

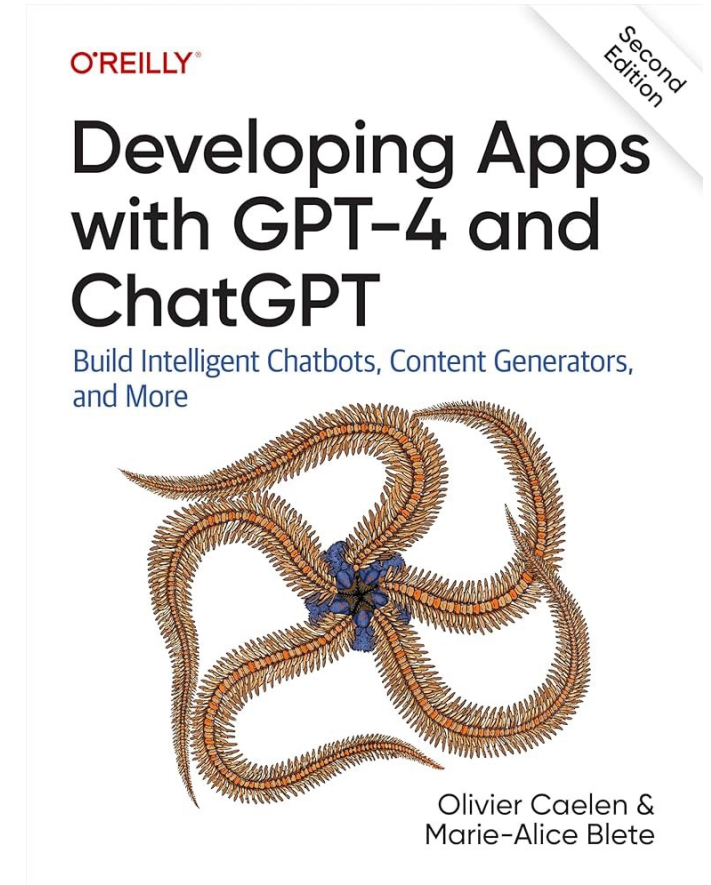
GOTO **COPENHAGEN**

Craft software with confidence in 2026

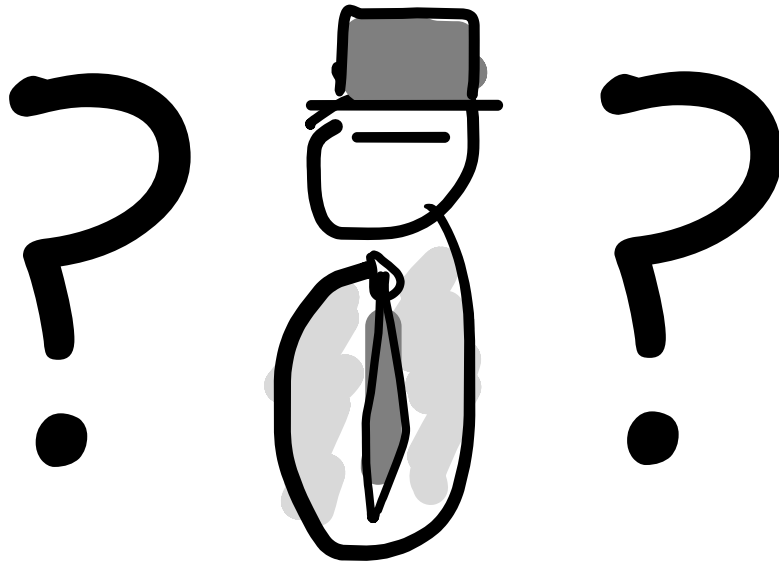
AI Engineering: from first LLM call to multi-agent chaos



Marie-Alice Blete
Staff AI Engineer @ Komodo Health



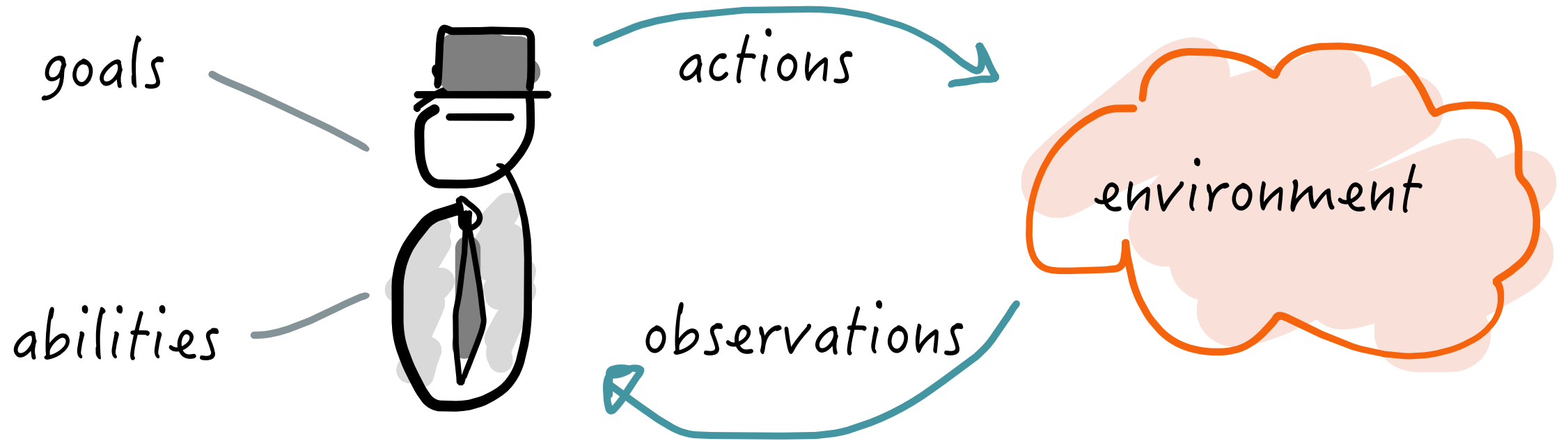
agent ?



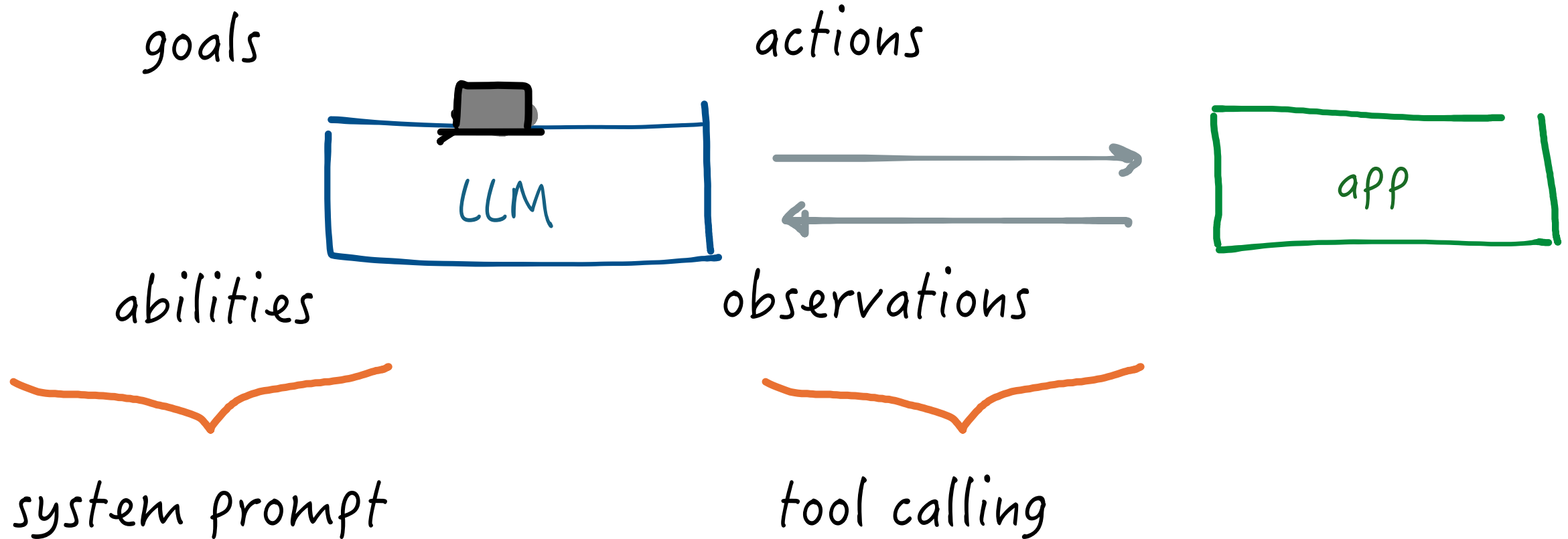
ReAct agents

Tool-calling agents

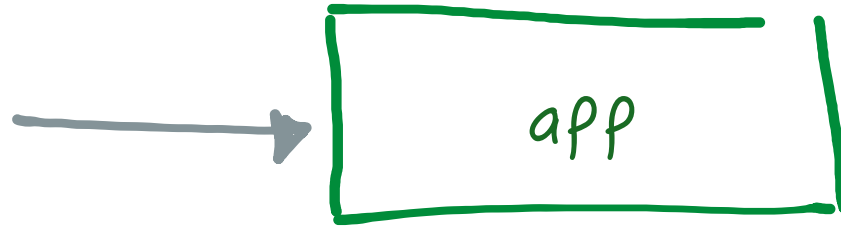
ReAct agent (2023)



