```
icalendar
```

```
httpx
```

```
mcp-weather-server
```

```
ollama
```

# Code Example: Building an MCP-based LLM Agentic System

- notes\_server.py

```
from fastmcp import FastMCP
from pathlib import Path

mcp = FastMCP("NotesServer")
VAULT_PATH = Path(__file__).parent # e.g.: Obsidian Vault
```

```
# — Tool: keyword search over Local .md notes
```

```
@mcp.tool()
```

```
def get_notes(query: str = "") -> list[dict]:
```

```
    """
    Searches the local folder (e.g., Obsidian vault) and returns notes
    matching the query keyword.
```

```
    Use when the user asks about plans, event details or venue
    options.
```

```
    Pass an empty string to retrieve all available notes.
```

```
    Args:
```

```
    query (str): Keyword to search across note titles and content.
    Use '' to list all notes.
```

```
    """
```

```
    results = []
    q = query.lower().strip()
    for md_file in VAULT_PATH.glob("*.md"):
        content = md_file.read_text(encoding="utf-8", errors="ignore")
        if not q or q in content.lower() or q in md_file.stem.lower():
            results.append({
                "title": md_file.stem,
                "content": content,
            })
    return results if results else [{"info": f"No notes found for query"}]
```

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

# Code Example: Building an MCP-based LLM Agentic System

- `calendar_server.py` (1 of 3)

```
from fastmcp import FastMCP
from icalendar import Calendar
from datetime import datetime, date, timedelta
from pathlib import Path

mcp = FastMCP("CalendarServer")
ICS = Path(__file__).parent / "calendar.ics"

# — Resource: raw .ics content (read-only) —————
# The Host can inject this directly into the LLM context as background knowledge.

@mcp.resource("calendar://raw")
def raw_calendar() -> str:
    """
    Returns the raw content of the local .ics calendar file as plain text.
    Use as read-only context – for structured event data use get_calendar_events.
    """
    return ICS.read_text(encoding="utf-8", errors="ignore")
```

# Code Example: Building an MCP-based LLM Agentic System

- calendar\_server.py (2 of 3)

```
# — Tool: List events in a date range —————  
  
@mcp.tool()  
def get_calendar_events(date_range: int = 7) -> list[dict]:  
    """  
    Returns calendar events for the next N days from the local .ics calendar file.  
    Use when the user asks about their schedule, appointments or agenda.  
    Args:  
        date_range (int): Number of days to look ahead from today. Default 7.  
    """  
    cal = Calendar.from_ical(ICS.read_bytes())  
    today = date.today()  
    limit = today + timedelta(days=int(date_range))  
    events = []  
    for component in cal.walk():  
        if component.name != "VEVENT":  
            continue  
        dtstart = component.get("DTSTART").dt  
        dt = dtstart.date() if isinstance(dtstart, datetime) else dtstart  
        if today <= dt <= limit:  
            events.append({  
                "date": str(dt),  
                "event": str(component.get("SUMMARY", "No title")),  
                "location": str(component.get("LOCATION", "-")),  
                "description": str(component.get("DESCRIPTION", "")),  
            })  
    return sorted(events, key=lambda e: e["date"])
```

# Code Example: Building an MCP-based LLM Agentic System

- calendar\_server.py (3 of 3)

```
# — Tool: update an existing event's description —————  
  
@mcp.tool()  
def update_event_description(event_summary: str, new_description: str) -> dict:  
    """  
    Updates the description of an existing calendar event.  
    Use after consulting weather data to write a venue recommendation.  
    Args:  
        event_summary (str): Exact title of the event to update,  
        new_description (str): New event description with useful recommendation  
    """  
  
    cal = Calendar.from_ical(ICS.read_bytes())  
    new_cal = Calendar()  
    for key, value in cal.items():  
        if key not in ("BEGIN", "END"):  
            new_cal.add(key, value)  
  
    updated = False  
    for component in cal.walk():  
        if component.name in ("VEVENT", "VTIMEZONE"):  
            if component.name == "VEVENT" and \  
                str(component.get("SUMMARY", "")) == event_summary:  
                if "DESCRIPTION" in component:  
                    del component["DESCRIPTION"]  
                    component.add("DESCRIPTION", new_description)  
                    updated = True  
    new_cal.add_component(component)
```

```
if updated:  
    ICS.write_bytes(new_cal.to_ical())  
    return {"status": "updated", "event": event_summary}  
return {"status": "not_found", "event": event_summary}  
  
if __name__ == "__main__":  
    mcp.run(transport="stdio")
```

# Code Example: Building an MCP-based LLM Agentic System

- host.py (1 of 5)

```
import asyncio, sys, httpx
from datetime import date
from fastmcp import Client
import ollama
```

```
MODEL          = "qwen3.5:9b"
WEATHER_PORT   = 8000
WEATHER_URL    = f"http://127.0.0.1:{WEATHER_PORT}/mcp"
MAX_ITER       = 6
```

```
NEEDED_TOOLS = {
    "CalendarClient_get_calendar_events",
    "CalendarClient_update_event_description",
    "NotesClient_get_notes",
    "WeatherClient_get_current_weather",
    "WeatherClient_get_weather_byDateTimeRange",
}
```

```
SYSTEM_PROMPT = (
    f"You are a personal planning assistant with access to the user's calendar, "
    f"local notes and live weather data. Today's date is {date.today().isoformat()}. "
    f"Use the available tools as needed to fulfill the user's request. "
    f"Be concise. Once the task is complete, produce a short day-by-day summary."
)
```

```
MCP_CONFIG = {
    "mcpServers": {
        "CalendarClient": {"command": "python", "args":
            ["calendar_server.py"]},
        "NotesClient":    {"command": "python", "args":
            ["notes_server.py"]},
        "WeatherClient": {"url": WEATHER_URL},
    }
}
```

# Code Example: Building an MCP-based LLM Agentic System

- `host.py` (2 of 5)

```
# — Weather server startup —————  
# This is a local deployment of an MCP Weather server with remote-style Streamable HTTP transport.
```

```
async def start_weather_server():  
    proc = await asyncio.create_subprocess_exec(  
        sys.executable, "-m", "mcp_weather_server",  
        "--mode", "streamable-http", "--host", "127.0.0.1", "--port", str(WEATHER_PORT),  
        stdout=asyncio.subprocess.DEVNULL, stderr=asyncio.subprocess.PIPE,  
    )  
    print(f"[host] weather server starting (PID {proc.pid})...")  
    deadline = asyncio.get_event_loop().time() + 30  
    async with httpx.AsyncClient() as http:  
        while asyncio.get_event_loop().time() < deadline:  
            try:  
                r = await http.post(WEATHER_URL,  
                    json={"jsonrpc": "2.0", "id": 0, "method": "initialize",  
                        "params": {"protocolVersion": "2024-11-05", "capabilities": {},  
                                "clientInfo": {"name": "probe", "version": "1"}}},  
                    headers={"Content-Type": "application/json",  
                             "Accept": "application/json, text/event-stream"}, timeout=2.0)  
                if r.status_code < 500:  
                    print(f"[host] weather server ready\n"); return proc  
            except Exception: pass  
        await asyncio.sleep(0.5)
```

