Agents, MCPs, & LangGraph

Linus A. Schneider, Jaisidh Singh, Robin Ruff, Mamen Chembakasseril,
Matthias Kümmerer, Peter Gehler

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Tübingen AI Center

MINERVA

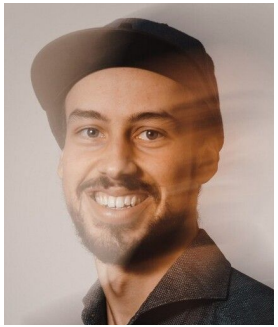
Who we are

Peter Gehler

Matthias Kümmerer

Linus A. Schneider

Jaisidh Singh



Robin

Mamen

What to expect

- A high-level introduction to agentic LLM systems
- Hands-on tutorials for getting started with some important frameworks
- at 5:30pm, you should be able to talk to your own PDFs

What not to expect

- An introduction to LLMs
- Theory
- An overview of research about agents

Overview

- TODO: fill with correct overview
- Introduction to **agentic systems**
- **hands-on**: GitHub Repo, API Keys, getting set up
- Introduction to **LangGraph**
- **hands-on**: building a ...
-
- Introduction to **MCP**
- **Orchestration of Ag**
- coffee break
- **hands-on**:
- time permitting: showcasing some examples from the audience

What are agents?

Remember the early days of ChatGPT?

ChatGPT could answer complicated questions about all sorts of topics, write poems, ... but

- questions about current events were answered completely wrong
  - because they were not included in the training data
- No reasoning
  - it could not reason about how to proceed with solving a problem
- No handling of PDF files and images
  - only text input was allowed
- no image generation
- no personalization
  - it didn't remember anything from past conversations

Things people did with LLMs

- make LLMs call tools (ReAct, ToolFormer, ViperGPT, ...)
- give models access to data (REALM, DPR, RETRO, InstructRetro, ...) Retrieval augmented generation: Give models access to additional databases for better responses

LLMs using tools

- ReAct (2022): tool usage via few-shot promps and reasoning
- ToolFormer (2023):
-
- Retrieval augmented generation: Give models access to additional databases for better responses

Agents are not a new thing

- ChatGPT transformed into an agent over time
  - Launched (Nov 2022)
  - Calling Wolfram Alpha (Apr 2023)
  - Bing & Webbrowsing (Mid 2023)
  - Memory (2024)
  - Deep Research (April 2025)
  - ChatGPT Agent (July 2025)

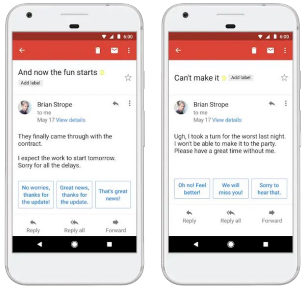
What is an agent?

- A system that independently accomplishes tasks on your behalf
    - It **leverages an LLM** to **manage workflow execution** and **make decisions**. It recognizes when a workflow is complete and can proactively correct its actions if needed. In case of failure, it can halt execution and transfer control back to the user.
    - It has access to various **tools** to interact with external systems—both to **gather context** and to **take actions**—and dynamically selects the appropriate tools depending on the workflow's current state, always operating within clearly defined guardrails.

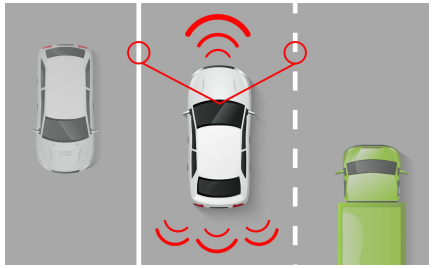
~ OpenAI, a practical guide to building agents

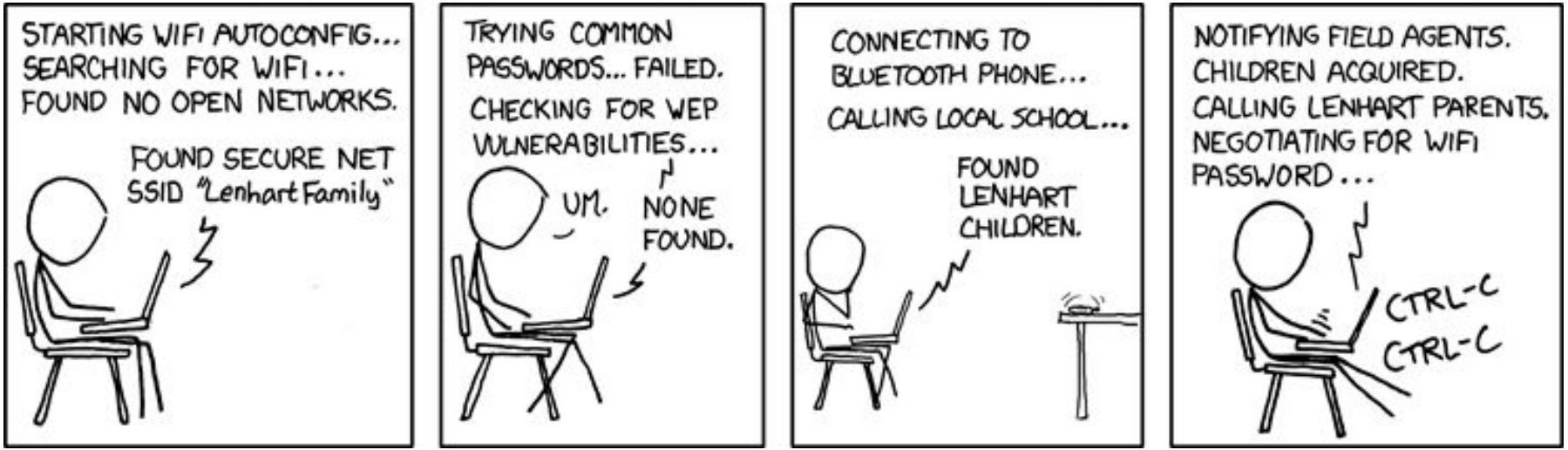
- you know it's an agent when you see it
- Agency is a spectrum, it's not binary

Agent or not?


And now the fun starts
Brian Strope
They finally came through with the contract.
I expect the work to start tomorrow.
Sorry for all the delays.
No worries, thanks for the update!
Great news, thanks for the update.
That's great news!
Reply
Reply all
Forward
Can't make it
Brian Strope
Ugh, I took a turn for the worst last night. I won't be able to make it to the party. Please have a great time without me.
Oh no! Feel better!
We will miss you!
Sorry to hear that.
Reply
Reply all
Forward


J.A.R.V.I.S
19:40





https://xkcd.com/416/, April 2008

Agentic Concepts

- tools: function calling
- has access to ressources
- has prompts(?)

Example

I see the issue. The GeoJSON file structure has different column names than what the script is expecting. Let's examine the actual structure of your GeoJSON file and fix the `static_map_generator.py` script accordingly.

```
head -20 9nt8-h7nd.geojson | grep -v "geometry"
```

Let's try a different approach to inspect the GeoJSON file:

```
python3 -c "import geopandas as gpd; df = gpd.read_file('9nt8-h7nd.
geojson'); print(df.columns.tolist()); print(df.head(1).to_dict('records')
[0].keys())"
```

Now I understand the issue! The GeoJSON file uses `boroname` instead of `boro_name`. Let's fix the `static_map_generator.py` script:

✓ Read `static_map_generator.py`, lines 58 to 68

`static_map_generator.py` +9 -4

Now let's also check if there might be other column name issues in the script:

› Searched text for `ntaname` (`**/static_map_generator.py`), 20 results

Great! The column name `ntaname` exists in both the script and the actual GeoJSON file, so no more changes are needed for those references.

