Building Production-Ready AI Agents with FastAPI, Pydantic-AI & MCP

Petros Savvakis

whoami_

** **Lead Software Engineer** @ Ethniki Asfalistiki — Tech Thirsty… shipping cloud-native FastAPI + K8s micro-services

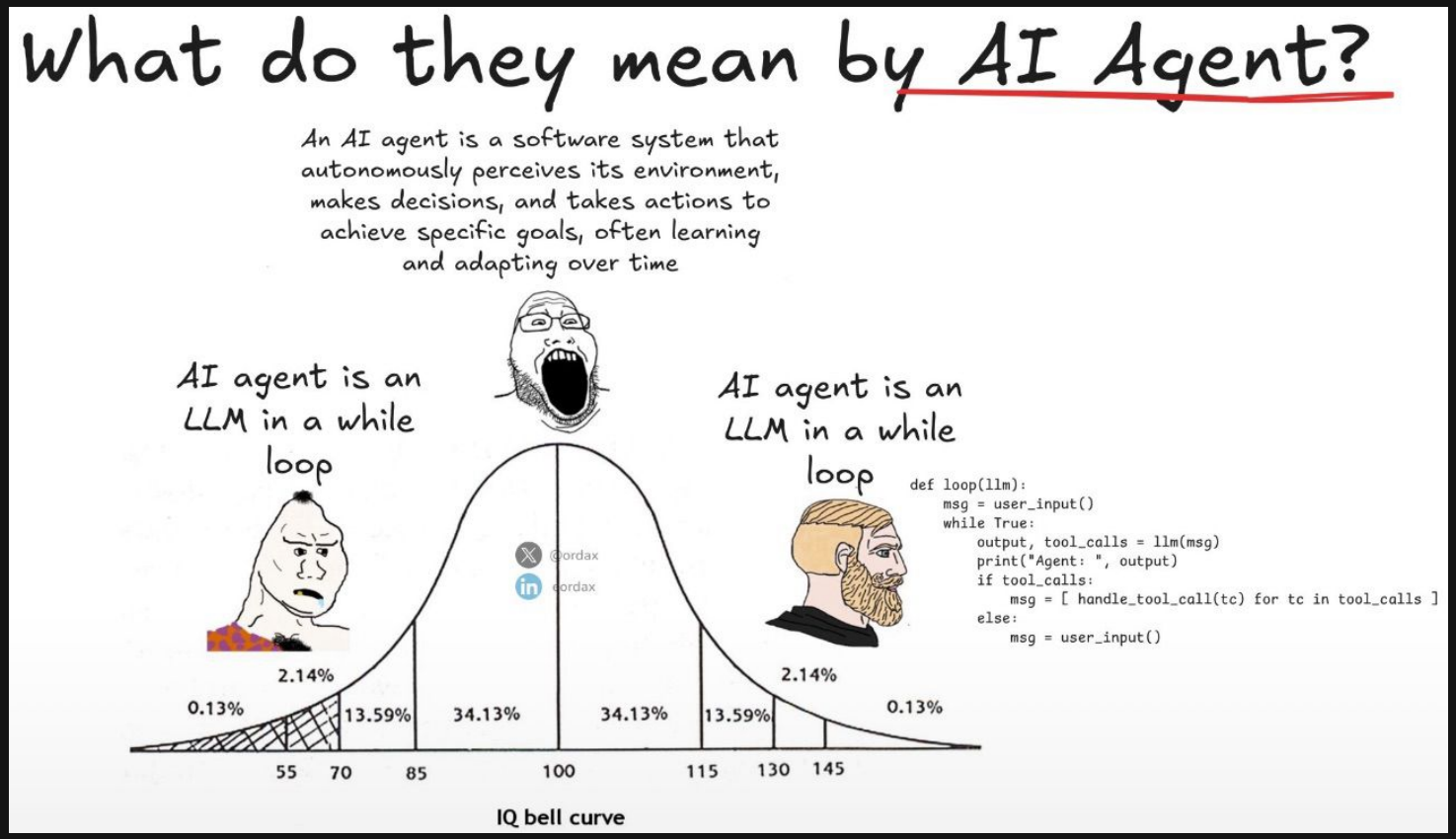
** **MSc Robotics & Electrical Engineering**; ex-PCB hacker-designer