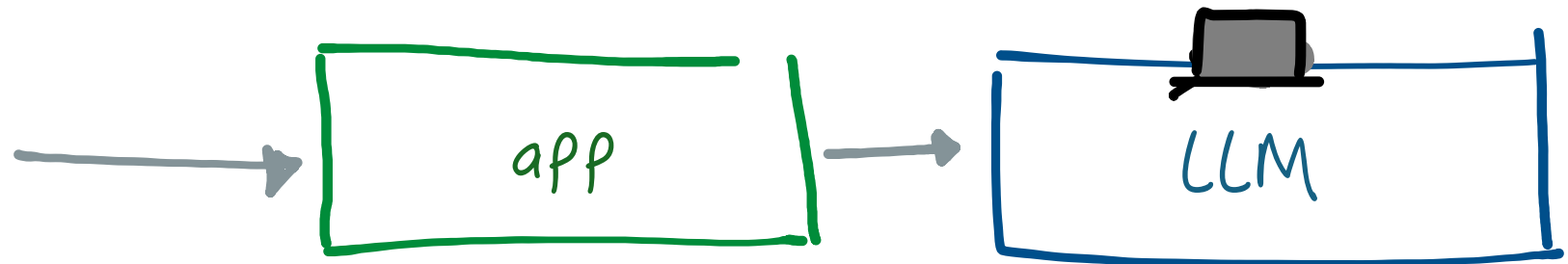
Tool (or Function) calling agent



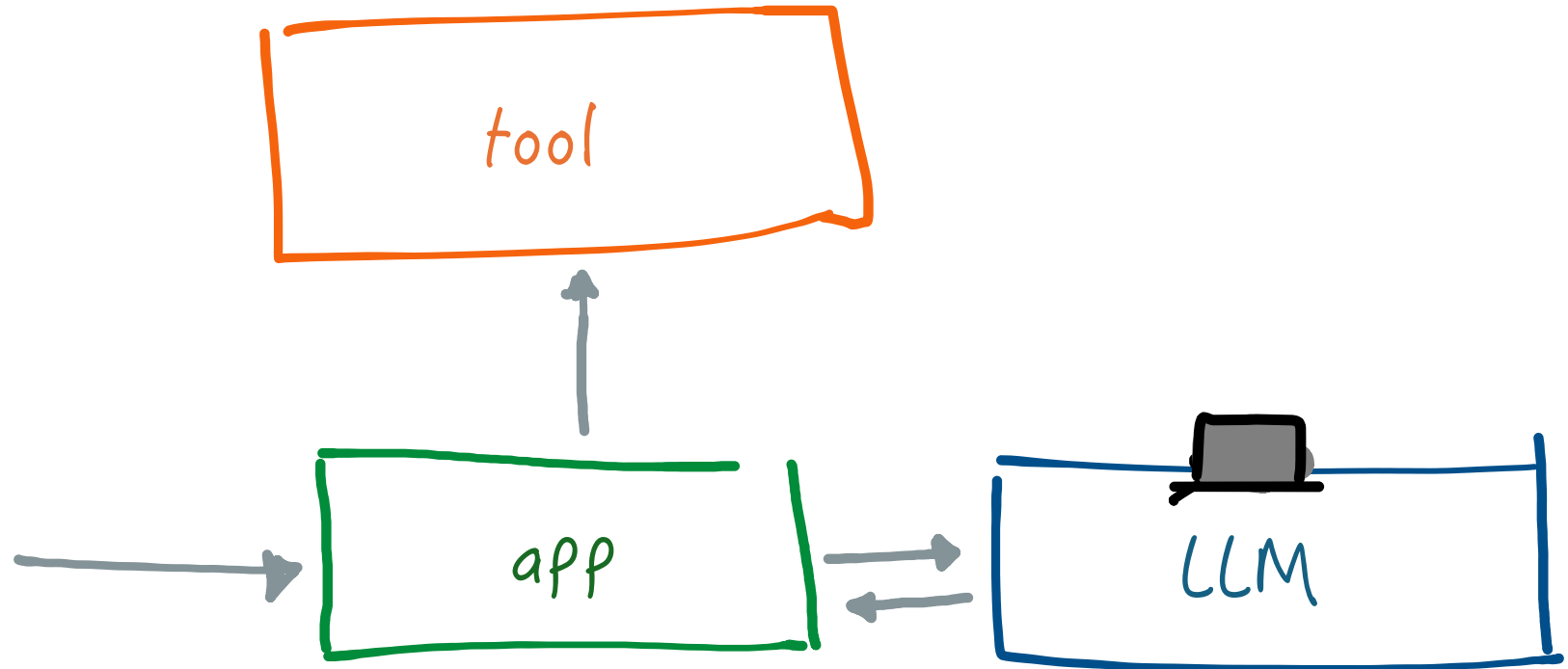
Tool (or Function) calling agent



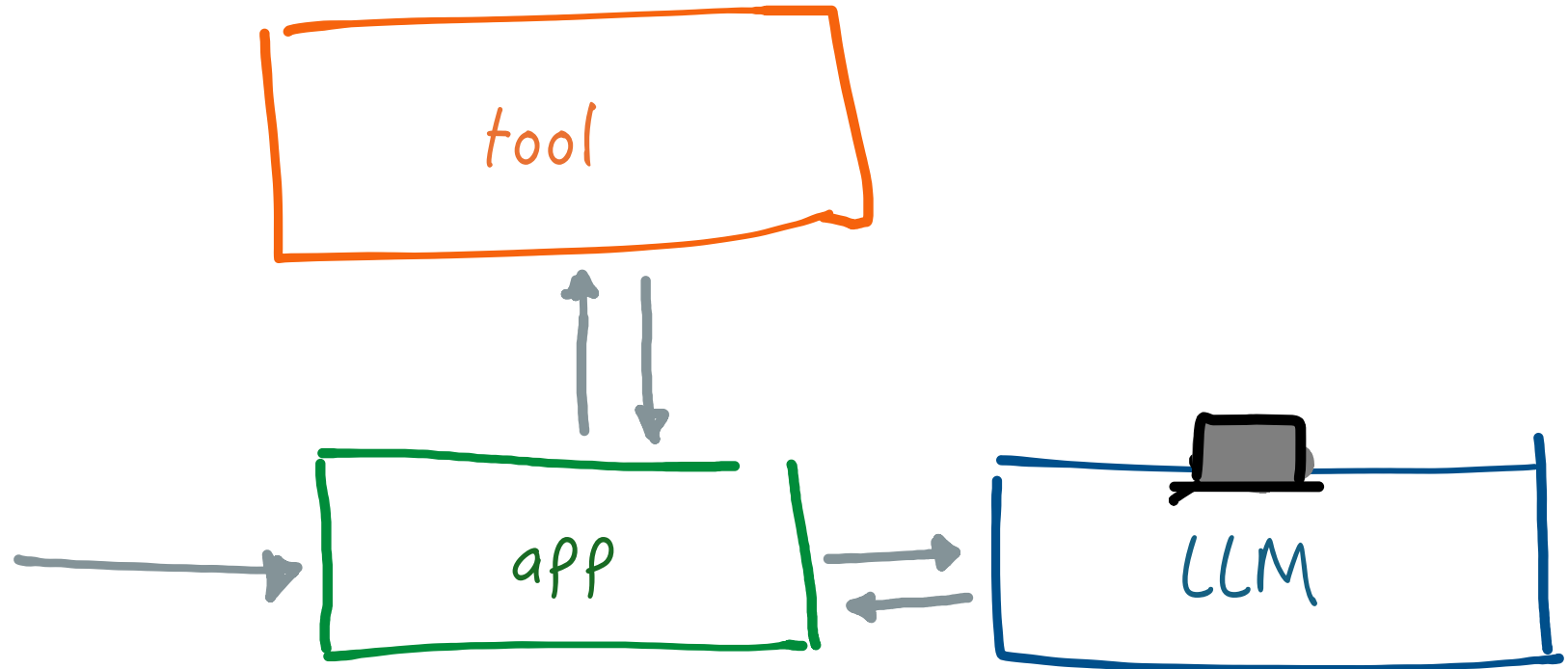
Tool (or Function) calling agent

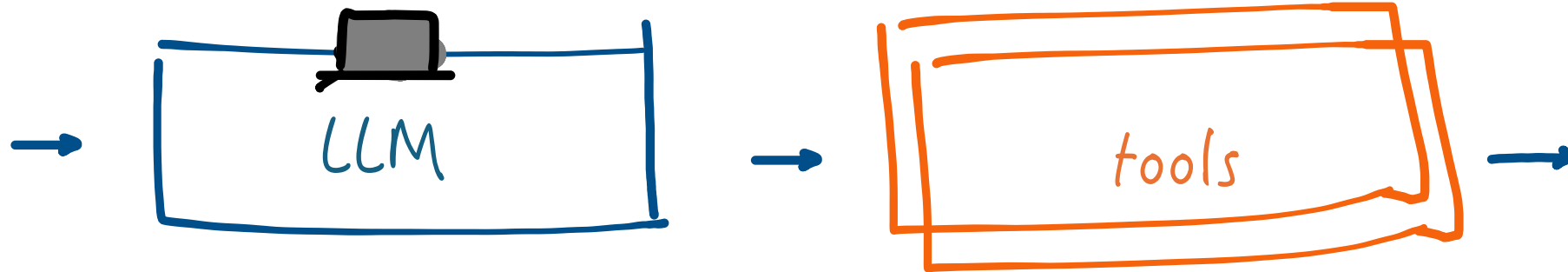


Tool (ou Function) calling agent



Tool (ou Function) calling agent

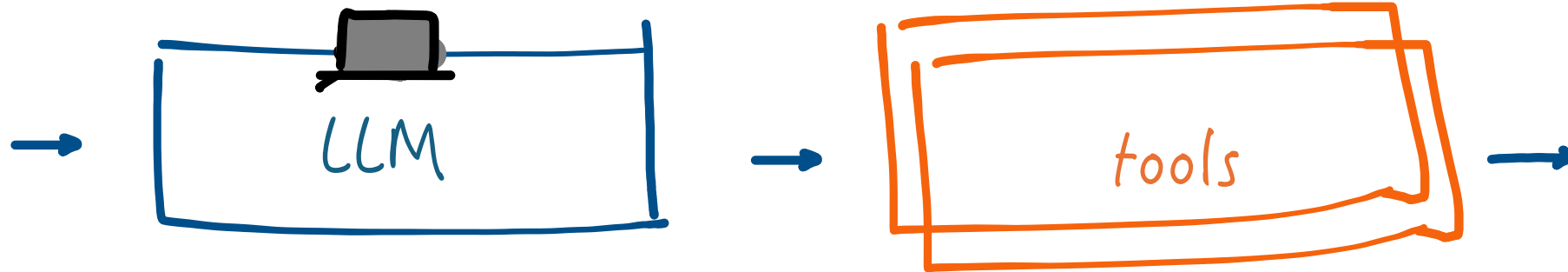




System: Your goal is to answer the user
You can use these tools <...>

User: Create a cohort of patients with Psoriasis in 2023

} API request 1



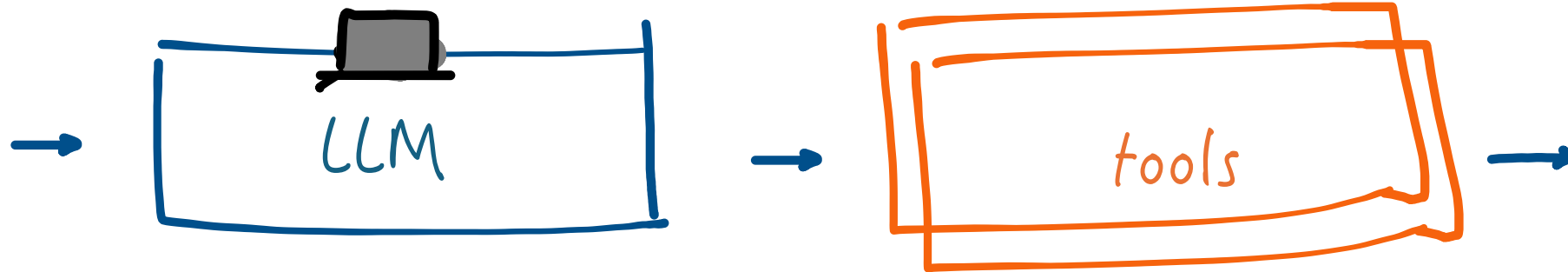
System: Your goal is to answer the user
You can use these tools <...>

User: Create a cohort of patients with Psoriasis in 2023

LLM: `tool_call create_cohort, args:`
`{date_start: 2023-01-01, date_end: 2023-12-31, criteria: psoriasis}`

} API request 1

API response 1



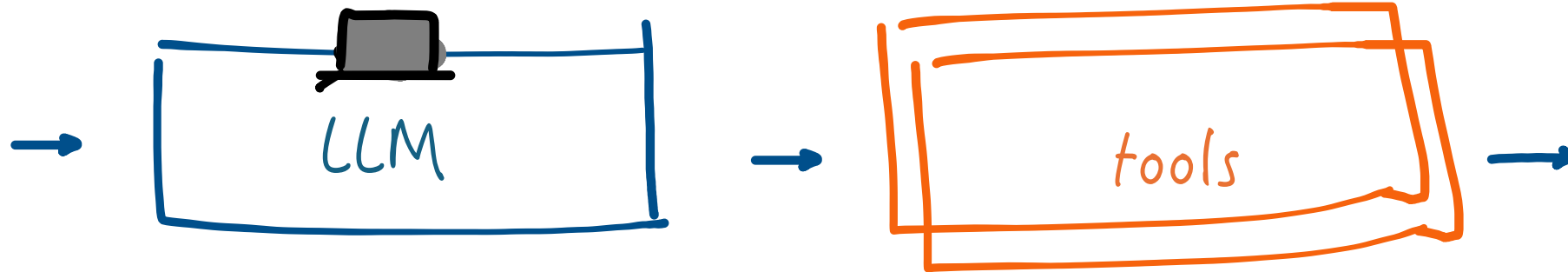
System: Your goal is to answer the user