Let's also check if there are any NTAS neighborhoods defined in the NTAS_COLORS dictionary that don't match the actual neighborhood names in the data:

```
python3 -c "import geopandas as gpd; df = gpd.read_file('9nt8-h7nd.
geojson'); manhattan = df[df['boroname'] == 'Manhattan']; print(manhattan
['ntaname'].tolist())"
```

Now we can see the mismatch between the neighborhood names in the script's `NTAS_COLORS` dictionary and the actual names in the data. Let's update the script with the correct neighborhood names:

✓ Read `static_map_generator.py`, lines 21 to 45

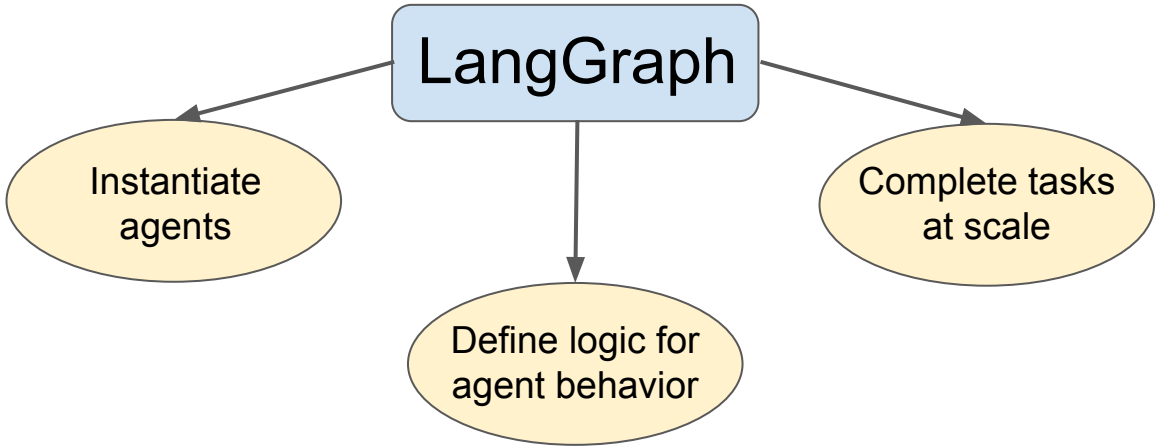
Examples

- Deep Research/ari (you.com)
- Perplexity
- Devin
- Claude Code/Codex
- Cursor/Copilot/replit
- jules(google)
- Wiso/Law firms/
- ChatGPT in general these days
- manus.im
- Google astra
- figma/loveable (originally GPT engineer)
- NotebookLM

Building agents with LangGraph

What is LangGraph?

- LangGraph is an *agent orchestration framework* developed by LangChain, a company that creates frameworks and tools for agent-driven systems.



Heard of LangGraph already?

- Internally uses “SQL-Bot” to write SQL from natural language.
- “SQL-Bot” is a multi-agent system made with LangGraph

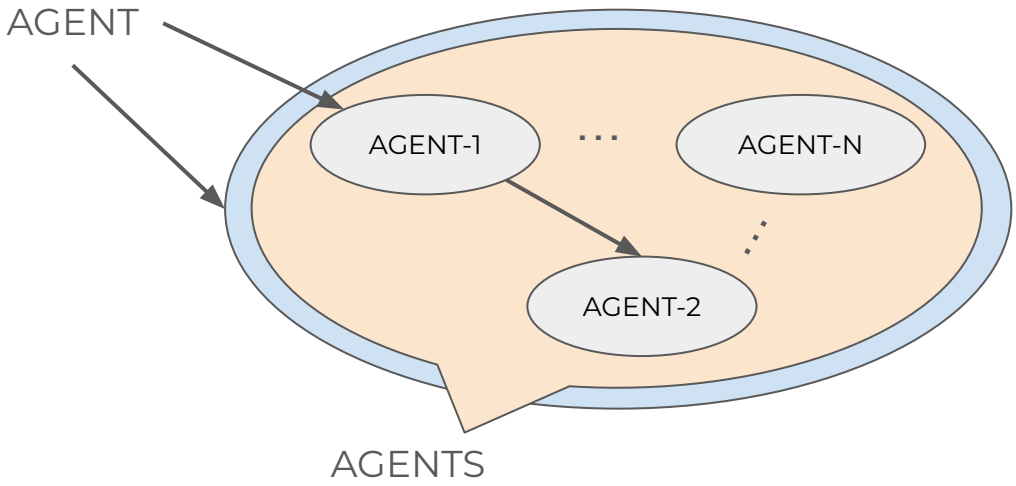
- Uses LangGraph to make a network of agents for internal use.
- Automates code migration & unit-test generation

- LangGraph-based AI assistant handles custom support tasks.
- Scales for as large as 85M active users & reduces resolution time by 80%.

Clarifying terminology: Agent v/s Agents

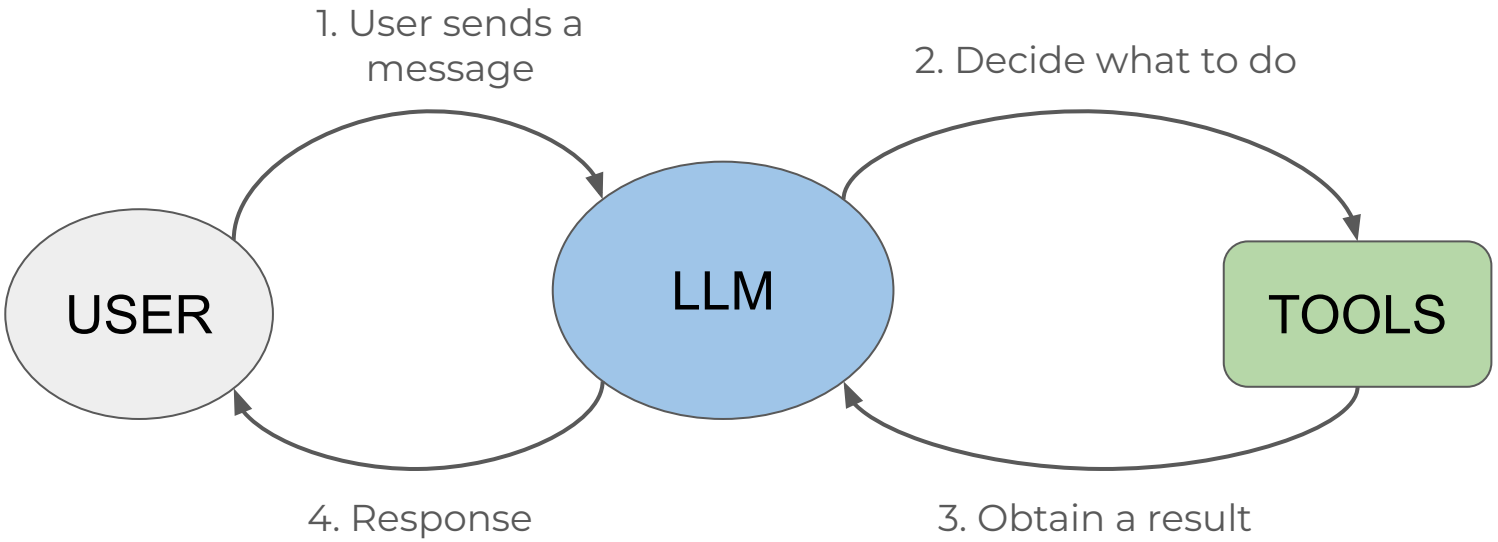
- Our system, that autonomously performs tasks for us ***is an agent***.
- A component of the system that also autonomously performs tasks ***is also an agent***
  - *just at a different hierarchical level*

- ***Agents*** (plural): 2 or more instances of ***agent*** *at the same hierarchical level.*



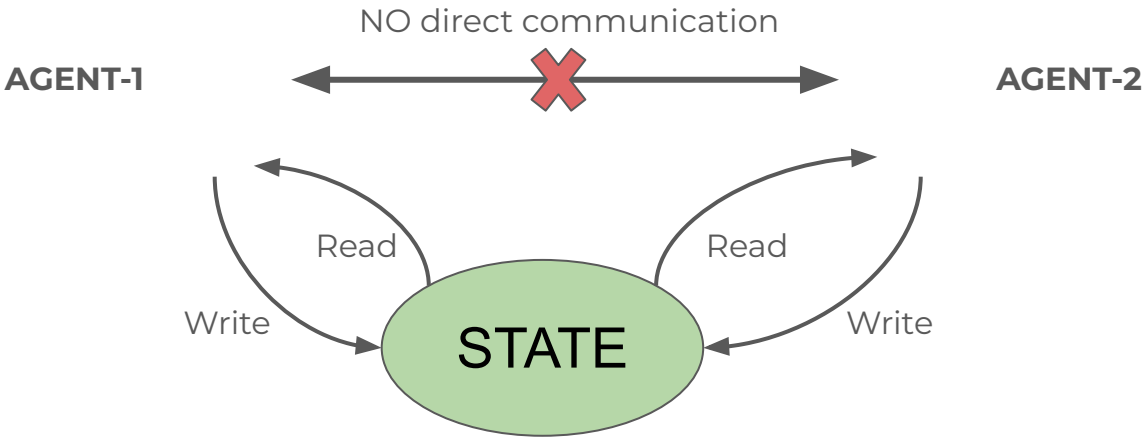
Yes

High-level abstraction: basic LangGraph system



Basic components: State

- **State**:
  - `Typed-Dict` all nodes operate on.
  - Node-to-node communication happens by reading and writing to the state.