** **OSS tinkerer & blogger** — respectablyAI, PeepDB, writing about tech at petrostechchronicles.com

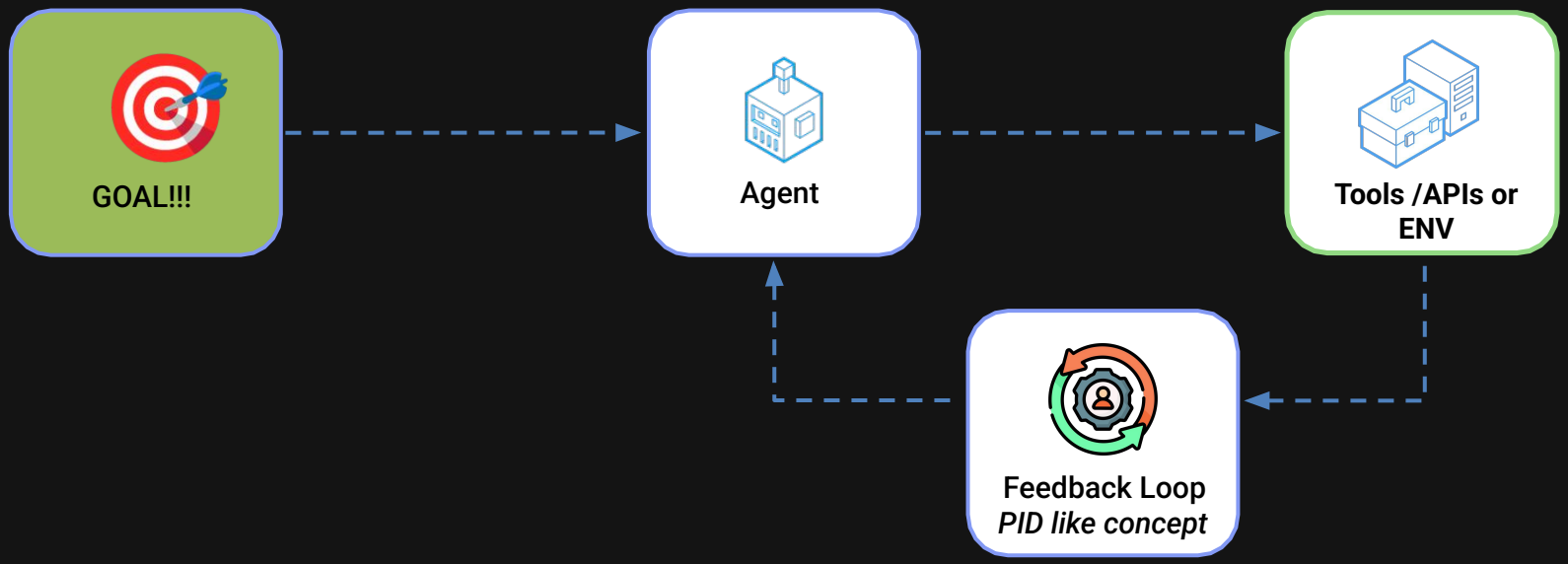
why?

WHY DO WE NEED AI AGENTS IN GENERAL?

Agents???


What do they mean by AI Agent?
An AI agent is a software system that autonomously perceives its environment, makes decisions, and takes actions to achieve specific goals, often learning and adapting over time
AI agent is an LLM in a while loop
AI agent is an LLM in a while loop
def loop(llm):
msg = user_input()
while True:
output, tool_calls = llm(msg)
print("Agent: ", output)
if tool_calls:
msg = [handle_tool_call(tc) for tc in tool_calls]
else:
msg = user_input()
@ordax
@ordax
0.13%
2.14%
13.59%
34.13%
34.13%
13.59%
2.14%
0.13%
55
70
85
100
115
130
145
IQ bell curve

what is AI Agent?



Why do we need Agents?

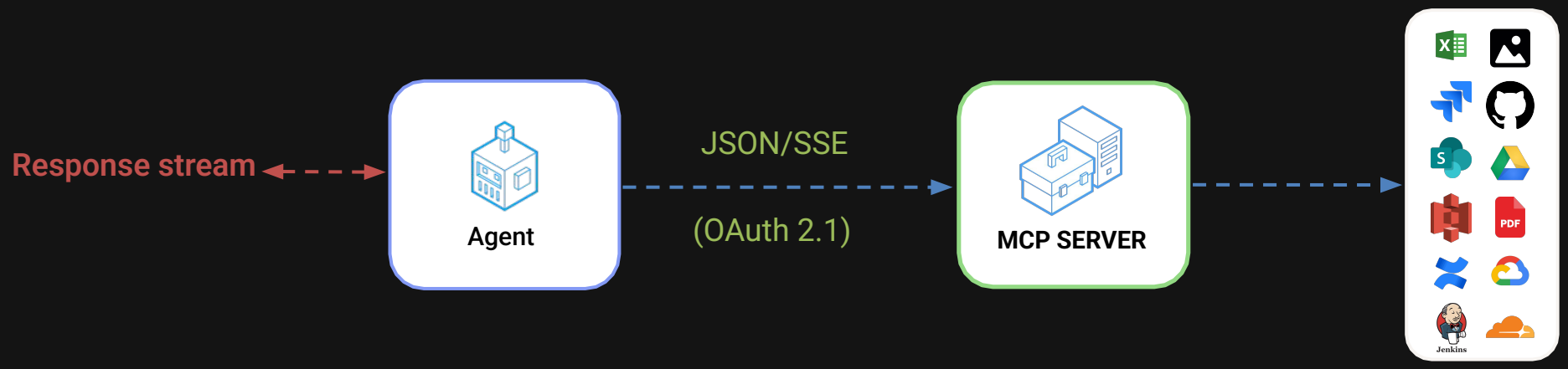
- Execute workflows autonomously
- Make decisions based on context
- Maintain conversation history
- Can use external tools

Picture an AI assistant that retains context and executes actions—that's the essence of an agent.