# Code Example: Building an MCP-based LLM Agentic System

- `host.py` (3 of 5)

```
def to_ollama_schema(tools) -> list[dict]:  
    """Convert FastMCP tool objects to Ollama-compatible OpenAI tool-calling format."""  
    result = []  
    for t in tools:  
        if t.name not in NEEDED_TOOLS: continue  
        schema = dict(t.inputSchema or {"type": "object", "properties": {}})  
        result.append({"type": "function", "function": {  
            "name": t.name, "description": (t.description or "")[:200],  
            "parameters": schema}})  
    return result  
  
def extract_obs(result, tool_name: str) -> str:  
    """Extract and truncate tool result to keep context window manageable."""  
    raw = result.content[0].text if result.content else str(result)  
    name = tool_name.lower()  
    if "bydatetimerange" in name: return raw[:2500] # Longer context for weather forecast  
    if "weather" in name: return raw[:500] # shorter context for current weather  
    return raw[:1000] # other tools
```

# Code Example: Building an MCP-based LLM Agentic System

- `host.py` (4 of 5)

# — ReAct orchestrator loop —

```
async def run(user_message: str) -> str:
    weather_proc = await start_weather_server()
    try:
        async with Client(MCP_CONFIG) as client:
            tools_schema = to_ollama_schema(await client.list_tools())
            print(f"[host] {len(tools_schema)} tools active: "
                  + ", ".join(t['function']['name'] for t in tools_schema) + "\n")
            context = [{"role": "system", "content": SYSTEM_PROMPT},
                       {"role": "user", "content": user_message}]
            seen_calls = set() # keeps track of previously executed tool calls to avoid duplicate calls

            for _ in range(MAX_ITER): # — ReAct Loop —
                msg = ollama.chat(model=MODEL, messages=context, tools=tools_schema)["message"]
                if msg.get("tool_calls"):
                    if msg.get("content"): print(f"[thought] {msg['content']}")
                else:
                    print(f"[final answer] {msg['content']}"); return msg["content"]
                context.append(msg)
                for call in msg["tool_calls"]:
                    name, args = call["function"]["name"], call["function"]["arguments"]
                    key = f"{name}:{sorted(args.items())}"
                    if key in seen_calls:
                        context.append({"role": "tool", "content": f"Already called {name}."})
                        continue
                    seen_calls.add(key); print(f"[action] {name}({args})")
```

# Code Example: Building an MCP-based LLM Agentic System

- host.py (5 of 5)

```
try:
    obs = extract_obs(await client.call_tool(name, args), name)
    print(f"[obs] {obs}\n")
    context.append({"role": "tool", "content": obs})
except Exception as e:
    context.append({"role": "tool", "content": f"Tool error: {e}"})

context.append({"role": "user", "content": "Summarise what you gathered."})
final = ollama.chat(model=MODEL, messages=context)["message"].get("content", "")
print(f"[final answer] {final}"); return final

finally:
    if weather_proc:
        try: weather_proc.terminate(); await weather_proc.wait()
        except ProcessLookupError: pass
        print("[host] weather server stopped")

if __name__ == "__main__":
    user_query = ("Check my upcoming calendar events and the related notes, then write "
                 "in my calendar file a venue recommendation based on the weather current "
                 "conditions or weather forecast for each event.")
    asyncio.run(run(user_query))
```

# Code Example: Building an MCP-based LLM Agentic System

- Output

```
[host] weather server starting (PID 64028)...  
[host] weather server ready at http://127.0.0.1:8000/mcp
```

```
+-----+  
FastMCP 2.13.3  
  
Server name: CalendarServer  
  
Transport:  STDIO  
  
Docs:       https://gofastmcp.com  
Hosting:    https://fastmcp.cloud  
+-----+
```

```
[04/26/26 17:44:23] INFO      Starting MCP server          server.py:1974  
                          'CalendarServer' with transport  
                          'stdio'
```

```
+-----+  
FastMCP 2.13.3  
  
Server name: NotesServer  
  
Transport:  STDIO  
  
Docs:       https://gofastmcp.com  
Hosting:    https://fastmcp.cloud  
+-----+
```

# Code Example: Building an MCP-based LLM Agentic System

## • Output

```
[04/26/26 17:44:24] INFO      Starting MCP server 'NotesServer'  server.py:1974
                             with transport 'stdio'
```

```
[host] 5 tools active:
```

- CalendarClient\_get\_calendar\_events
- CalendarClient\_update\_event\_description
- NotesClient\_get\_notes
- WeatherClient\_get\_current\_weather
- WeatherClient\_get\_weather\_byDateTimeRange

```
[action]  CalendarClient_get_calendar_events({'date_range': 7})
```

```
[obs]     [{"date":"2026-04-26","event":"Aperitivo in Florence","location":"Florence, Italy","description":""}, {"date":"2026-04-28","event":"Dinner in Milan","location":"Milan, Italy","description":""}]
```

```
[action]  NotesClient_get_notes({'query': ''})
```

```
[obs]     [{"title":"Milan dinner","content":"# Milan team dinner\n\nFor my team dinner in Milan, the outdoor rooftop terrace at Terrazza Aperol is booked as the first option. Check the weather forecast before confirming: if rain is expected, switch to the indoor backup at Ristorante Berton, Via Mike Bongiorno 13.\n"}, {"title":"Florence event","content":"# Florence aperitivo\n\nI have to book a venue for my aperitivo in Florence. Check the weather: if conditions are suitable, I will book a table at Terraz
```

```
[action]  WeatherClient_get_current_weather({'city': 'Florence'})
```

```
[obs]     The weather in Florence is Clear sky with a temperature of 22.8°C (feels like 19.3°C), relative humidity at 27%, and dew point at 2.9°C. Wind is blowing from the W at 13.0 km/h with gusts up to 28.8 km/h. Atmospheric pressure is 1018.1 hPa with 0% cloud cover. Visibility is 48.5 km.
```

```
[action]  WeatherClient_get_weather_byDateTimeRange({'city': 'Florence', 'start_date': '2026-04-26', 'end_date': '2026-04-26'})
```

```
[obs]     Please analyze the following JSON weather forecast information and generate a comprehensive report.
```