Basic components: Nodes & Edges

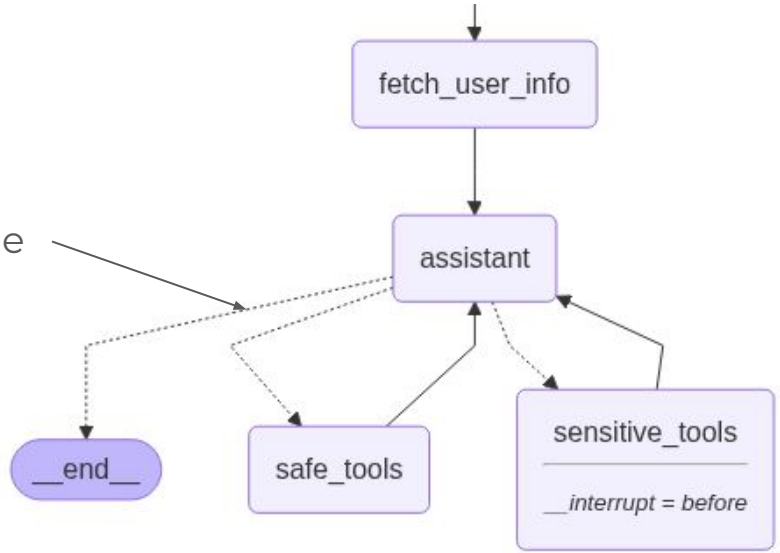
- **Nodes**:
  - Are basically functions (**input:** current state+config, **output:** update to state)
  - Usually will make use of calling an LLM (with some prompt).

- **Edges**:
  - Specify how the agent should transition between nodes.
  - used as `graph.add_edge(start_node, end_node)`

Conditional Edge

- Used when we want to transit between nodes based on certain conditions.
- Implemented via: `graph.add_conditional_edge("node", routing_function)`
- 2nd argument (`routing_function`) specifies the conditional routing from `"node"`.

Dotted lines = conditional edge



We need to compile our graph!

- **`StateGraph`**
  - ONLY the graph that we model our system with.

- **`CompiledStateGraph`**
  - We can't "run" anything with an un-compiled **`StateGraph`**
  - Compiling the graph lets us send inputs and get outputs, using **`invoke()`** and **`stream()`**

`invoke()`
Run the graph (get an output) given an input and a config.

`stream()`
Display each step of the graph's execution given a single input.

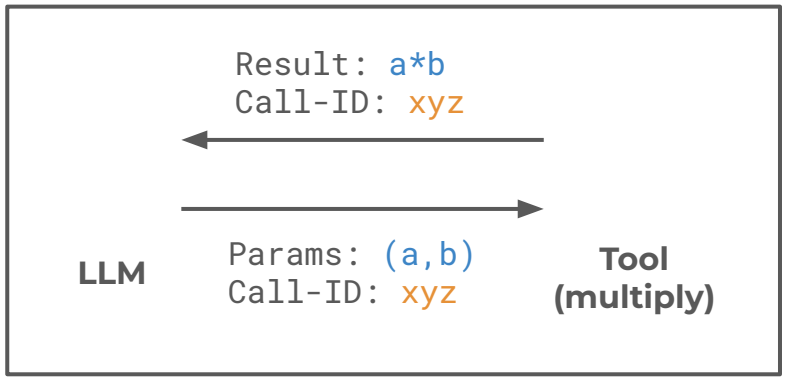
Tools

- Wraps a function & its input schema to pass it to an LLM tuned to use tools.
- LLM requests the execution of the function by passing the input parameters.
- Each tool call has an ID given with the input by the LLM.
- Tool calls terminate after the state receives a matching `ToolMessage`.

```python
from langchain_core.tools import tool

@tool("multiply_tool", parse_docstring=True)
def multiply(a: int, b: int) -> int:
    """Multiply two numbers.

    Args:
        a: First operand
        b: Second operand
    """
    return a * b
```



ToolNode

- Like a prebuilt “toolbox” in LangGraph

```
tools_by_name = {tool.name: tool for tool in tools}
def tool_node(state: dict):
    result = []
    for tool_call in state["messages"][-1].tool_calls:
        tool = tools_by_name[tool_call["name"]]
        observation = tool.invoke(tool_call["args"])
        result.append(ToolMessage(content=observation, tool_call_id=tool_call["id"]
    return {"messages": result}
```

Handling the State - *reducer functions*

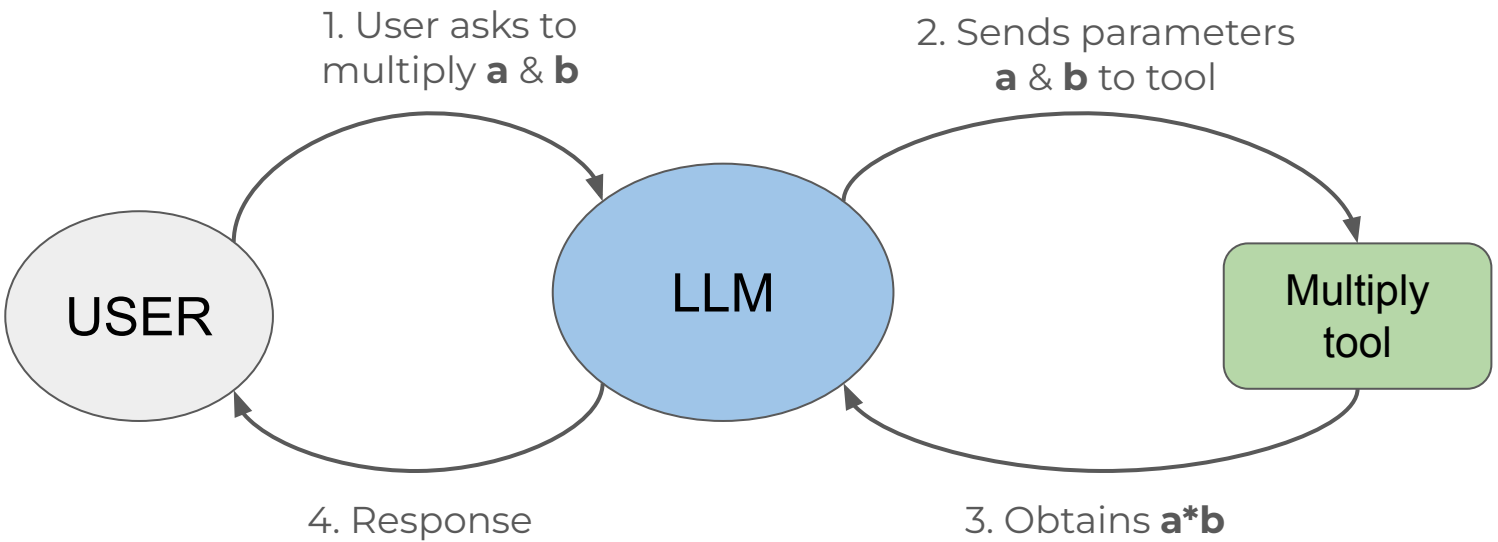
- Specifies how existing state (first positional argument) and update (second positional argument) should reduce to a new state.
- Otherwise the state key will be overwritten!

```python
def add(left, right):
    """Can also import `add` from the `operator` built-in."""
    return left + right


class State(TypedDict):
    messages: Annotated[list[AnyMessage], add]
    extra_field: int
```

Let’s look at a simple example

Example 1: multiply two numbers



Could just as easily be another tool, e.g. web search, code editor, email, etc.

Example 1: we want you to run it with us!

- Scan QR code → README/Quickstart → open `01_basic_langgraph_chatbot.ipynb`
- You should be able to run the code & replicate the example → set up done ✅

Github: https://github.com/TuebingenAICenter/agent-tutorial

Questions so far?