You can use these tools <...>

User: Create a cohort of patients with Psoriasis in 2023

LLM: `tool_call create_cohort, args:`
`{date_start: 2023-01-01, date_end: 2023-12-31, criteria: psoriasis}`

Tool: cohort created with id: 123456

API request 2



System: Your goal is to answer the user
You can use these tools <...>

User: Create a cohort of patients with Psoriasis in 2023

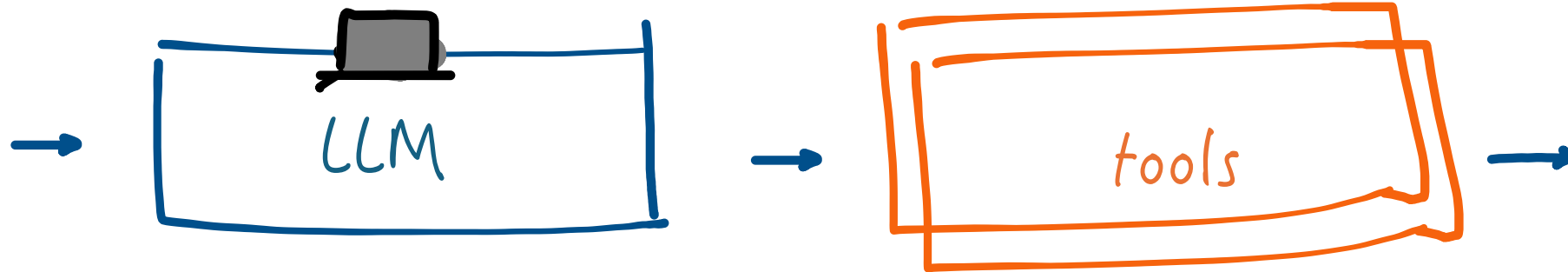
LLM: `tool_call create_cohort, args:`
`{date_start: 2023-01-01, date_end: 2023-12-31, criteria: psoriasis}`

Tool: cohort created with id: 123456

LLM: `tool_call render_cohort, args: {id: 123456}`

API request 2

API response 2



System: Your goal is to answer the user
You can use these tools <...>

User: Create a cohort of patients with Psoriasis in 2023

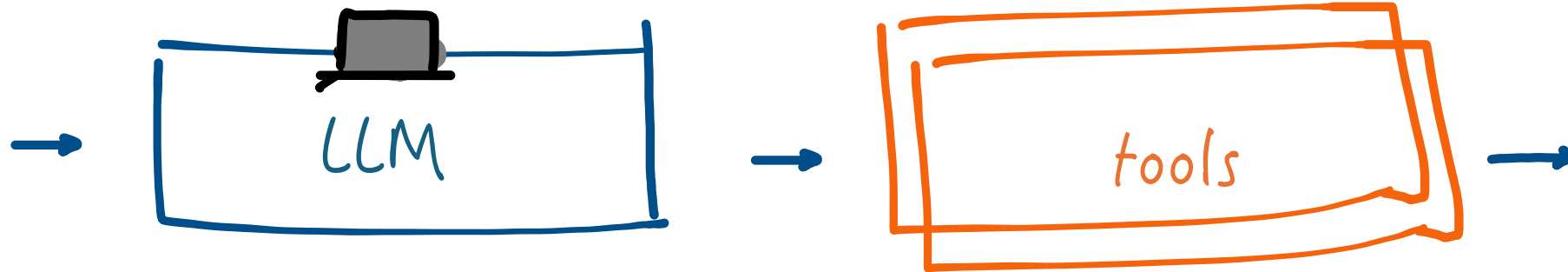
LLM: `tool_call create_cohort, args:`
`{date_start: 2023-01-01, date_end: 2023-12-31, criteria: psoriasis}`