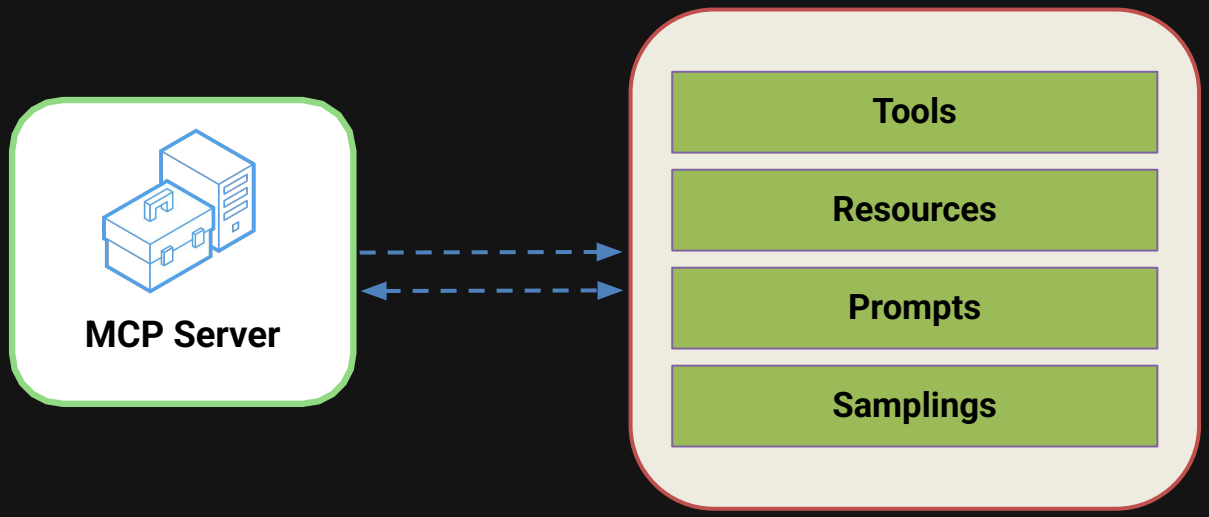
What is MCP(Model Context Protocol) ?

- **Open protocol for AI** ↔ tool handshakes (HTTP + JSON/SSE)
- **Secure *(partially)* - using** (OAuth 2 + RBAC)
- **Language/model agnostic clients** (Python, JS, CLI)
- **Ready-made adapters for GitLab, Confluence, Jira, Slack, DBs**

why do we need MCP?



MCP Server-tools can be...



Tools (“*get_weather*”):

```python
import json, requests

# ❶ Describe & expose a callable tool on the server
GET_WEATHER = {
    "name": "get_weather",
    "description": "Return current weather for a city",
    "inputSchema": {
        "type": "object",
        "properties": {"location": {"type": "string"}},
        "required": ["location"],
    },
}
# (server would put this in the list returned by tools/list)

# ❷ Model decides to call the tool
payload = {
    "jsonrpc": "2.0",
    "id": 2,
    "method": "tools/call",
    "params": {"name": "get_weather", "arguments": {"location": "New
York"}},
}
resp = requests.post("http://mcp-server.example/tools/call", json=payload)
print(resp.json())
```

Resources (*project file*):

```python
import requests, pprint, json

# List everything the server has exposed
res_list = requests.post(
    "http://mcp-server.example/resources/list",
    json={"jsonrpc": "2.0", "id": 1, "method": "resources/list"},
).json()["resources"]

# Grab the first Rust source file
rust_uri = next(r["uri"] for r in res_list if r["mimeType"] == "text/x-
rust")
contents = requests.post(
    "http://mcp-server.example/resources/read",
    json={
        "jsonrpc": "2.0",
        "id": 2,
        "method": "resources/read",
        "params": {"uri": rust_uri},
    },
).json()["content"]

pprint.pp(contents.splitlines()[:10])
```

Prompts (*re-usable code_review*):

```python
import requests, json

# Discover reusable prompt templates
prompts = requests.post(
    url="http://mcp-server.example/prompts/list",
    json={"jsonrpc": "2.0", "id": 1, "method": "prompts/list"},
).json()["prompts"]

# Fill the “code_review” template with the user’s snippet
hydrated = requests.post(
    url="http://mcp-server.example/prompts/get",
    json={
        "jsonrpc": "2.0",
        "id": 2,
        "method": "prompts/get",
        "params": {
            "name": "code_review",
            "arguments": {"code": "def hello():\n    print('world')"},
        },
    },
).json()["messages"]  # ready-to-drop chat messages
```

Samplings (*server-initiated completion*):