Handling the State - *updating the state*

- Updates through returning from a node

```python
def node(state: State):
    messages = state["messages"]
    new_message = AIMessage("Hello!")

    return {"messages": messages + [new_message], "extra_field": 10}
```

- Updates (+ Routing) from within a node/tool

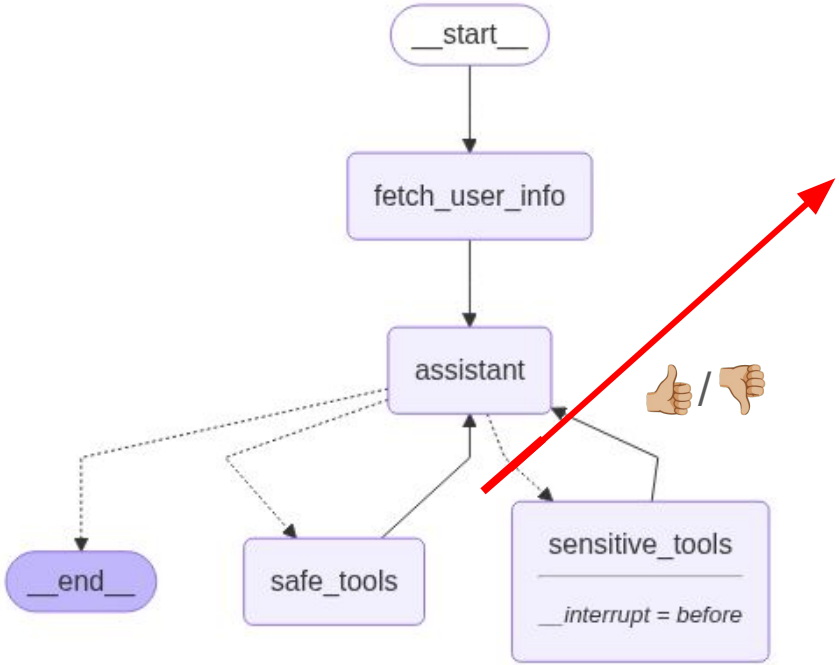
```python
def my_node(state: State) -> Command[Literal["my_other_node"]]:
    if state["foo"] == "bar":
        return Command(update={"foo": "baz"}, goto="my_other_node")
```

- Updates on a compiled graph from outside

```python
graph.update_state(config, {"name": "LangGraph (library)"})
```

```python
graph.invoke({"user_input":"My"})
```

Interruptions/Human in the Loop



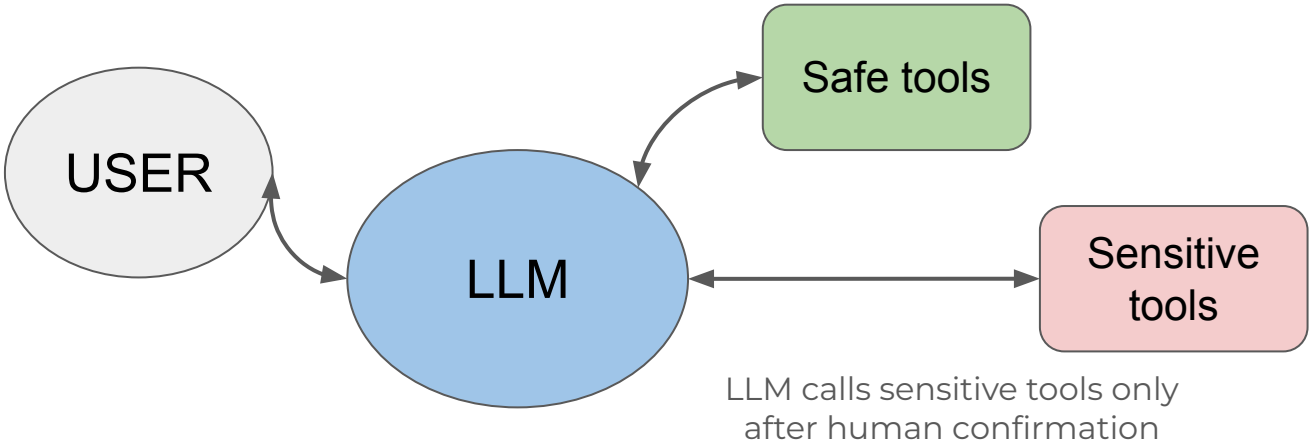
If execution is interrupted: state stores where its supposed to go next in its **.next** attribute

```
snapshot = graph.get_state(config)
snapshot.next[0] # “name_of_node”
```

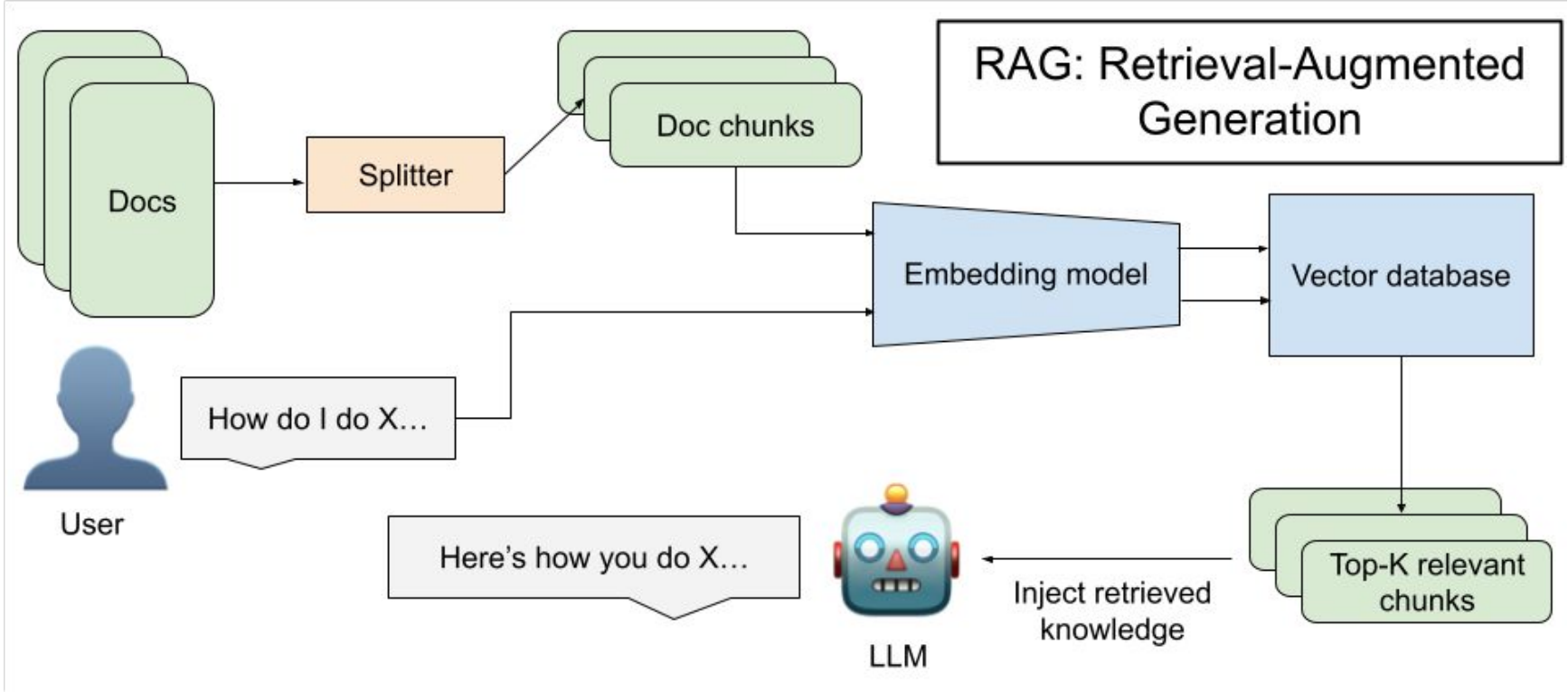
Let’s look at another example

Example 2: Chat with PDF / YouTube video

- A system that uses ***RAG*** to inject relevant knowledge from PDFs and YouTube transcripts into the LLM context. Allows users to chat about specific topics reliably
- **Key lesson:** interrupting execution to put humans-in-the loop



Example 2: what is RAG?