Tool: cohort created with id: 123456

LLM: `tool_call render_cohort, args: {id: 123456}`

Tool: cohort visualization id 123456 generated successfully

} API request 3



System: Your goal is to answer the user
You can use these tools <...>

User: Create a cohort of patients with Psoriasis in 2023

LLM: `tool_call create_cohort, args:`
`{date_start: 2023-01-01, date_end: 2023-12-31, criteria: psoriasis}`

Tool: cohort created with id: 123456

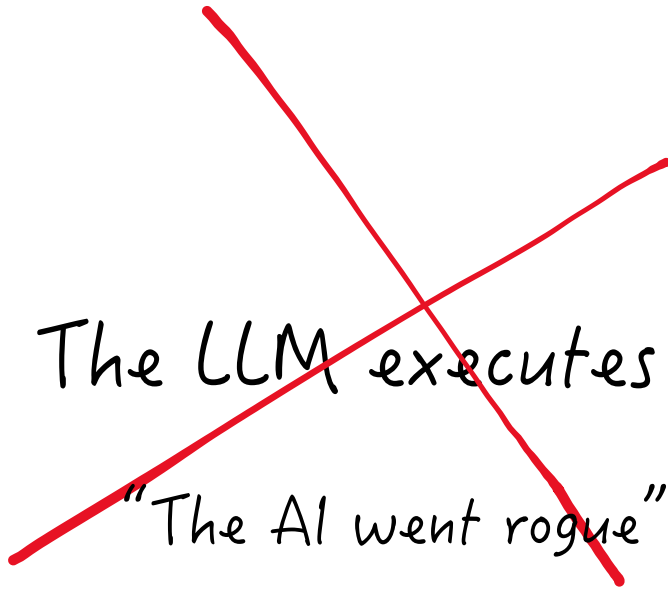
LLM: `tool_call render_cohort, args: {id: 123456}`

Tool: cohort visualization id 123456 generated successfully

LLM : Here is the requested patient cohort.

API request 3

API response 3



The LLM executes tools

"The AI went rogue"

Key takeaways

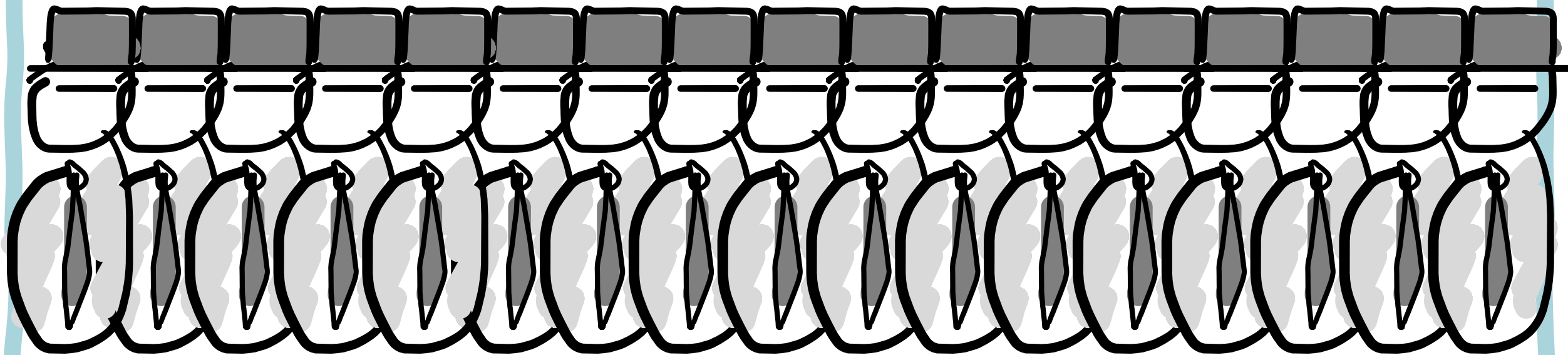
Agent = LLM + application code combined

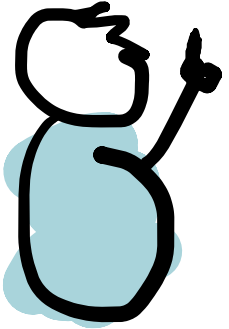
LLM loop = one user input,
multiple autonomous tool calls



The LLM is autonomous in requesting code execution,
receives the result and iterates to reach a goal

more agents, more...

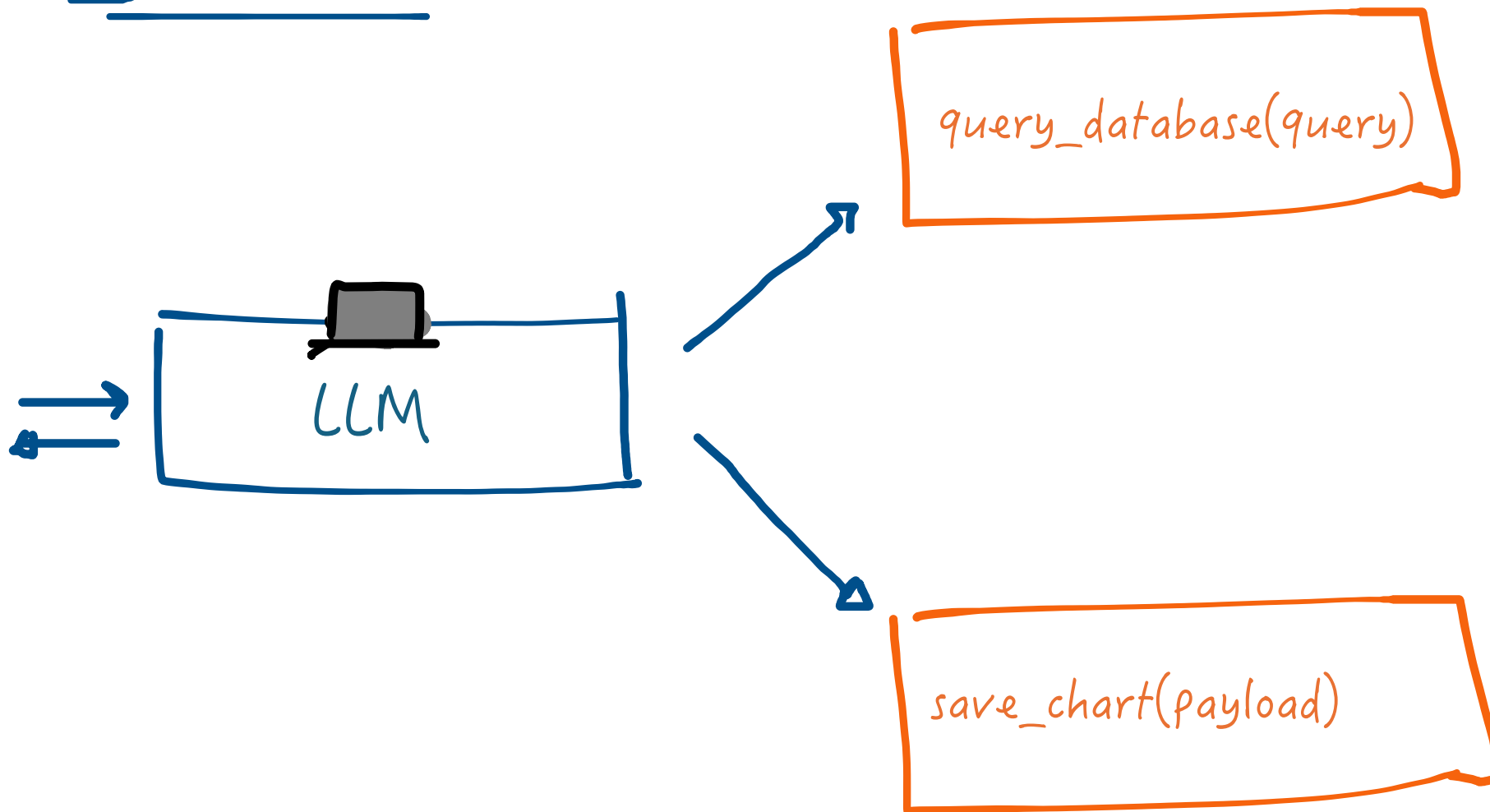




Feature request

Build a report (charts, text, conclusions) based on data,
from a user question.