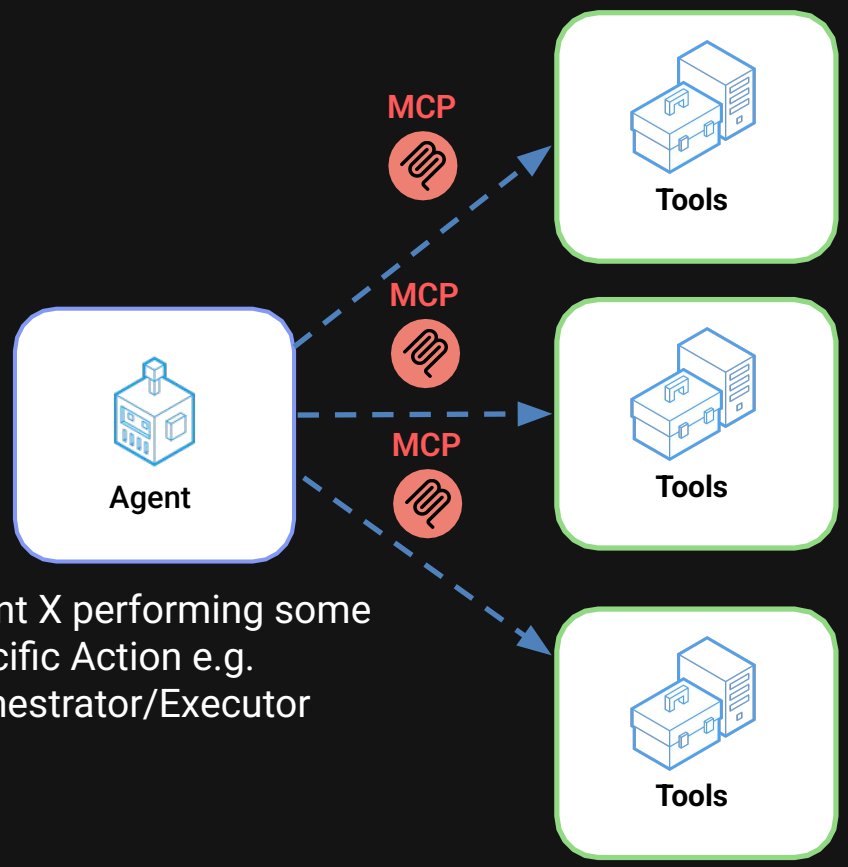
```python
import requests, json

sampling_req = {
    "jsonrpc": "2.0",
    "id": 1,
    "method": "sampling/createMessage",
    "params": {
        "messages": [
            {"role": "user", "content": {"type": "text", "text": "What’s the capital of France?"}}
        ],
        "systemPrompt": "You are a helpful assistant.",
        "modelPreferences": {
            "hints": [{"name": "claude-3-sonnet"}],
            "intelligencePriority": 0.8,
        },
    },
}

answer = requests.post("http://mcp-server.example/sampling/createMessage",
json=sampling_req).json()
print(answer["result"]["choices"][0]["message"]["content"]["text"])
```

MCP server transport options

Transport	Best-fit use case	How it works	Extra notes
stdio (local / “spawn & pipe”)	When the client can launch the server as a subprocess on the same machine *(e.g., VSCode, Cursor, local CLI tools)*.	JSON-RPC messages flow over stdin → stdout; each message is newline-delimited UTF-8.	Easiest to support; every MCP client should implement it.
Server-Sent Events (SSE)	Legacy remote transport for long-running cloud servers where you want true streaming but haven’t upgraded yet.	Two HTTP endpoints: POST for requests, long-lived GET that returns Content-Type: text/event-stream for streaming responses and server-initiated notifications.	Still widely supported, but being phased out in favor of Streamable HTTP.
Streamable HTTP	Recommended remote transport for new deployments *(Cloudflare Workers, FastAPI, etc.)*.	Single HTTP endpoint that supports: • POST for client → server messages • Optional SSE stream (same URL) for server → client messages.	Supersedes SSE; supports resumable streams and simplifies firewall / CORS setup.

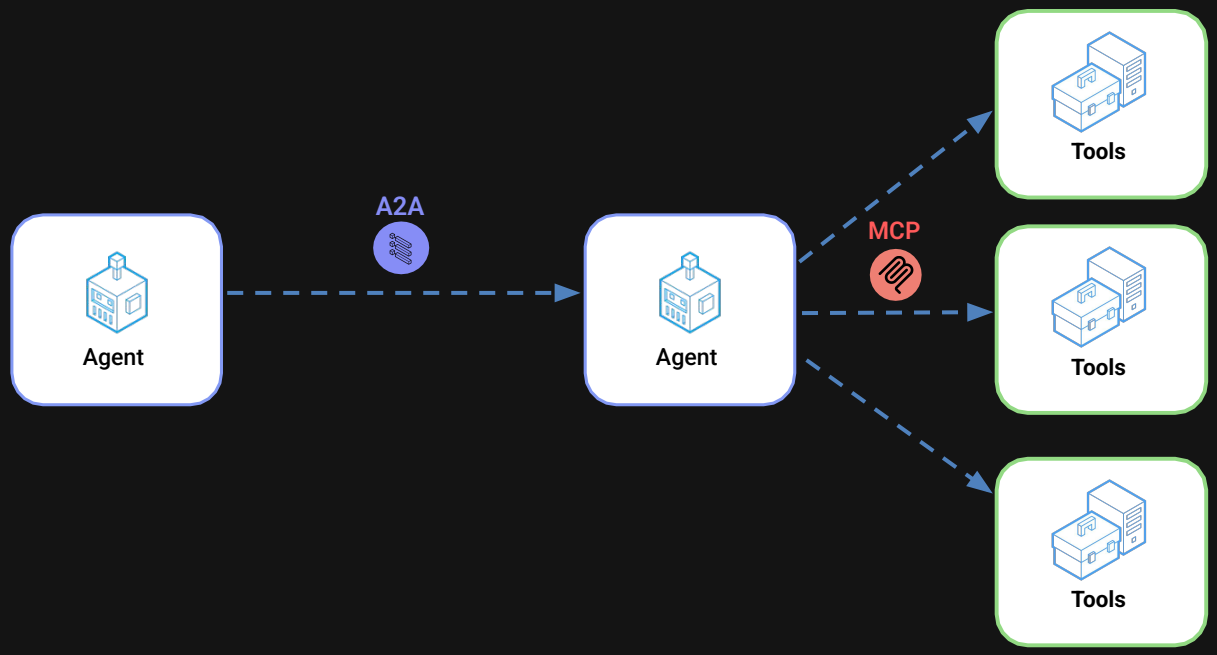
Fusing Agents with MCP

Agent with MCP Communication



Agent X performing some specific Action e.g. Orchestrator/Executor

We also need A2A… for complex pipelines



What is A2A(Agent2Agent) ?