Questions so far?

Model Context Protocol

(MCP)

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Tübingen AI Center

The USB-C of AI Applications

- No new revolutionary tool/idea/innovation
- Standardization of ideas and patterns that came before:
  Tool Calling, RAG, Prompt Engineering, ...

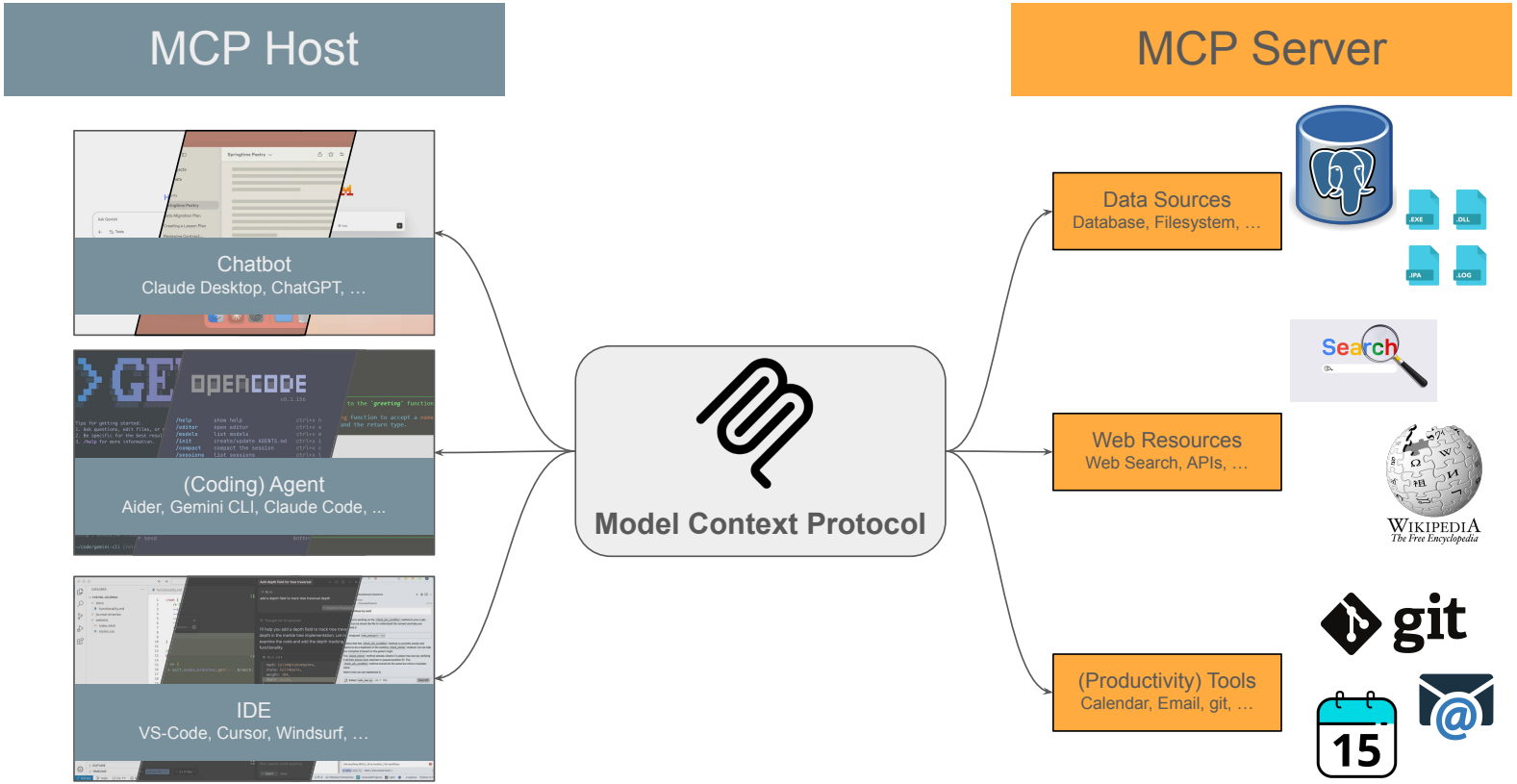
> *"Think of MCP like a USB-C port for AI applications"*
>
> modelcontextprotocol.io

The real value lies in standardization, not innovation

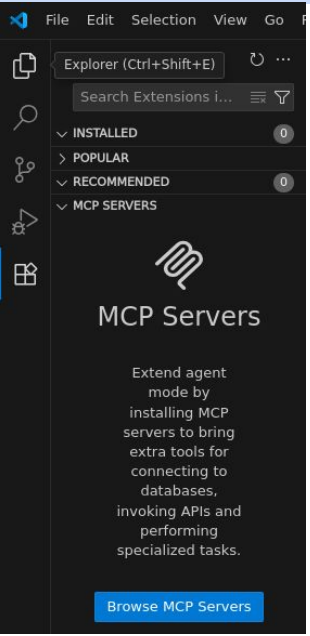
MCP as Communication Protocol

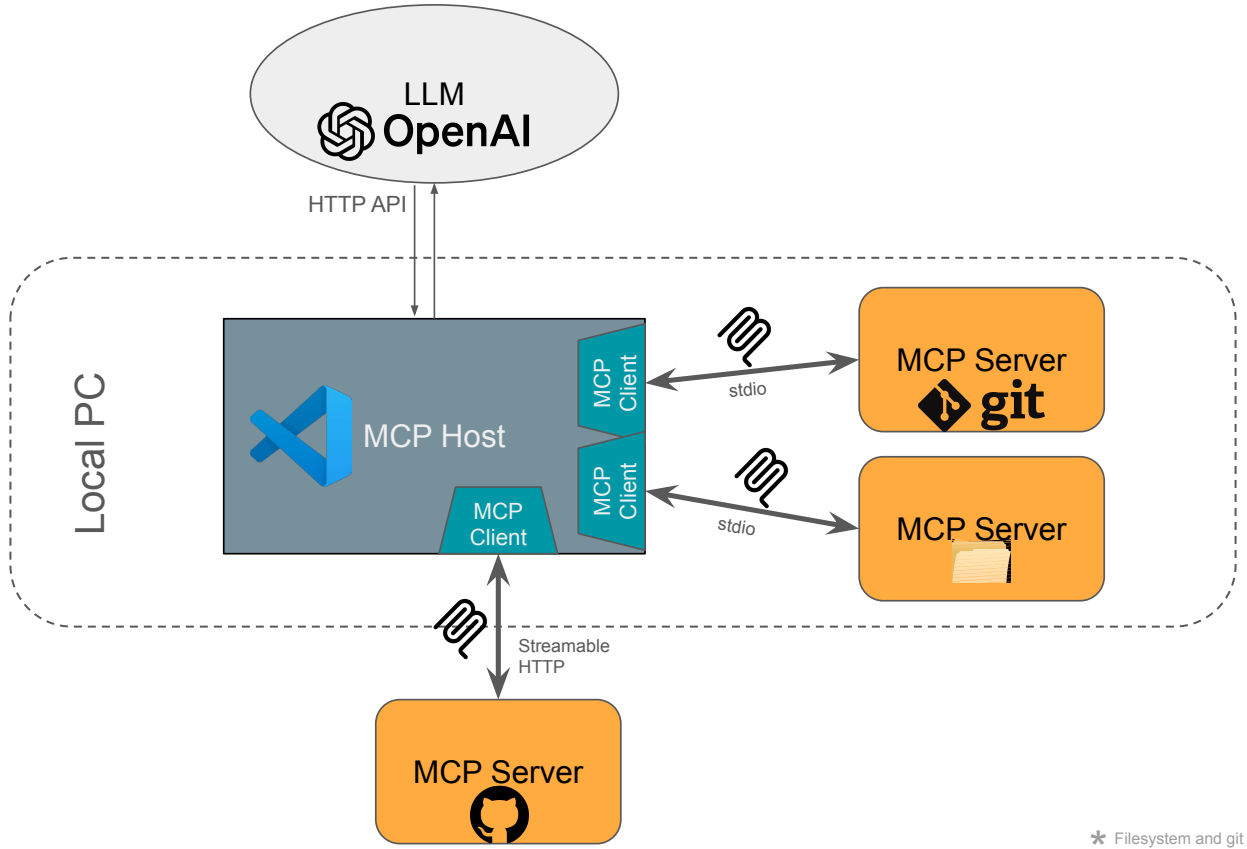
Think: API Specification tailored to AI-Applications

MCP as Communication Protocol



MCP Architecture





* Filesystem and git tools for LLMs are actually built-in in VS-Code. We ignore this fact on this slide for the sake of this example.