# Code Example: Building an MCP-based LLM Agentic System

- Output

```
=== WEATHER DATA ===
```

```
{  
  "city": "Florence",  
  "latitude": 43.77925,  
  "longitude": 11.24626,  
  "start_date": "2026-04-26",  
  "end_date": "2026-04-26",  
  "weather_data": [  
    { "time": "2026-04-26T00:00",  
      "temperature_c": 8.5,  
      "humidity_percent": 76}]
```

```
[action] WeatherClient_get_weather_byDateTimeRange({'city': 'Milan', 'start_date': '2026-04-28', 'end_date': '2026-04-28'})
```

```
[obs] Please analyze the following JSON weather forecast information and generate a comprehensive report.
```

```
=== WEATHER DATA ===
```

```
{  
  "city": "Milan",  
  "latitude": 45.46427,  
  "longitude": 9.18951,  
  "start_date": "2026-04-28",  
  "end_date": "2026-04-28",  
  "weather_data": [  
    { "time": "2026-04-28T00:00",  
      "temperature_c": 16.6,  
      "humidity_percent": 59}]
```

# Code Example: Building an MCP-based LLM Agentic System

## • Output

```
[action] CalendarClient_update_event_description({'event_summary': 'Aperitivo in Florence', 'new_description': 'Weather: Clear sky, 22.8°C. OUTDOOR venue recommended. Book table at Terrazza (outdoor rooftop).'})
```

```
[obs] {"status": "updated", "event": "Aperitivo in Florence"}
```

```
[action] CalendarClient_update_event_description({'event_summary': 'Dinner in Milan', 'new_description': 'Weather: Clear sky, 16.6°C. OUTDOOR venue recommended. Book rooftop terrace at Terrazza Aperol.'})
```

```
[obs] {"status": "updated", "event": "Dinner in Milan"}
```

```
[thought] **Calendar Events Updated with Weather-Based Venue Recommendations:**
```

```
✓ **2026-04-26: Aperitivo in Florence (Florence, Italy)**
```

```
- Weather: Clear sky, 22.8°C, low humidity (27%)
```

```
- **Recommendation:** OUTDOOR – Book table at Terrazza (outdoor rooftop)
```

```
✓ **2026-04-28: Dinner in Milan (Milan, Italy)**
```

```
- Weather: Clear sky, 16.6°C
```

```
- **Recommendation:** OUTDOOR – Book rooftop terrace at Terrazza Aperol
```

# Code Example: Building an MCP-based LLM Agentic System

## • Output

---

**\*\*Summary:\*\***

All forecasted weather conditions are favorable for outdoor dining. Both events have been updated with recommended outdoor venue locations. No indoor backup reservations are required for these dates.

[final answer] **\*\*Calendar Events Updated with Weather-Based Venue Recommendations:\*\***

✓ **\*\*2026-04-26: Aperitivo in Florence (Florence, Italy)\*\***

- Weather: Clear sky, 22.8°C, low humidity (27%)

- **\*\*Recommendation:\*\*** OUTDOOR – Book table at Terrazza (outdoor rooftop)

✓ **\*\*2026-04-28: Dinner in Milan (Milan, Italy)\*\***

- Weather: Clear sky, 16.6°C

- **\*\*Recommendation:\*\*** OUTDOOR – Book rooftop terrace at Terrazza Aperol

---

**\*\*Summary:\*\***

All forecasted weather conditions are favorable for outdoor dining. Both events have been updated with recommended outdoor venue locations. No indoor backup reservations are required for these dates.

[host] weather server stopped

# Code Example: Building an MCP-based LLM Agentic System

- calendar.ics (updated)

April 2026 ●

APR ○

26

**Aperitivo in Florence**

**Description:** Weather: Clear sky, 22.8°C. OUTDOOR venue recommended. Book table at Terrazza (outdoor rooftop).

**Start:** 26/04/2026, 18:00:00 **Location:** Florence, Italy

**End:** 26/04/2026, 20:00:00 **Organizer:** N/A

April 2026 ●

APR ○

28

**Dinner in Milan**

**Description:** Weather: Clear sky, 16.6°C. OUTDOOR venue recommended. Book rooftop terrace at Terrazza Aperol.

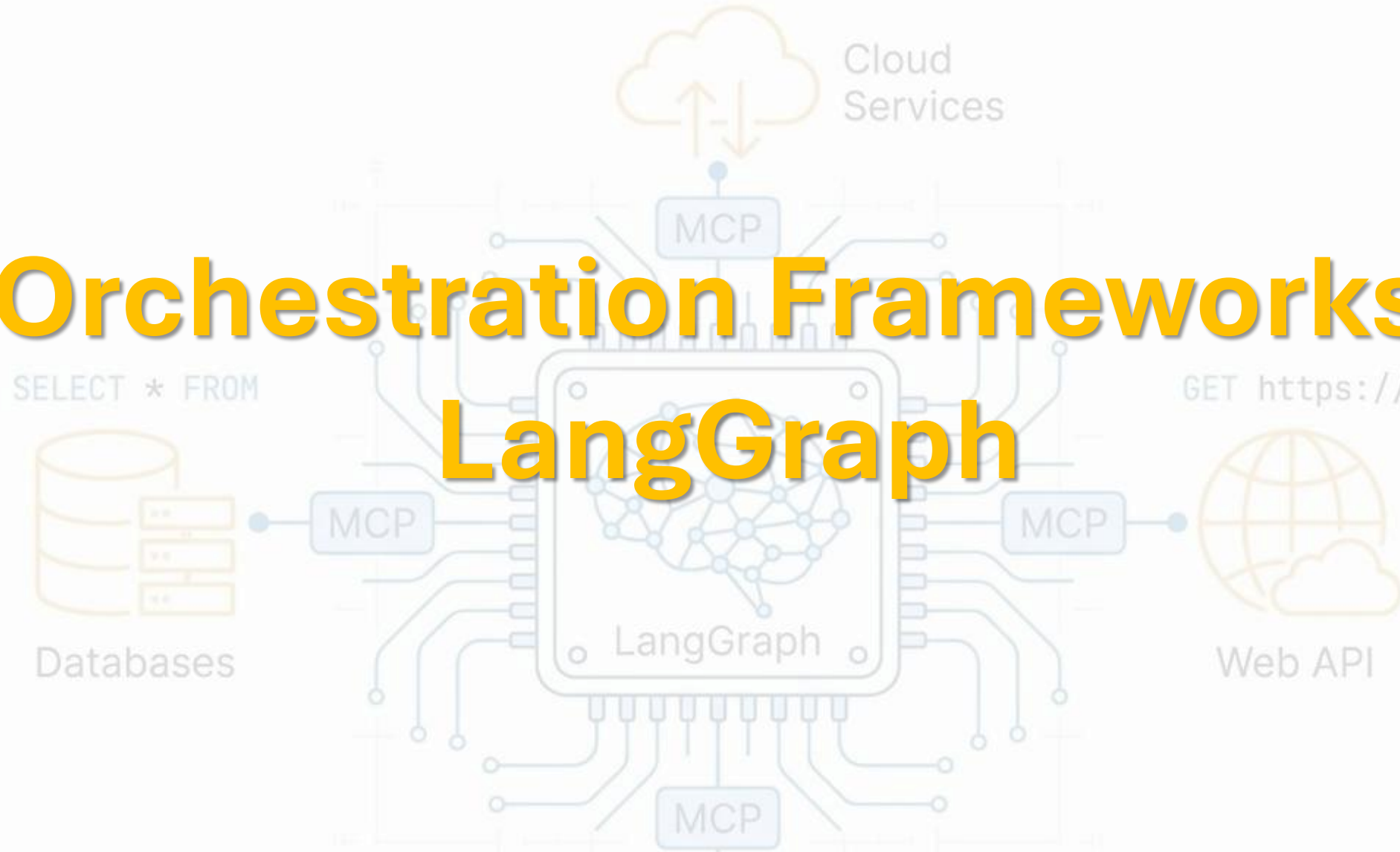
**Start:** 28/04/2026, 20:00:00 **Location:** Milan, Italy