- **Open Agent-to-Agent protocol** for direct communication, task delegation & real-time result streaming between heterogeneous AI agents (HTTP + JSON-RPC/SSE)
- Agents publish a discoverable "**Agent Card**" (ID, skills, endpoints) so peers can auto-discover and negotiate work
- **Shared security model** OAuth 2 / scoped keys with signed messages—to keep cross-vendor traffic safe and auditable
- Enables **multi-agent "swarming"** workflows that complement MCP's agent-to-tool layer (plan → execute → verify) without a central orchestrator

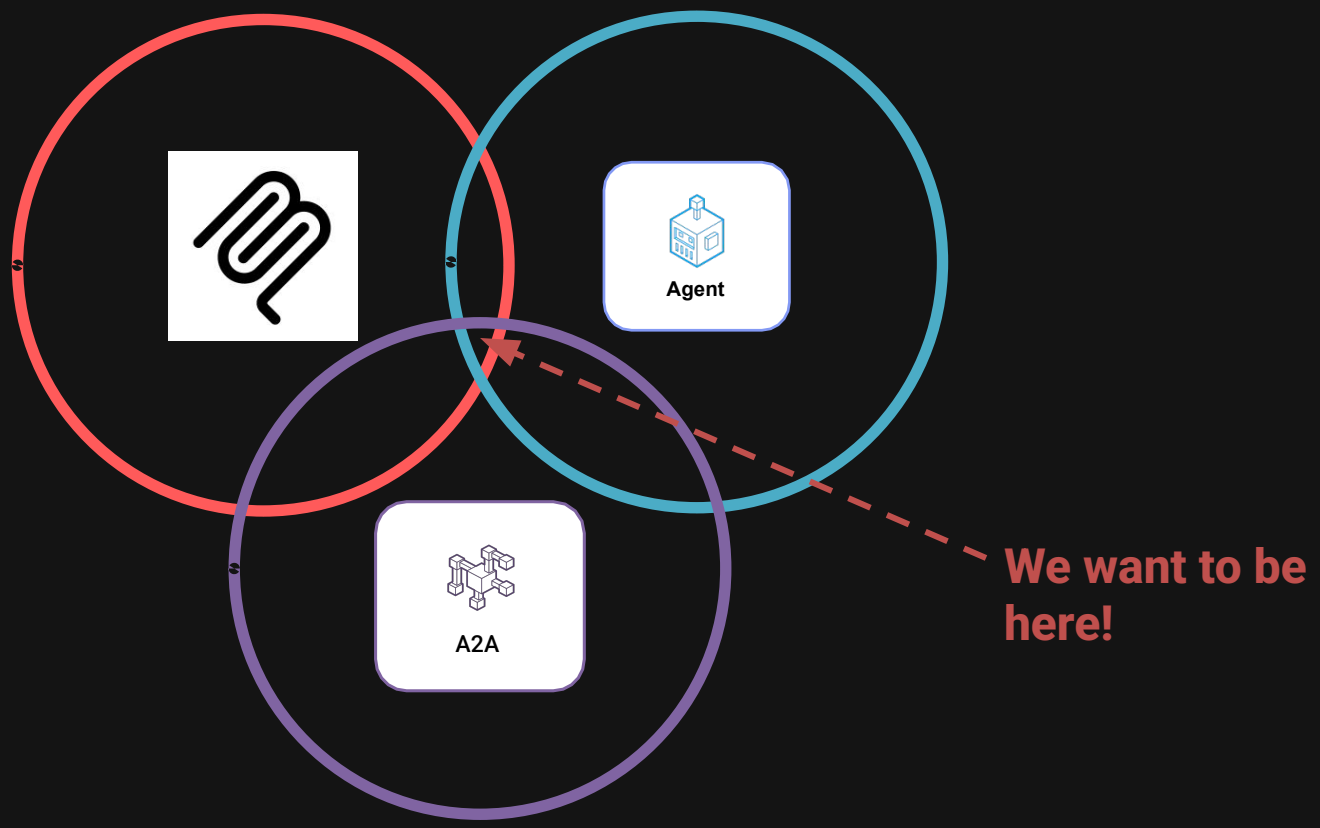
Why do we need A2A to connect multiple Agents?

- Secure Collaboration
- Task and State Management between Agents
- UX Negotiation
- Capability Discovery