MCP Features

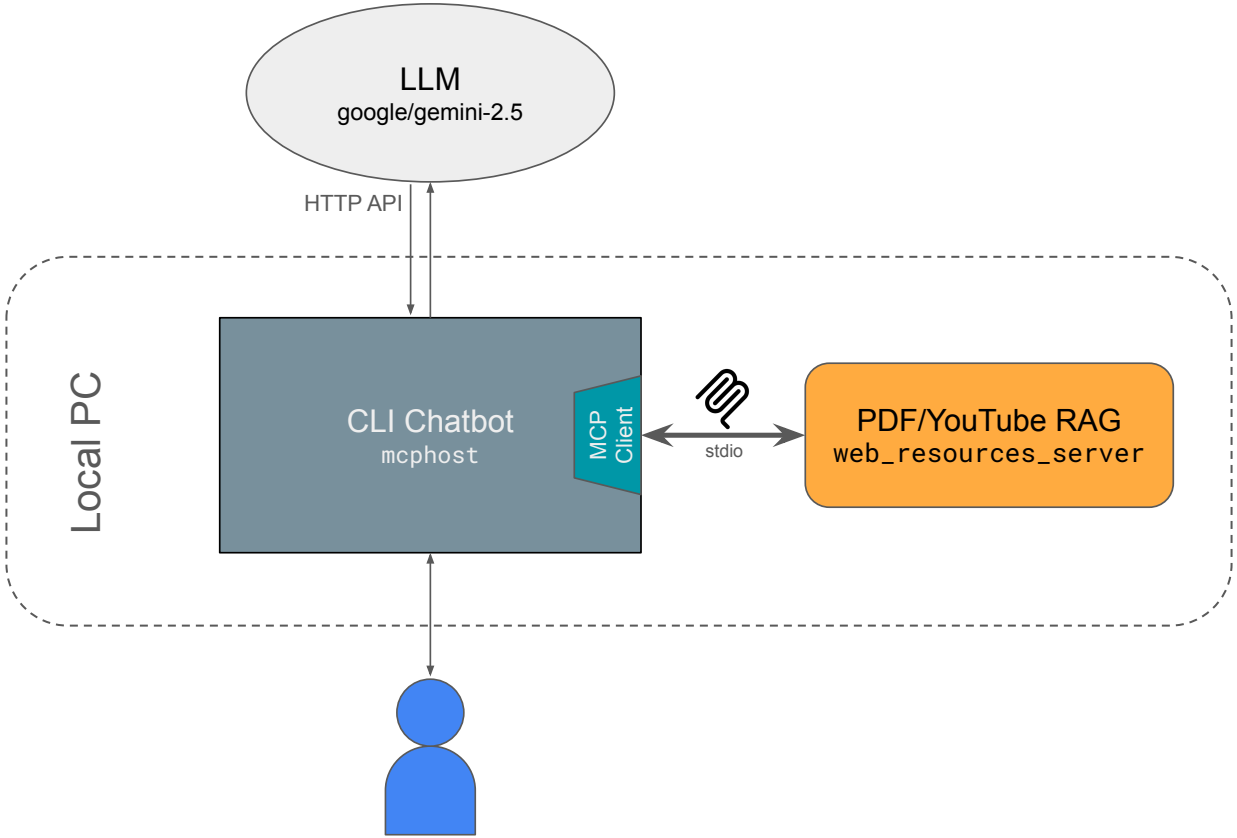
Server Features

Client Features

MCP Server: Tools

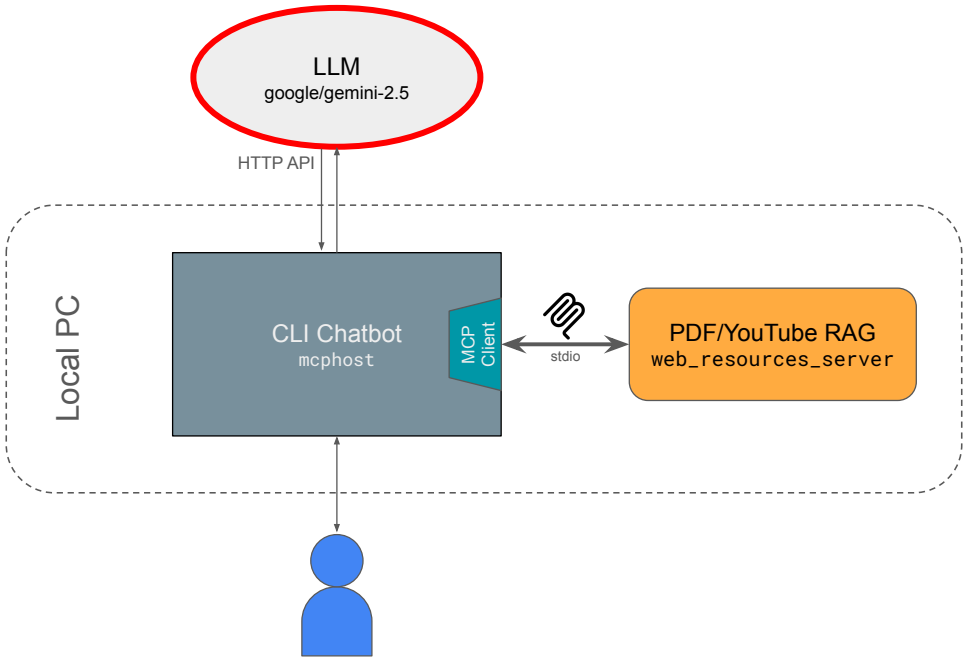
- Tools enable LLMs to interact with external systems
- *Model-controlled*:
  LLM invokes tool calls and reacts to responses

Minimal Example



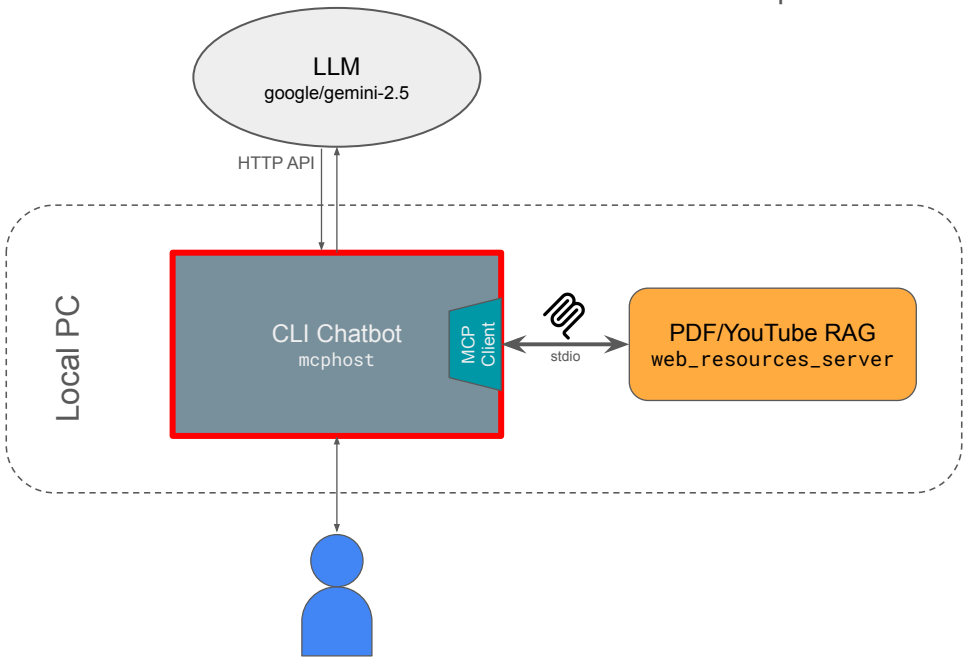
MCP Server: Tools

- Tools enable LLMs to interact with external systems
- *Model-controlled*:
  LLM invokes tool calls and reacts to responses