1 Single Agent



```
from strands.agent import Agent
from strands.tools import tool
```

STRANDS

<https://strandsagents.com/>

```
@tool
```

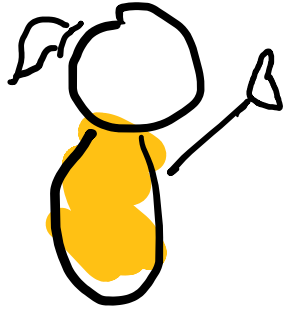
```
def query_database(sql: str) -> list[dict]:
```

```
# Implementation...
```

```
pass
```

```
agent = Agent(
    name="Healthcare Analyst",
    system_prompt="You are a data analyst with database and viz tools",
    model=model,
    tools=[query_database, save_chart]
)
```

```
result = await agent.invoke_async("Show top 10 HCPs by prescription volume")
```



Evaluate an agent?

Similar to "standard software"
but with semantic checks and non determinism



agent output:
are the results correct ?

NEW !



agent behavior:
did we get the results efficiently?



Evaluate an agent?

Similar to "standard software"
but with semantic checks and non determinism



agent output:
are the results correct ?

- LLM as judge
- datasets of questions/answers
- human-in-the-loop

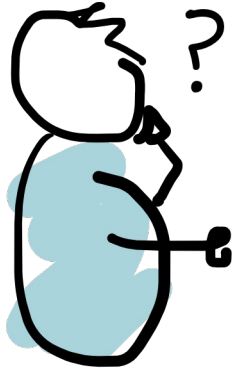
NEW !



agent behavior:
did we get the results efficiently?

- tokens spent (\$)
- agent loops and tool calls
- latency/total time

(demo)



How does the LLM know how to query the database?



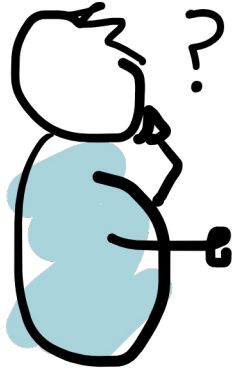
Ideas

~~- Let the agent explore by itself~~

unreliable, risk of hallucinations, waste of tokens

~~- Add a "get_schema" tool~~ → waste of LLM loop

- Dump the schema in the system prompt, with examples of common queries, joining patterns + leverage cache



What about security ?



Prompt injection

Behavioral manipulation

System exposure

Unauthorized data access and modification

Takeaways

Evaluations should be on
agent output AND agent behavior

- ✓ Instrument - monitor
- ✓ Create baselines and compare



Takeaways

Your code is the security boundary: treat LLM output as untrusted input

- ✓ Authorization and authentication in code
- ✓ Validate tool arguments before execution
- ✓ Trace everything
- ✓ External pen testing



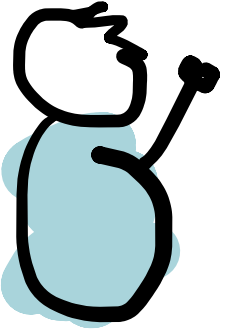
Evaluation of architecture 1:

✓ Feature is implemented.

BUT

Efficiency metrics show:

- Many LLM loops to get the answer
cause: LLM has trouble getting the query right
- Huge token spend
cause: SQL queries and results bloat the context



Constraints

User questions and logic do not match how the database was designed

Database schema is complicated – many tables, many joins

Ideas

- Expose API instead of DB



Ideas

~~- Expose API instead of DB~~

could work + good for security,
but user queries are too unpredictable in our case



Ideas

~~- Expose API instead of DB~~

could work + good for security,
but user queries are too unpredictable in our case

- Use a more powerful model

Ideas

~~- Expose API instead of DB~~

could work + good for security,
but user queries are too unpredictable in our case

~~- Use a more powerful model~~



Ideas

~~- Expose API instead of DB~~

could work + good for security,
but user queries are too unpredictable in our case

~~- Use a more powerful model~~



- Use a multi agent design

Ideas

~~- Expose API instead of DB~~

could work + good for security,
but user queries are too unpredictable in our case

~~- Use a more powerful model~~



- Use a multi agent design

Ideas

~~- Expose API instead of DB~~

could work + good for security,
but user queries are too unpredictable in our case