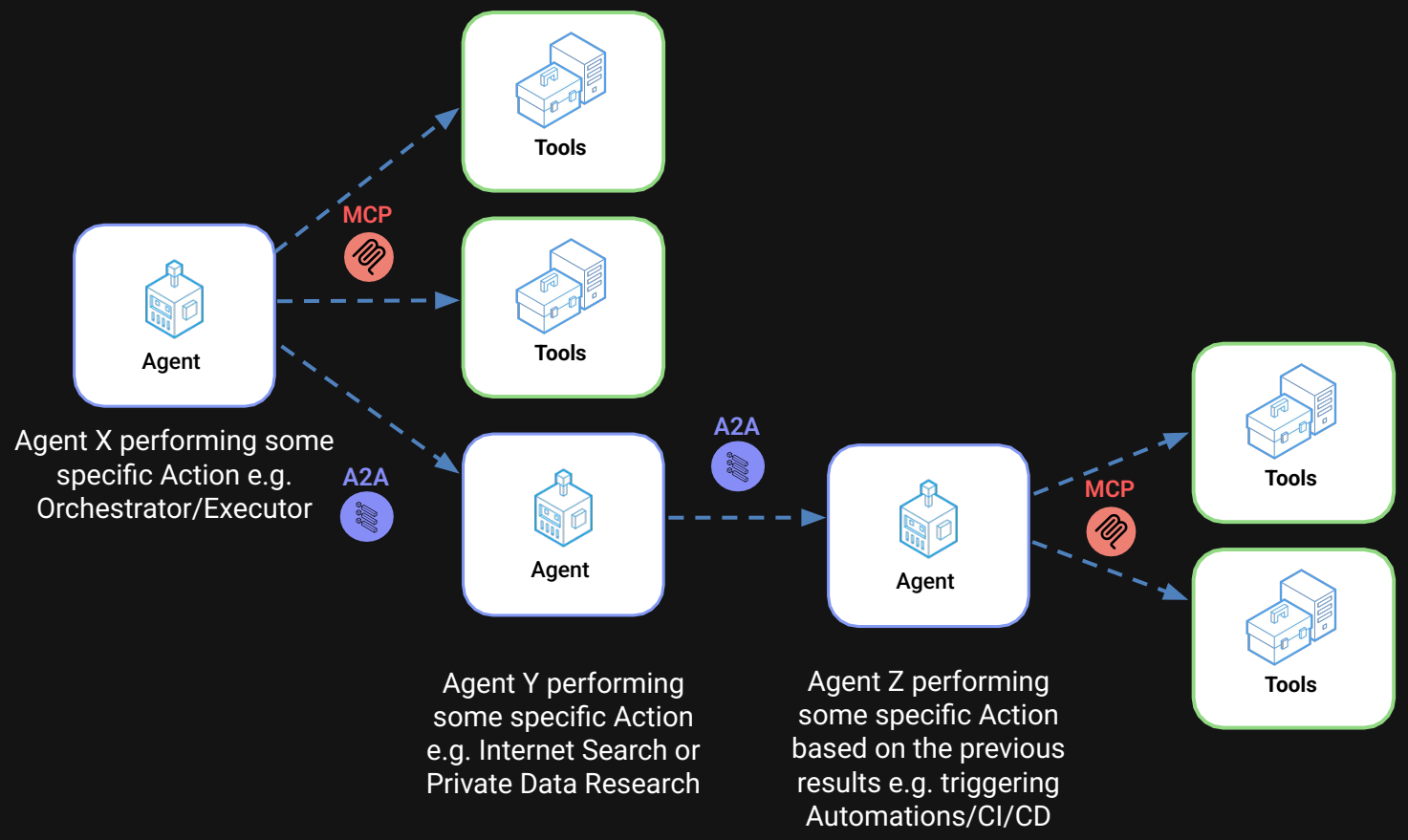
Clarifying why we need each protocol...

Feature / Protocol	MCP	A2A
Focus Area	Context Sharing	Peer Task Collaboration
Type	Context Protocol	Communication Protocol
Best Use Case	Multi-model memory sharing	Decentralized agent operations
Scalability	High with MCP servers	High in P2P networks
Complexity	High	Moderate
Standardization	Evolving	Emerging *(more early stage than MCP)*
Security Layers	Context visibility control *(poor performance security wise)*	Authenticated exchanges