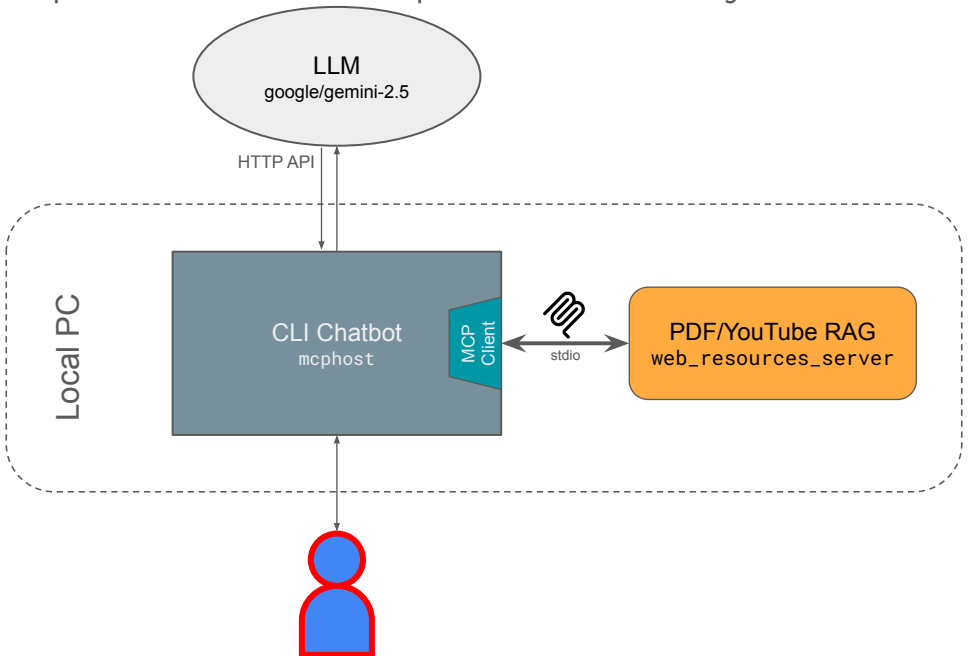
MCP Server: Resources

- Resources allow servers to share data with the MCP Host
- *Application-driven*:
  MCP Host incorporates resources in workflows based on specific use-case



MCP Server: Prompts

- Prompts give users shortcuts through detailed templates tailored to the servers domain
- *User-controlled*:
  Users invoke prompts to steer LLM responses effectively

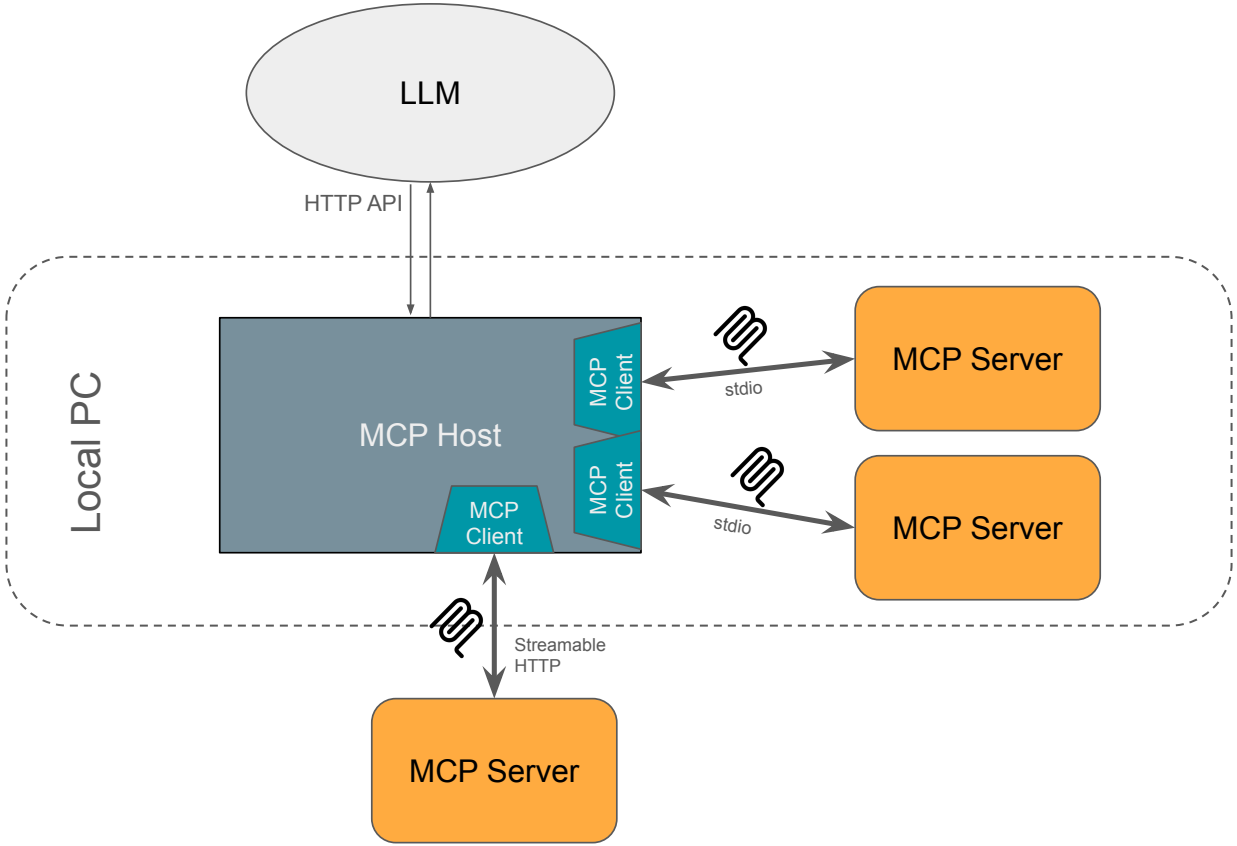


MCP Server Features

Feature	Controlled by…	Description
Tools	Model/LLM LLM decides what tools to call	Functions called by LLM • API calls • File writes • Calculations • …
Resources	Application Host/Agent/User decides what resources to view/add to context/…	Context injected by the Host • File content • Git history • Config files • …
Prompts	User User decides what prompts to expand	Templated Prompts Preconfigured fine-tune prompts tailored to domain of MCP-server

Questions so far?

Questions so far?



Let’s build an MCP Server!

Questions so far?

Scaling up Orchestration Complexity

What & why – workflows

- **Workflows:**
  - LLMs and tools are orchestrated via pre-defined code paths.
  - Predefined rules control how a task is completed.

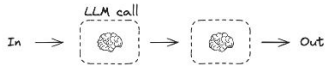
- **Agents** are DIFFERENT from **Workflows**!
  - Agents: systems where LLMs autonomously decide their processes and tool usage & how a task is completed

- Many ways to design workflows ⟹ many ways to design complex agentic systems.

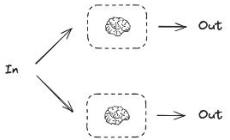
Workflow architectures

Workflows

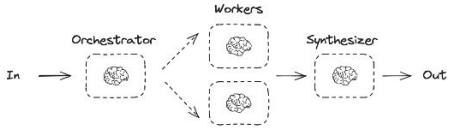
Prompt Chaining



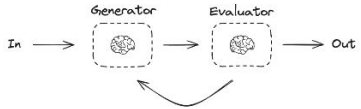
Parallelization



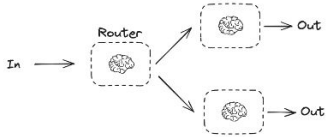
Orchestrator-Worker



Evaluator-optimizer



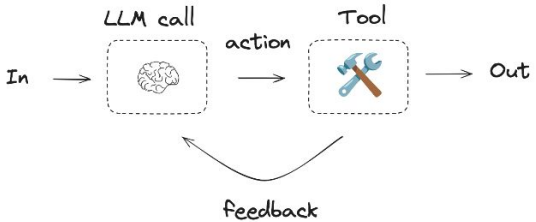
Routing



LLM is embedded in predefined code paths

LLM directs control flow through predefined code paths

Agent



LLM directs its own actions based on environmental feedback

Source: LangGraph documentation

Send API

- Sometimes it's useful for different versions of the `State` to exist in parallel to speed up processing.
- It's also useful to be able to create them dynamically, depending on the current state.
- For that we need to be able to *send* custom inputs to any number of nodes (or instances of those nodes) in our graph.

```
def continue_to_jokes(state: OverallState):
    return [Send("generate_joke", {"subject": s}) for s in state['subjects']]

graph.add_conditional_edges("node_a", continue_to_jokes)
```

Configurables

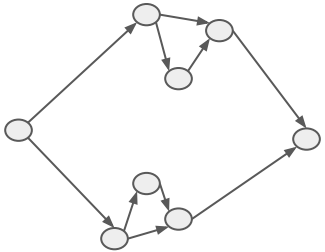
- A `configurable` is used for passing in values to our invocation of the graph that are not altered during runtime.
- It's a dictionary found in `config["configurable"]`
- Examples::
  - Memory thread_id
  - Which LLM/provider to use
  - Limits to parallel executions and looping
  - Verbosity

State
Typed Dictionary used for values that are altered during graph's execution.