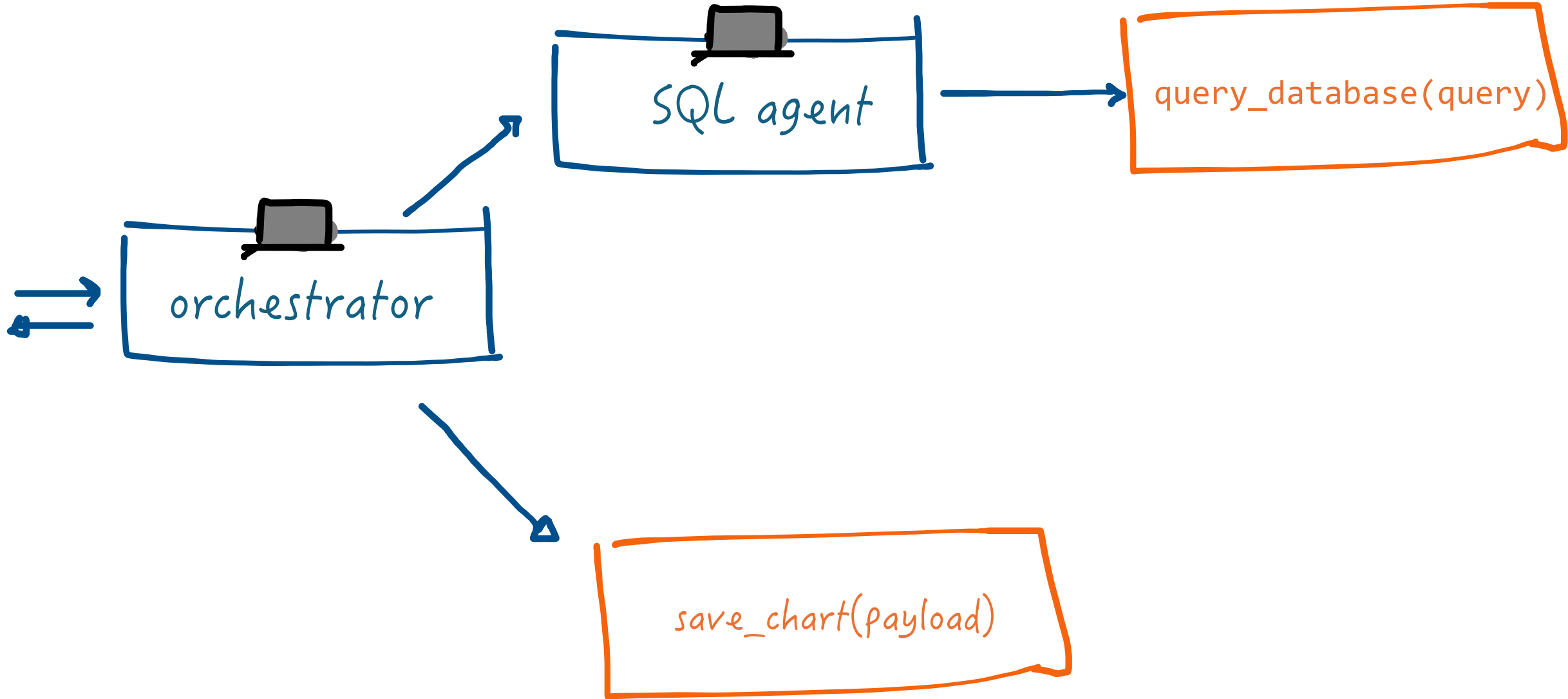
~~- Use a more powerful model~~



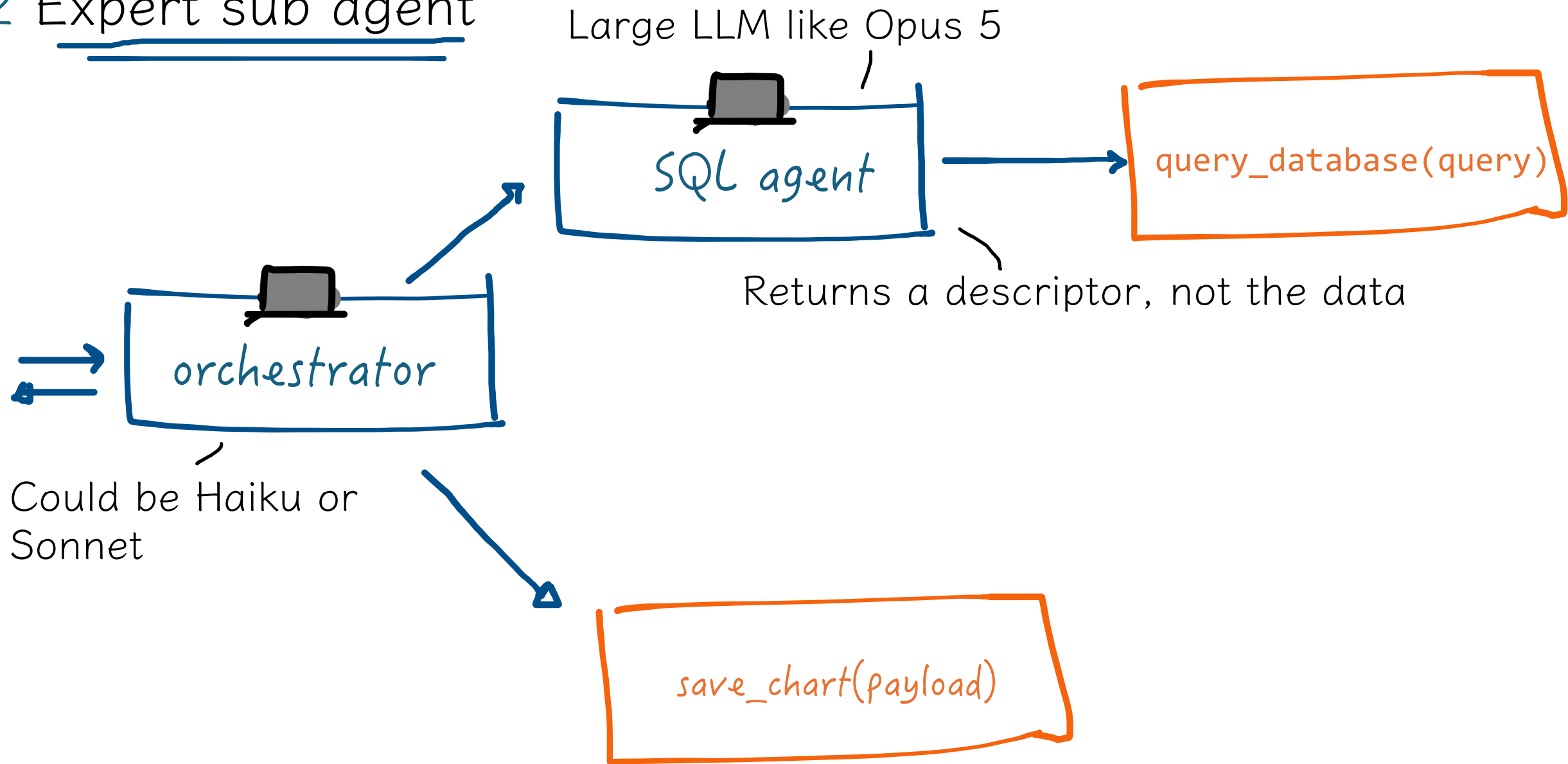
- Use a multi agent design

+ Data written to
memory and not context

2 Expert sub agent



2 Expert sub agent



`query_database` writes the result somewhere and returns a descriptor

```
{
  "result_id": "res_8f3a21",
  "row_count": 48,
  "columns": [
    {"name": "month", "type": "date"},
    {"name": "revenue", "type": "numeric"}
  ],
  "sample": [
    {"month": "2023-01-01", "revenue": 84210},
    {"month": "2023-02-01", "revenue": 91355}
  ]
}
```

descriptor is used by save_chart

```
save_chart(  
    {  
        "result_id": "res_8f3a21",  
        "spec": {  
            "type": "line",  
            "x": "month",  
            "y": "revenue",  
            "title": "Monthly revenue 2023"}  
    })
```

DB schema description

SQL queries and results

Next conversation steps

SQL Expert agent

DB schema description

SQL queries and results

ephemeral

Main agent

SQL Agent handoff

Next conversation steps



Evaluate a multi-agent system?

evaluate individual agents

+

system output:
are the results correct ?

system behavior:
did we get the results efficiently?

Evaluation of architecture 2:

- ✓ Efficiency metrics show clear improvement:

Data never stored in context
SQL iterations do not bloat

- ✓ And we also have hallucination control:

Data comes directly from database
Can be audited – data and its source query can be traced
main agent context



Evaluation of architecture 2:

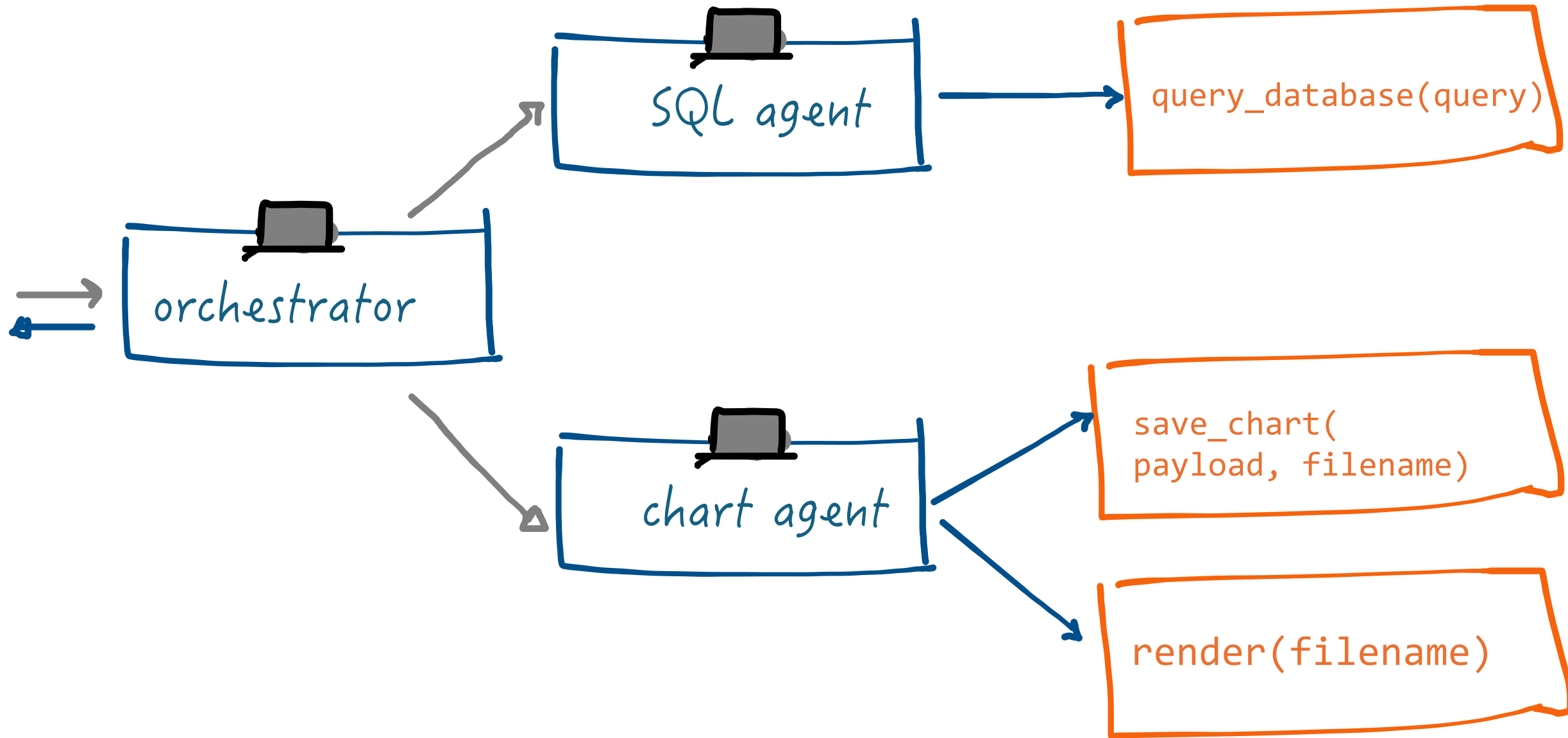
BUT

We notice that charts are not always great ...