**End:** 28/04/2026, 22:30:00 **Organizer:** N/A

# Outline

- AI Agents: Definitions & Motivations
- The Roadmap to AI Agents
  - ❖ *Reasoning*
  - ❖ *Acting: Tool calling*
  - ❖ *Reasoning + Acting: the ReAct framework*
- Taxonomies and Design Patterns of AI Agents
- The Model Context Protocol: MCP
- **Orchestration Frameworks: LangGraph** ←

# Orchestration Frameworks: LangGraph



# LangGraph Core Concepts: State, Nodes and Edges

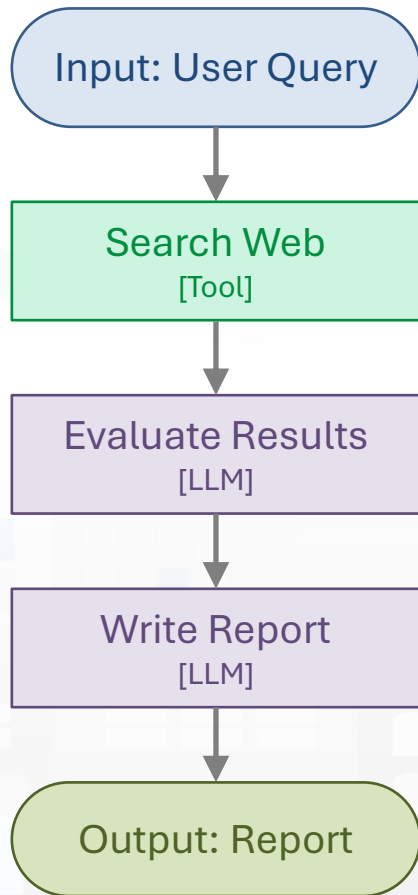
- **LangGraph** is a low-level framework, developed by **LangChain**, for orchestrating stateful AI agents as graphs
- **LangChain** allow sequential chains only: no loops, no conditional branches, no shared state between calls Each call is stateless, therefore the context must be manually passed
- **LangGraph** allows to model and build multi-step agent workflows as explicit directed graph, i.e., the StateGraph, supporting cycles, conditional edges, and iterative loops natively.
  - **Nodes** are individual units of logic, action or processing steps, i.e., any Python function (sync or async) that takes the state and returns a partial update. Can invoke LLMs, execute tools, interact with human-in-the-loop, or apply logic.
  - **Edges** are transitions defining the flow between nodes, including conditional branching, and loops instead of single prompt-tool chains.
  - **State** (TypedDict): A Python TypedDict shared across all nodes. Each node receives the current state, returns a partial update, and LangGraph merges it automatically. The state acts as shared memory, preserving context across nodes.

# Why LangGraph: From (ReAct) Loops to Graphs

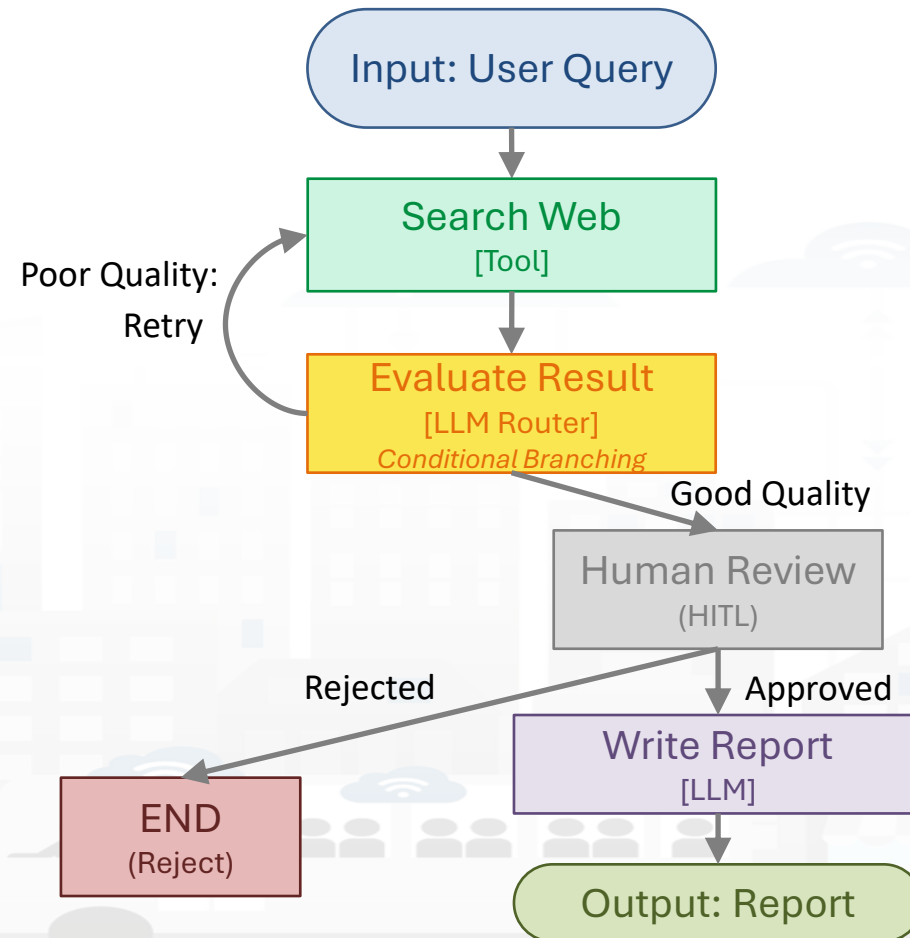
- ✗ ReAct loops, though implemented with MCP, are suitable for simple tasks: single-hop reasoning, a few tool calls. Not suitable for more complex tasks or multi-agent contexts, which requires more expressive logic.
- ✓ **LangGraph** models the agent workflow as a typed state graph (TypedDict) with named nodes, conditional edges, and native support for cycles, branching, and multi-actor coordination, enabling arbitrarily complex orchestration patterns.
- ✗ Manual hard-coded ReAct loops aggregate two responsibilities: the LLM decides **WHAT** to do AND the code decides **HOW** to proceed: no separation of concerns.
- ✓ **LangGraph** allow separation of concerns structurally: the LLM reasons *within* each node, the graph topology decides *which* node runs next
- ✗ In raw ReAct loops, the developer must translate manually the JSON schema for every tool into the proper format to be sent to the LLM (e.g., OpenAI format)
- ✓ Adopting **LangGraph** allows to automatically derive the correct LLM payload format.