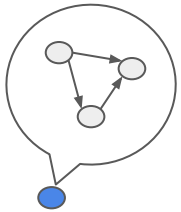
config["configurable"]
Dictionary used for passing in runtime arguments for the graph's execution.

SubGraph as Node

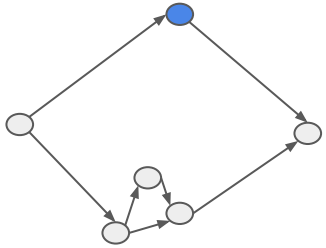
- When complexity & number of nodes increases, thinking in hierarchical levels can help.
- Instead of [A → B → C → ..... P → Q → R] we can wrap the sub-graph C → ... → Q in a node too!
- Graph then becomes much simpler to operate.
- It also allows us to easily parallelize subgraph execution using **Send( )**



Graph with multiple instances of a sub-graph



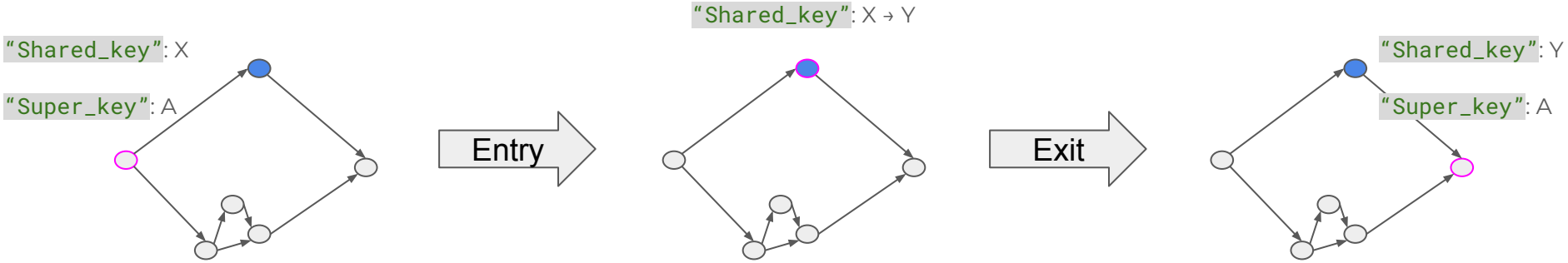
Wrap sub-graph as one node



Reduced complexity

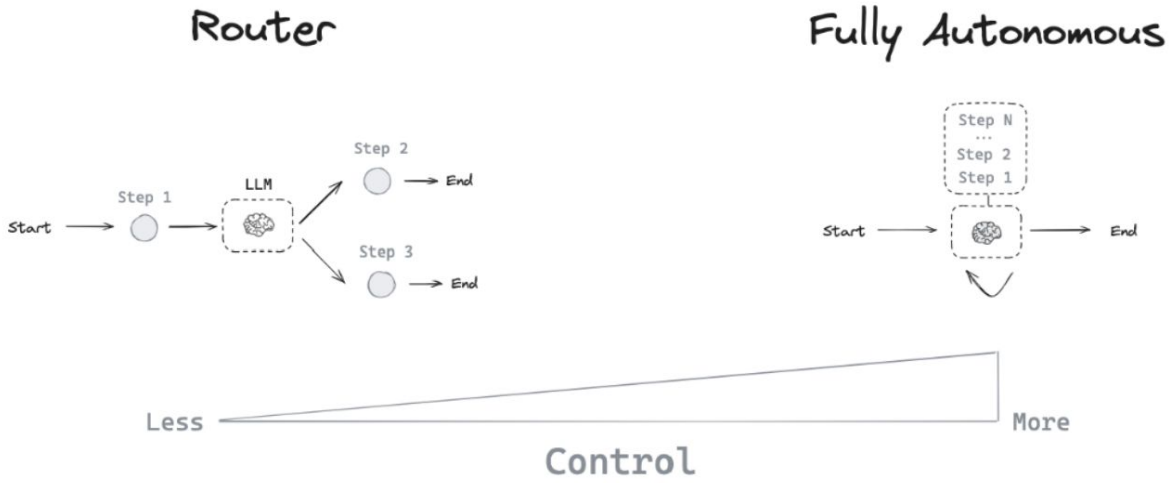
SuperStates & SubStates

- (Super-)graphs and sub-graphs can have separate state schemas.
- Sub-graph's state is not persistent between calls!
- If those share a key called `"shared_key"`:
  - **Entry:** Sub-graph receives the current value from super-graph.
  - **Exit:** Sub-graph's output is merged into super-graph using the super-graph's reducer.



Design patterns to scale agentic systems

- How does orchestration complexity grow with increase in problem complexity?
- Example: agents instructing & iterating with parallel sub-agents

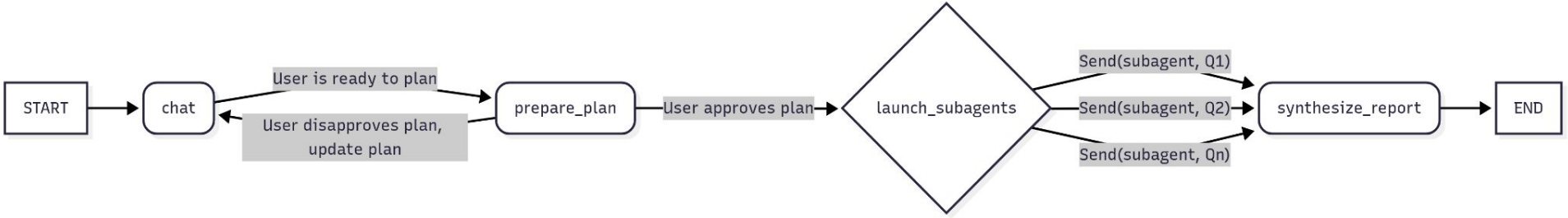


Final example: Deep-Research

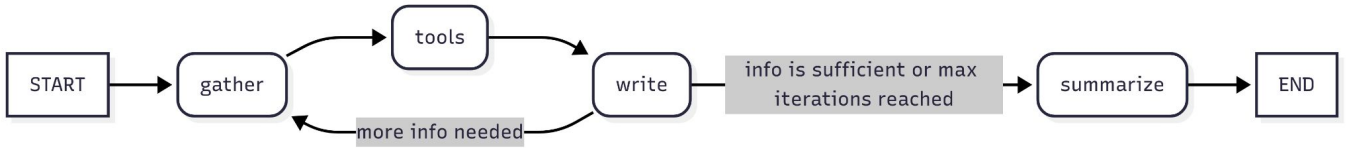
Example: deep research agent

- Given a research question, a Supervisor agent dynamically instantiate and instructs ***K*** parallel Sub-Agents that research sub-topics.
- After each Sub-Agent satisfactorily researches the sub-topic, a final report is written.

Full graph



Sub-(agent) graph



Questions?

Recap: what did we learn?

- **Basic syntax and concepts**
  - States
  - Nodes
  - Edges

- **Designing agentic systems**
  - Tools
  - Interruptions and checkpointing
  - Human-in-the-loop
  - Agent as a supervisor

Dive further!

- More design patterns!
  - Router
  - Structured output
  - Custom-agent Architectures
  - Parallelization
  - Subgraphs
  - Reflection
  - Prompt optimisation

- Deployment

Hands On

Some Pointers

- MCP Registry via Github: A curated list of MCP servers

Some Ideas

- Issue tracker
-