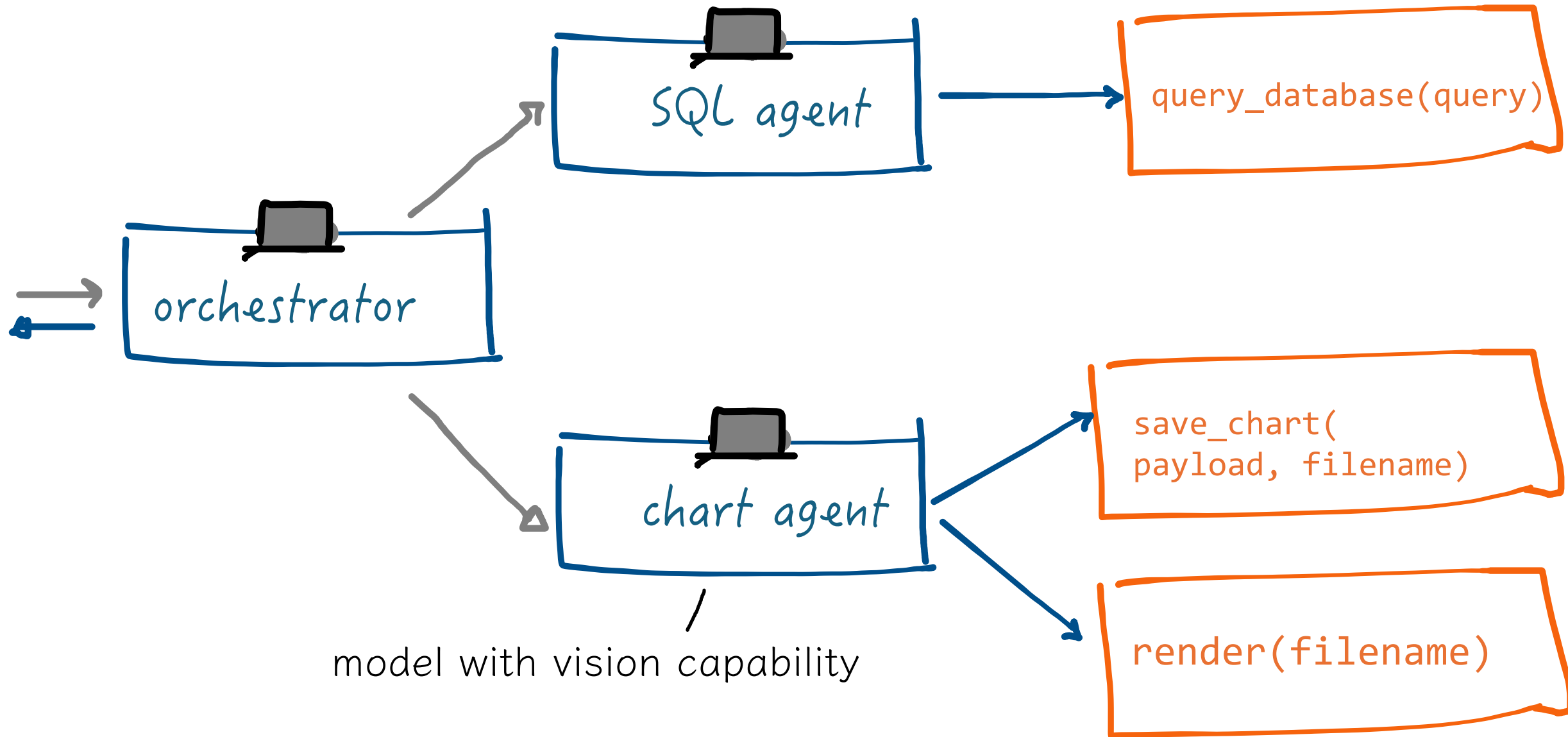
Ideas

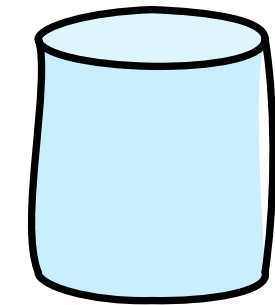
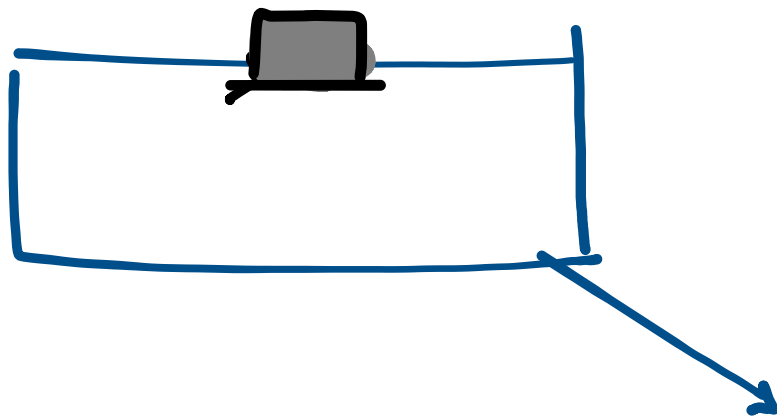
- Add another subagent with vision capabilities

2+ Orchestrator + experts

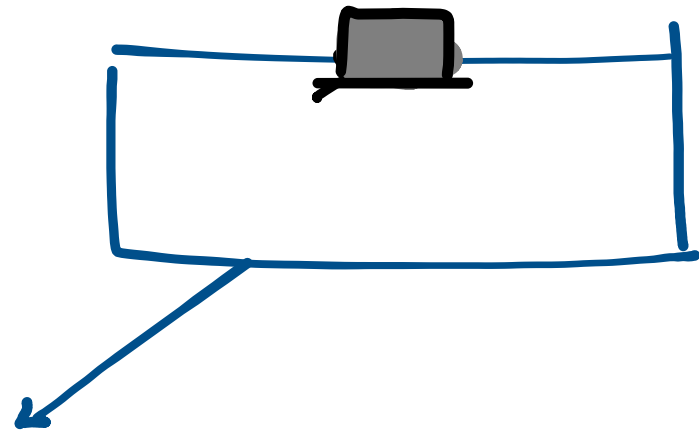


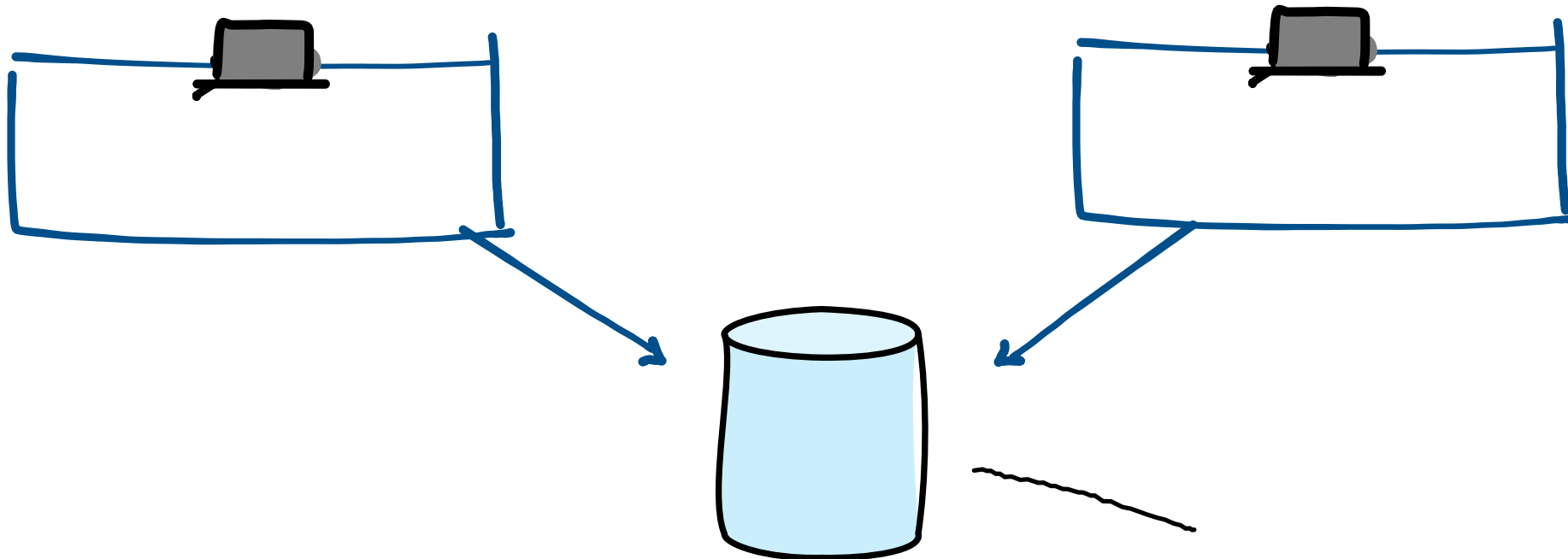
2+ Orchestrator + experts





shared
memory





shared
memory

Options:
In-memory map of objects
Blob storage
Shared filesystem
...

```
@tool
async def delegate_to_sql_expert(question: str):
    sql_agent = create_sql_expert()
    return await sql_agent.invoke_async(question)

@tool
async def delegate_to_plotly_expert(storage_key: str, request: str):
    plotly_agent = create_plotly_expert()
    return await plotly_agent.invoke_async(request)

orchestrator = Agent(
    name="Orchestrator",
    system_prompt="Delegate to SQL Expert or Plotly Expert",
    tools=[delegate_to_sql_expert, delegate_to_plotly_expert]
)
```

Evaluation of architecture 2+:

✓ Efficiency:

Better context management

Cost optimization:

- simple model for simple tasks
- complex model for complex tasks

✓ Results:

Better charts!