# LangGraph Core Concepts: State, Nodes and Edges

## LangChain: Sequential Chain



## LangGraph: Directed Graph, Conditional Branches, Loops



# LangGraph vs LangChain

## LangChain — Sequential Chains:

Fixed linear pipeline:

No loops, no conditional branches, no shared state between calls

Each call is stateless — the context must be manually passed as a string

## LangGraph — Stateful Directed Graph:

Workflow = directed graph of nodes (LLM calls, tools, human checkpoints)

Supports cycles, conditional edges, and iterative loops natively

A TypedDict state is shared across all nodes — updated at each transition

## When to use LangGraph vs LangChain:

LangChain: suitable for fixed, sequential pipelines (e.g., RAG, summarisation)

LangGraph: required when the workflow involves branching, loops, or HITL

# LangGraph Core Concepts: State, Nodes and Edges

- **StateGraph:** The root object: `StateGraph(MyState)` — declares the state schema and owns all nodes and edges
- **State (TypedDict):** A Python `TypedDict` shared across all nodes. Each node receives the current state, returns a partial update, and LangGraph merges it automatically.
- **Nodes:** Any Python function (or async routine) that takes the state and returns a partial update. Can invoke LLMs, execute tools, prompt humans, or apply logic.
- **Edges:**
  - `graph.add_edge(A, B)` — unconditional: always go from A to B after A completes
  - `graph.add_conditional_edges(A, fn, map)` — conditional: `fn(state)` returns a key, `map` routes to the destination node
- **Compile and run:**
  - `graph = builder.compile()` — validates the graph and returns a runnable executor.
  - Invoke with `graph.invoke(state)` for synchronous execution, `graph.ainvoke(state)` for asynchronous execution, stream with `graph.astream(state)`.

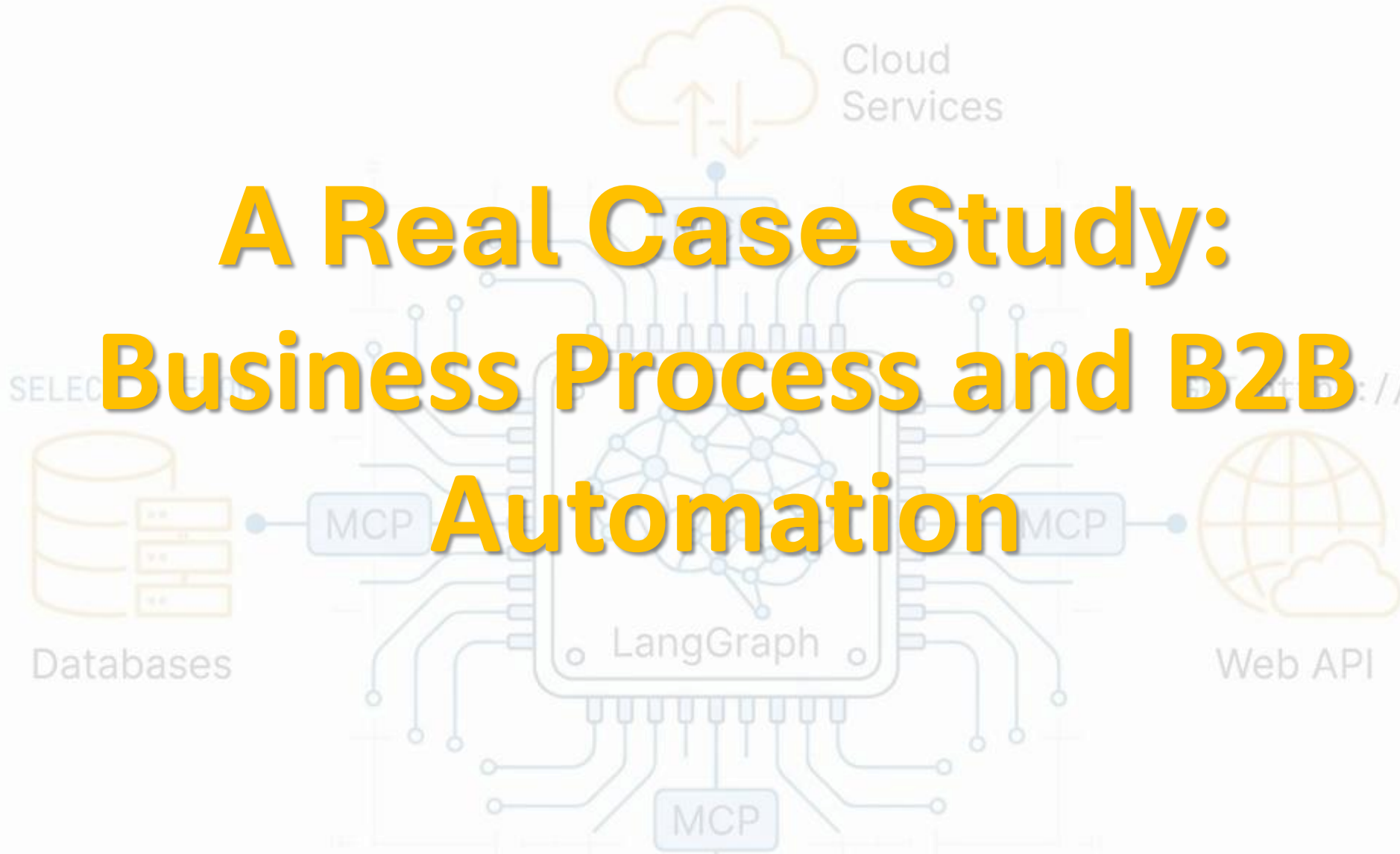
# LangGraph (Code Example): State & Nodes

```
# — 1. State —  
from langgraph.graph import StateGraph, START, END  
from langgraph.types import interrupt, Command  
from typing_extensions import TypedDict, Literal  
  
class ResearchState(TypedDict):  
    query: str  
    search_results: list # to append instead of replacing, use reducers such as: Annotated[list, operator.add]  
    quality: Literal["good", "poor"]  
    approved: bool  
    report: str  
  
# — 2. Nodes —  
def search_web(s: ResearchState):  
    results = web_search_tool.run(s["query"]) # MCP tool  
    return {"search_results": results}  
  
def evaluate_result(s: ResearchState):  
    verdict = llm.invoke("Rate quality (answering with one word: good or poor) of: " + str(s["search_results"]))  
    q = "good" if "good" in verdict.content.lower() else "poor"  
    return {"quality": q}  
  
def human_review(s: ResearchState):  
    decision = interrupt(s["search_results"]) # suspends graph, await user's response (HITL)  
    return {"approved": decision == "approve"}  
  
def write_report(s: ResearchState):  
    report = llm.invoke("Write report on: " + str(s["search_results"]))  
    return {"report": report.content}
```