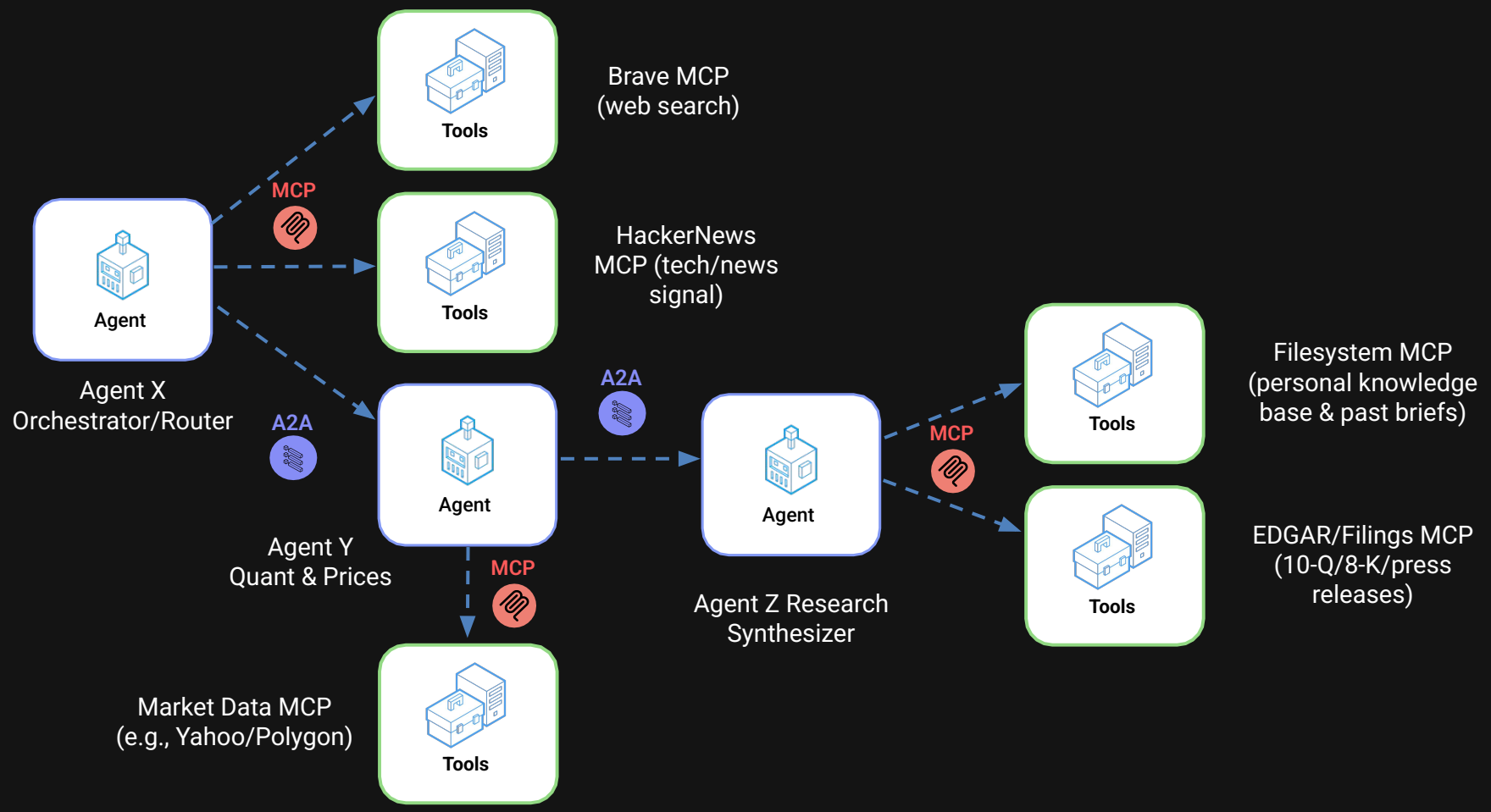
Fusing Agents with MCP + A2A



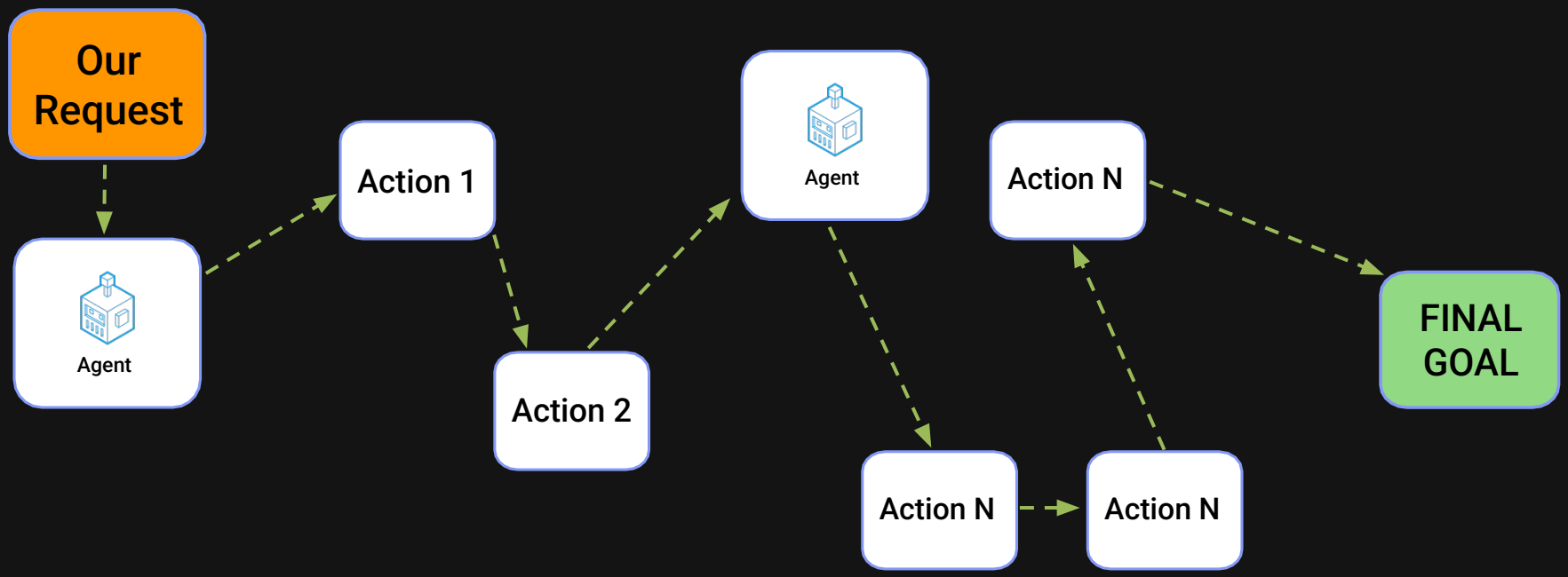
MCP + A2A multi-Agent Communication Pipeline