# LangGraph (Code Example): Routing, Wiring & Compile

```
# — 3. Routing functions —————  
def route_quality(s: ResearchState):  
    return "search_web" if s["quality"] == "poor" else "human_review"  
  
def route_human(s: ResearchState):  
    # interrupt() stored the decision in state["approved"]  
    return "write_report" if s.get("approved") else END  
  
# — 4. Build & wire the graph —————  
g = StateGraph(ResearchState)  
g.add_node("search_web", search_web)  
g.add_node("evaluate_result", evaluate_result)  
g.add_node("human_review", human_review)  
g.add_node("write_report", write_report)  
  
g.add_edge(START, "search_web"); g.add_edge("search_web", "evaluate_result")  
g.add_edge("write_report", END)  
g.add_conditional_edges("evaluate_result", route_quality, {"search_web": "search_web", "human_review": "human_review"})  
g.add_conditional_edges("human_review", route_human, {"write_report": "write_report", END: END})  
  
# — 5. Compile & run —————  
from langgraph.checkpoint.memory import MemorySaver  
app = g.compile(checkpointer=MemorySaver())  
cfg = {"configurable": {"thread_id": "1"}} # checkpointer thread  
  
result = app.invoke({"query": "Agentic Frameworks based on Large Language Models"}, cfg) # runs until interrupt()  
print(result.interrupts) # payload shown to human evaluator  
final = app.invoke(Command(resume="approve"), cfg) # or 'reject'
```

# A Real Case Study: Business Process and B2B Automation

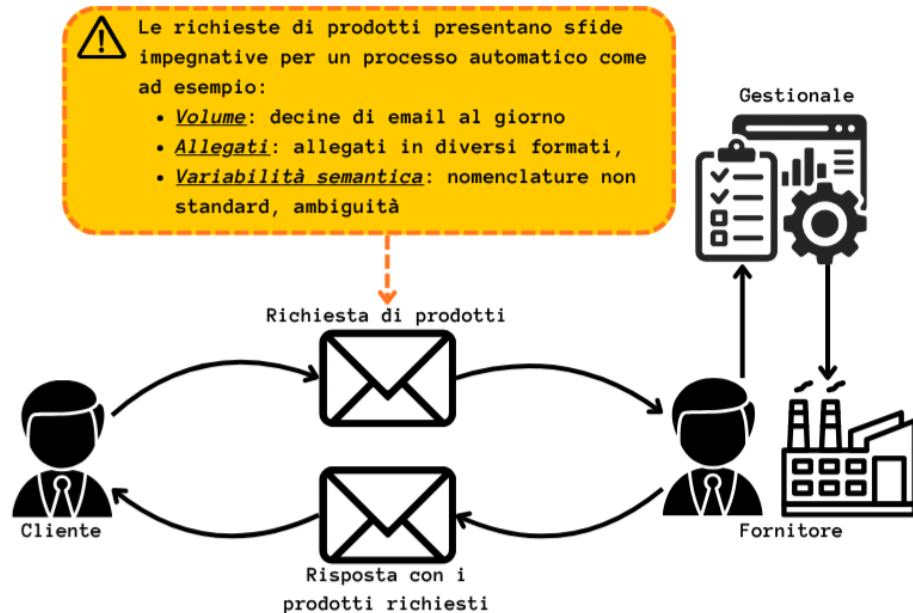


# A Real Case Study: Business Process and B2B Automation

**Goal:** Automating the management of B2B commercial communications: Implement an agentic system that automates the process of replying to emails sent by customers who want to place orders with a supplier of small mechanical components, or request product availability from the catalog.

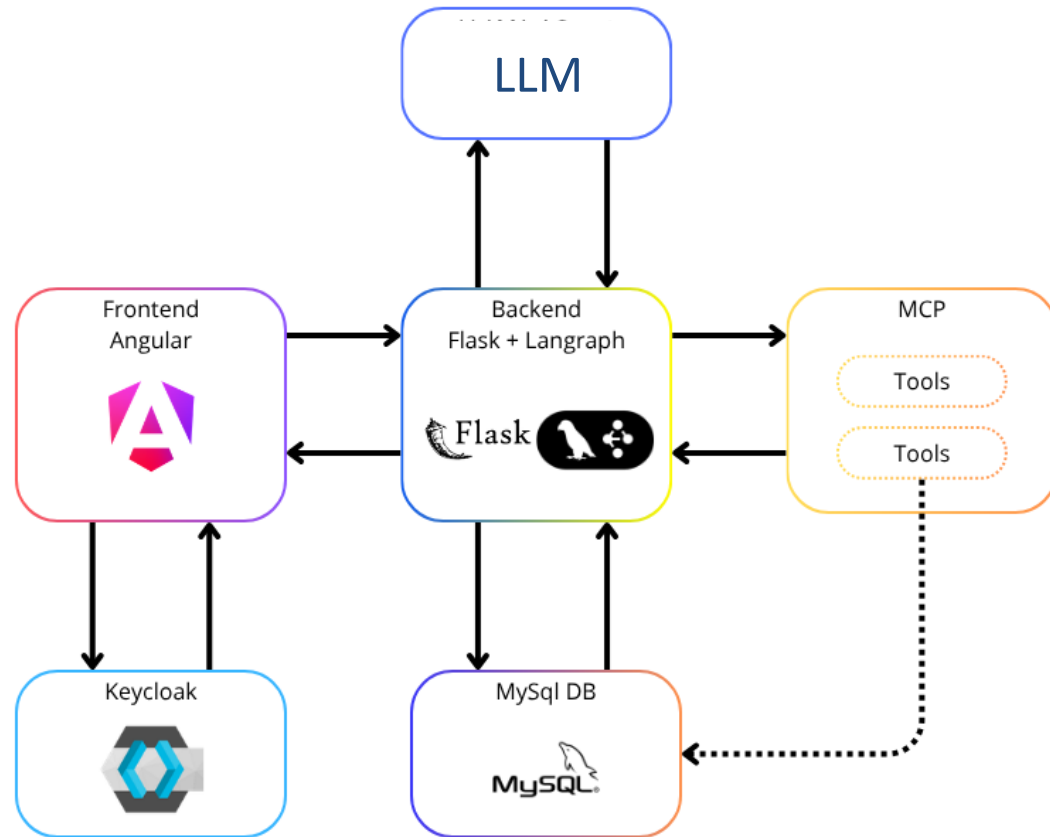
## Challenges:

- This is a process that is still almost entirely manual: operators have to read every email received, interpret technical specifications of the products requested, consults the management system and responds with prices and availability.
- A difficult process to automate due to the great semantic variability of nomenclatures, the need for text disambiguation, the high volume of emails received, attachments in various formats.



# A Real Case Study: Business Process and B2B Automation

## System Architecture



- **Frontend (Angular):** real-time communication via Socket.IO, Keycloak + JWT authentication
- **Backend (Flask + LangGraph):** orchestrates the agent graph and propagates context between layers
- **MCP Server (FastMCP via HTTP/SSE):** exposes the two instruments to the model, keeps the FAISS indexes in memory
- **Keycloak (Autenticazione):** Manage users authentication and access
- **MySQL (Database):** Save interactions and maintain the product catalog
- Nvidia H100 96GB NVL x2 for Local LLM Inference
- Each layer is containerized independently (Docker Compose) and can be updated without stopping the system

# A Real Case Study: Business Process and B2B Automation

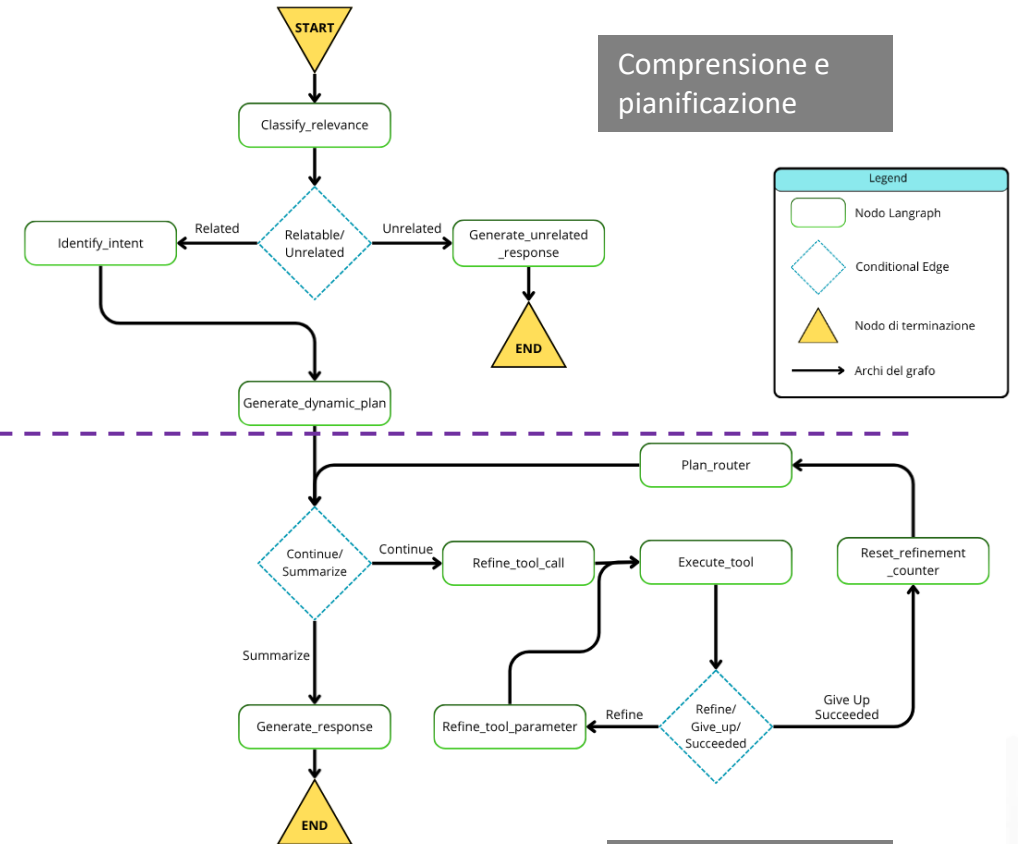
## LangGraph Implementation

- classify\_relevance:** filter spam e email out of context.
- identify\_intent:** identify order / invoice / general question.
- generate\_dynamic\_plan:** produces the ordered list of tool calls to be executed, inferred from the text.

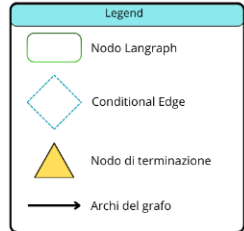
The model doesn't search directly for products: first it understands what the customer wants, then it builds an explicit and verifiable plan.

- refine\_tool\_call:** resolves the product family through FAISS before querying the DB.
- execute\_tool:** call *query\_catalog* via MCP, remove step from list.
- check\_tool\_result:** if the result is empty or incorrect it calls **refine\_tool\_parameters** (max 2 attempts).
- generate\_final\_response:** synthesizes the real results of the DB in natural language.

The retry sub-cycle automatically handles ambiguities and overly strict parameters, without interrupting the processing of other items.



Comprensione e pianificazione



Esecuzione e autocorrezione

# A Real Case Study: Business Process and B2B Automation

## MCP Tools

### find\_best\_category

- **Input:** Natural Language Product Description
- Encoding with SentenceTransformers → FAISS research
- **Output:** Top-3 product families
- Handle semantic variability:  
«Sbarra» = «BARRA FILETTATA»

### query\_catalog

- **Input:** Family + Technical Parameters (Diameter, Length, Material...)
- Query SQL → Relaxed fallback if no results
- **Output:** Semantic re-ranking on returned candidates
- It ensures that prices and availability always come from the DB, never inferred from the model

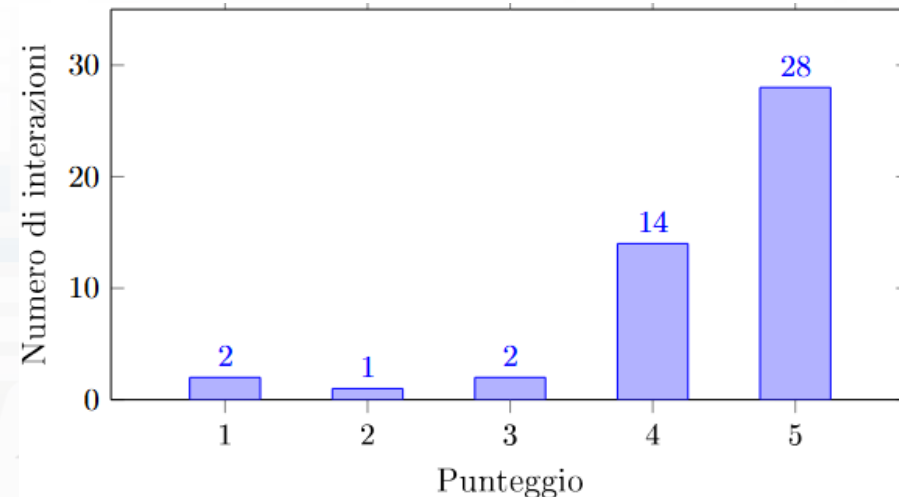
# A Real Case Study: Business Process and B2B Automation