Pilot Use Case

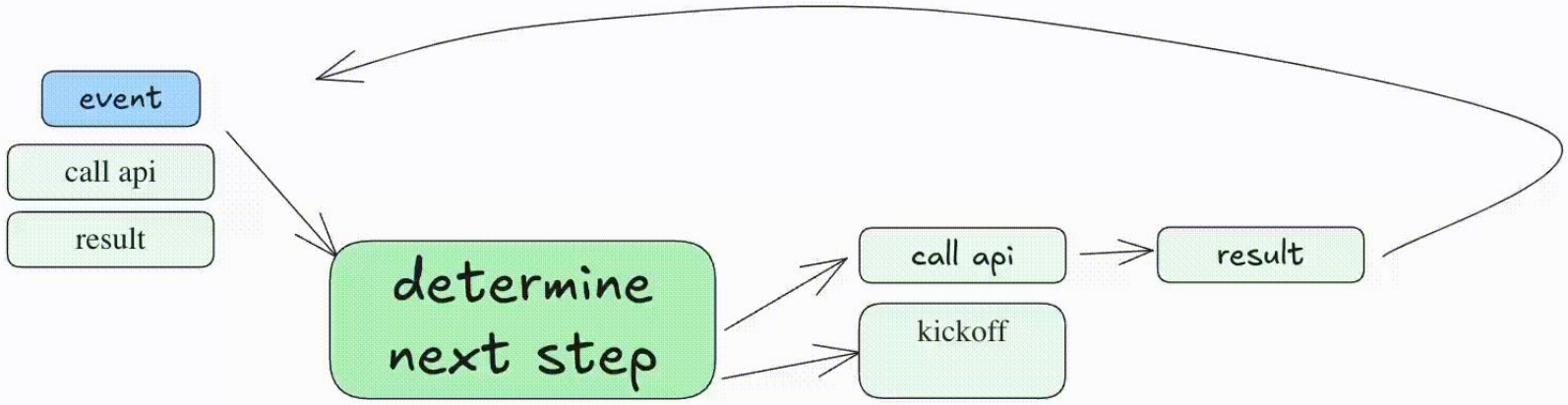


What we hope to achieve…



In reality...

In reality...



Source: *https://github.com/humanlayer/12-factor-agents/blob/main/img/027-agent-loop-animation.gif*

Now that we have an idea about all that…

Let’s build something more plausible…

Each small agent handles a focused task (e.g. summarisation or classification), making the overall system easier to debug and scale

- Structured tool calls & schemas
- Own your prompts & context
- Deterministic control flow & logging
- Human-in-the-loop triggers

Let’s have 90 seconds FastAPI Explanation!

FastAPI is a modern, fast (high-performance) web framework for building APIs in Python.

It’s built on Starlette and Pydantic, so you get high speed and automatic validation

Key Features include:

- Very high performance (comparable to Node.js and Go or *at least trying*😅)
- Standards-based: uses OpenAPI and JSON Schema for automatic interactive docs
- Fast to code with editor autocompletion and fewer bugs

Let’s now pair it with Pydantic AI 🥳

PydanticAI

Multi-Agent Framework
With Validation

What is Pydantic AI?

- Pydantic AI is a Python agent framework that brings the “FastAPI feeling” (type-safety, great DX, automatic validation) to Gen-AI app development
- Built and maintained by the core Pydantic team the same validation layer trusted by OpenAI, Anthropic, LangChain, etc.

Why bother using it?

- **Structured output ⇄ LLM flexibility:** Define a Pydantic model → Pydantic AI guides the LLM to emit JSON that matches it → auto-parses & validates every run (no regex hacks).
- **Model-agnostic:** Works with OpenAI, Anthropic, Gemini, DeepSeek, Ollama, Groq, Cohere, Mistral—and you can plug in any new model with a tiny adapter
- **First-class observability:** Plugs straight into Pydantic Logfire for real-time debugging and usage metrics
- **Type-safe & async-friendly:** Static type-checkers catch mistakes; supports synchronous & asynchronous runs out-of-the-box.

{LET'S START CODING REAL EXAMPLES...}

Thank you!!!

Feel free to connect with me at **Ln**:

or feel free to visit my blog:

Or even better come and chat with me!!! 😀

What it takes to run a pipeline like this locally?

1) Here get data valid to say what resources you need to run llms benchmarks etc
2) Display benchmarks from Huggin or other sources of papers from https://arxiv.org/ with state of the art benchmarks on what HW resources it you need to run big models locally or on cloud H100's

NOT INCLUDED YET

Say about Logfire?

1) Include one slide to explain what logfire is and how to quick use it?
2) Show how I add it in code and what I use it for(tokens usage, logging etc)?
3) Show the UI where logs and tokens used are displayed?

NOT INCLUDED YET

Caveats and Security risks

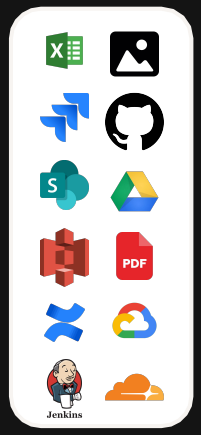
1) A slide saying that llms and agents and mcp servers have many cybersec problems now.
2) Another slide displaying Docker posts with MCP flaws - llms cyber sec flaws
3) Why use in your agents things like E2B MCP for code execution in (firecracker) micro VMs (explain maybe why those tools are needed)

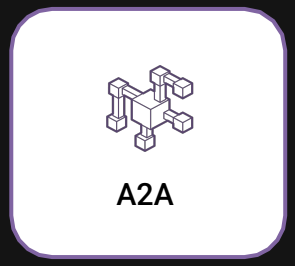
NOT INCLUDED YET

Icons

MCP

A2A





SURVEY TIME!!!

Let’s see what audience believes about AI 🔥