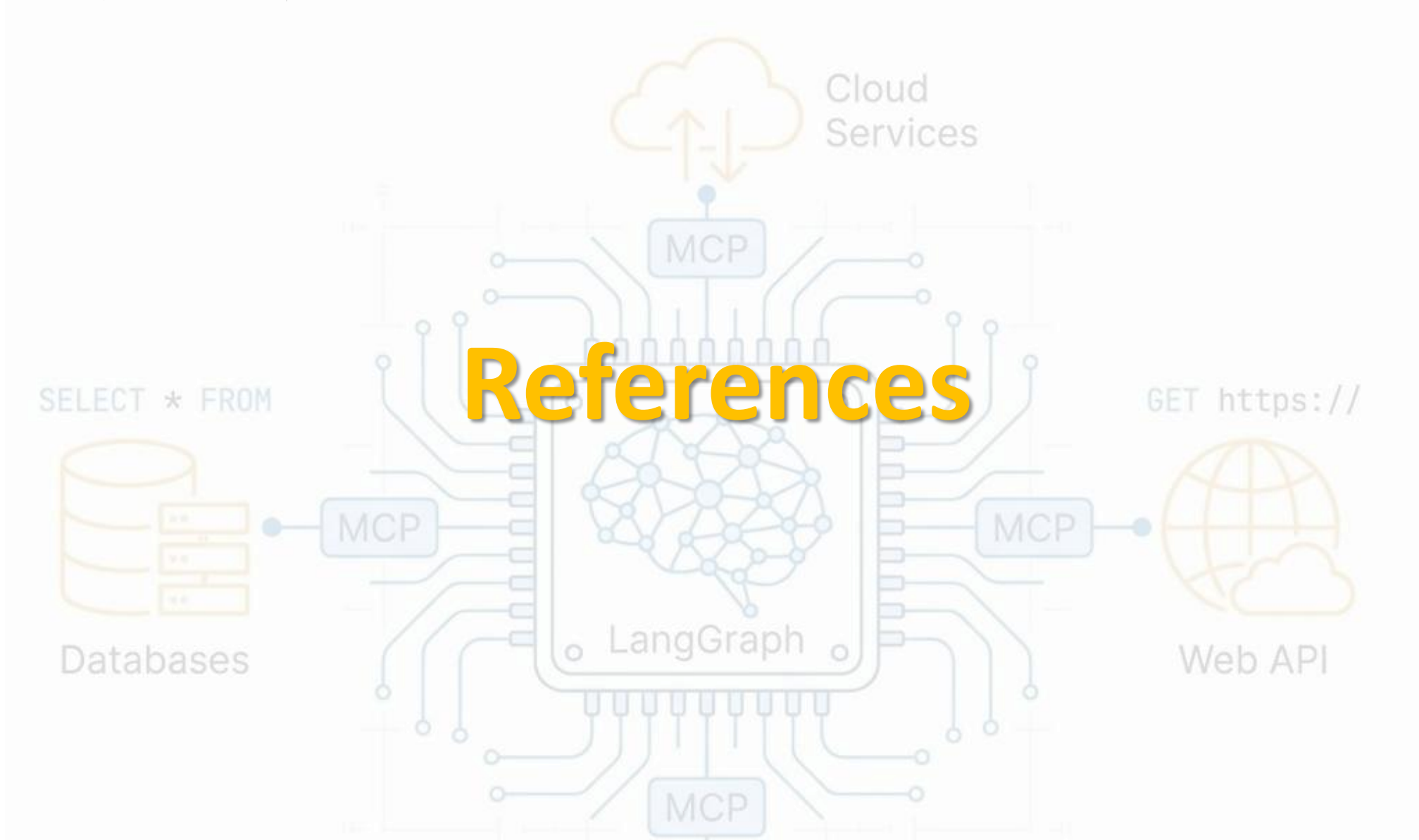
## Validation Results

### Accuracy of system responses

- **Score 5:** Intent understood, correct response without modification.
- **Score 4:** small imperfections, directly usable answer.
- **Score 3:** ~20% corrections, mostly ambiguous articles.
- **Score 1-2:** very informal terminology or products absent from the catalog.

- **Average score:** 4,38/5
- **Score  $\geq 4$ :** 89,4%
- **Critical cases:** 6,4%





# References

- [Anthropic, 2024a]** “Building effective agents”. Online: <https://www.anthropic.com/engineering/building-effective-agents>
- [Anthropic, 2024b]** “Introducing the Model Context Protocol”. Online: <https://www.anthropic.com/news/model-context-protocol>
- [Besta et al., 2024]** Besta, M., Blach, N., Kubicek, A., Gerstenberger, R., Gianinazzi, L., Gajda, J., Lehmann, T., Podstawski, M., Niewiadomski, H., Nyczyk, P., & Hoefler, T., “Graph of Thoughts: Solving Elaborate Problems with Large Language Models”, in Proceedings of the AAIL Conference on Artificial Intelligence, 38(16), 17682–17690. arXiv:2308.09687, 2024.
- [Kojima et al., 2022]** Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., & Iwasawa, Y., “Large Language Models are Zero-Shot Reasoners”, in Proceedings of the 36<sup>th</sup> International Conference on Neural Information Processing Systems (NIPS'22). Article No.: 1613, Pages 22199 - 22213. arXiv:2205.11916, 2022.
- [OECD, 2026]** “The Agentic AI Landscape and its Conceptual Foundations”, OECD Artificial Intelligence Papers, No. 5, February 2026. [https://www.oecd.org/content/dam/oecd/en/publications/reports/2026/02/the-agentic-ai-landscape-and-its-conceptual-foundations\\_a9d4b451/396cf758-en.pdf](https://www.oecd.org/content/dam/oecd/en/publications/reports/2026/02/the-agentic-ai-landscape-and-its-conceptual-foundations_a9d4b451/396cf758-en.pdf)
- [Russell & Norvig, 1995]** Russell, S. J., & Norvig, P. (1995). Artificial Intelligence: A Modern Approach. Prentice Hall
- [Shinn et al., 2023]** Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S., “Reflexion: Language Agents with Verbal Reinforcement Learning”. In Proceedings of the 37th Conference on Neural Information Processing Systems (NeurIPS 2023). arXiv:2303.11366, 2023.
- [Wei et al., 2022]** Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D., “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”, in Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS'22). Article No.: 1800, Pages 24824 – 24837, 2022. arXiv:2201.11903, 2022.

# References

**[Wooldridge & Jennings, 1995]** Wooldridge, M. and Jennings, N. R., “Intelligent agents: theory and practice”, The Knowledge Engineering Review, Vol. 10:2,115-152, 1995.

**[Yao et al., 2023a]** Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T.L., Cao, Y., & Narasimhan, K., “Tree of Thoughts: Deliberate Problem Solving with Large Language Models”, in Proceedings of the 37<sup>th</sup> International Conference on Neural Information Processing Systems (NIPS’23). Article No.: 517, Pages 11809 – 11822. arXiv:2305.10601, 2023.

**[Yao et al., 2023b]** Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y., “ReAct: Synergizing reasoning and acting in language models”, in Proceedings of the 11th International Conference on Learning Representations (ICLR 2023). arXiv preprint arXiv:2210.03629v3, 2023.