✓ Software:

Modularity

Evaluation of architecture 2+:

BUT

Could we improve efficiency even more?

Unreliable: the orchestrator sometimes forgets the chart step

Wasteful: an LLM call for a flow we already know



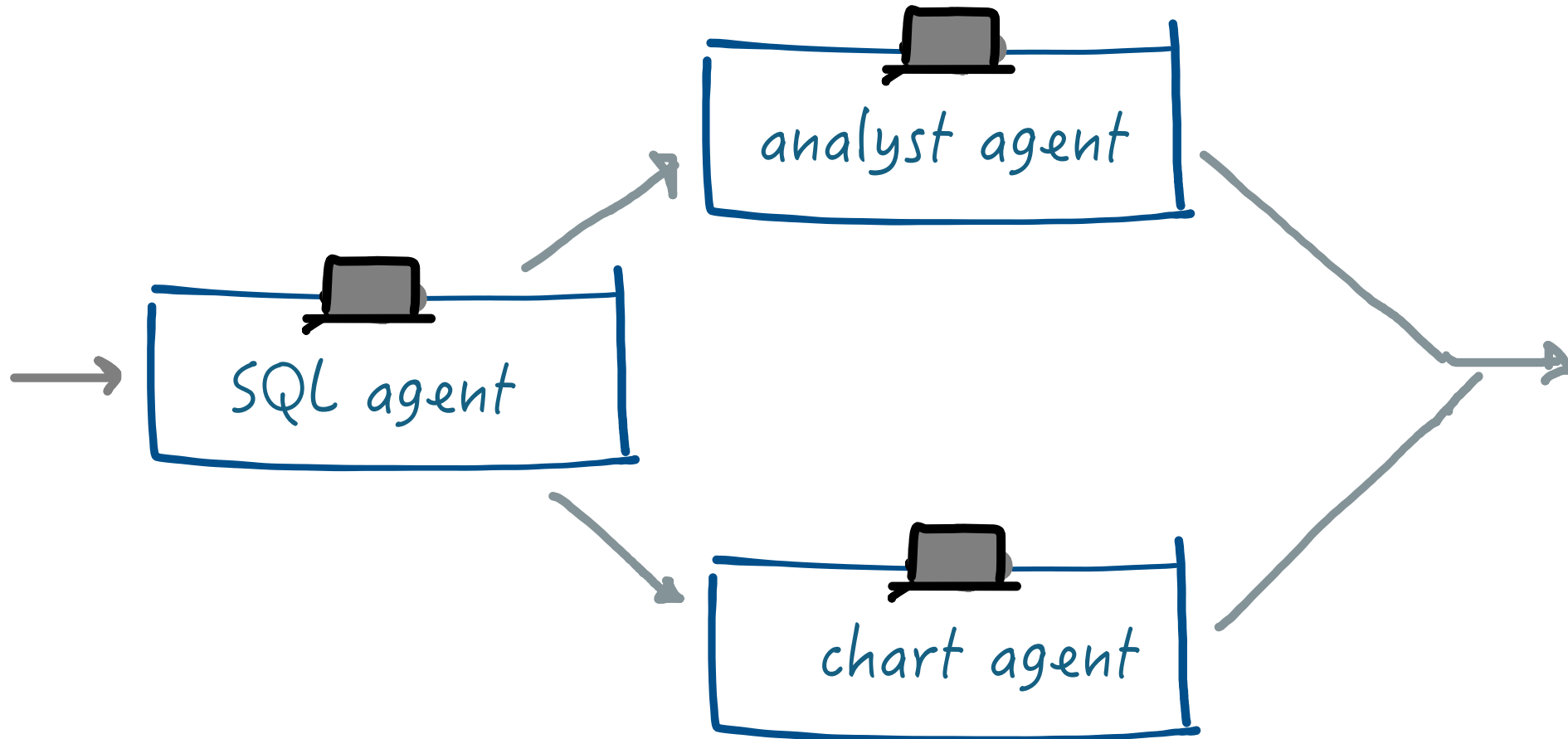
Feature request

We want a written analysis of the data along with the chart

Ideas

- Make the flow deterministic, drop the orchestrator
- Add an Analyst subagent, that could do some web search and cross reference with our data

3 Graph workflow



```
from strands.multiagent import GraphBuilder

builder = GraphBuilder()

# Agents are nodes
builder.add_node(sql_agent, "sql")
builder.add_node(analyst_agent, "analyst")
builder.add_node(visualizer_agent, "visualizer")

# Edges
builder.add_edge("sql", "analyst") # sql → analyst
builder.add_edge("sql", "visualizer") # sql → visualizer

# Analyst & visualizer run in parallel after SQL completes!
graph = builder.build()
result = await graph.invoke_async(request)
```

Evaluation of architecture 3:

✓ Software

Modular

Deterministic execution

Clear interfaces between agents



✓ Efficiency:

Less LLM loops at every interaction



Feature request

Actually, we want a chatbot that can do everything.

Ideas

- Add more nodes and edges



Ideas

- ~~Add more nodes and edges~~



Unmaintainable — every flow must be decided beforehand. New features can impact the whole graph



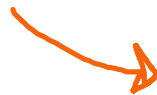
Ideas

- ~~Add more nodes and edges~~



Unmaintainable — every flow must be decided beforehand. New features can impact the whole graph

- Let's have a swarm of agents that work with each other



Because why not, YOLO



Ideas

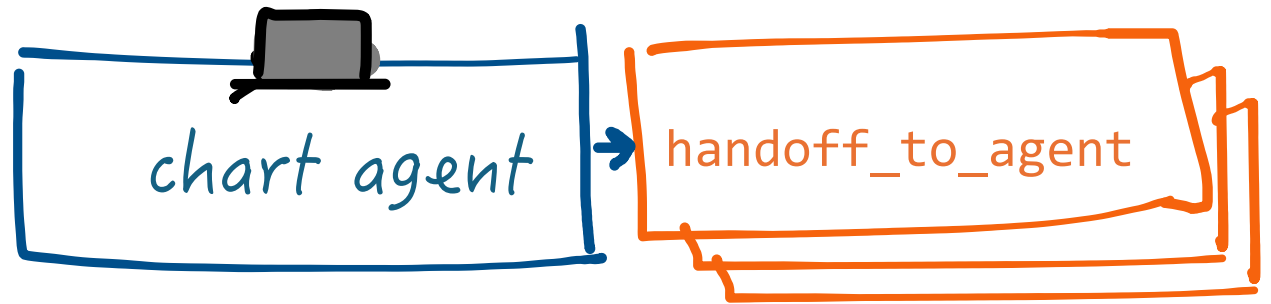
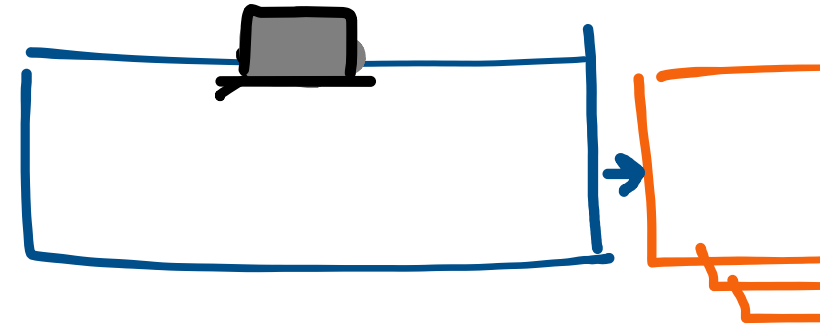
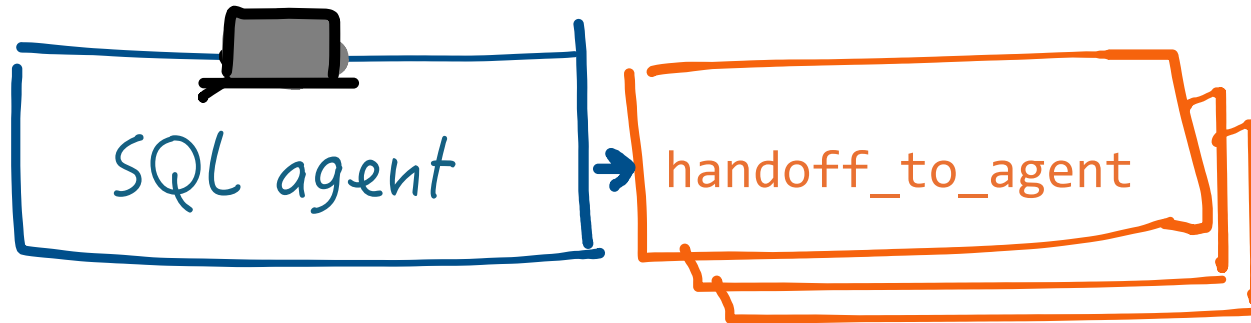
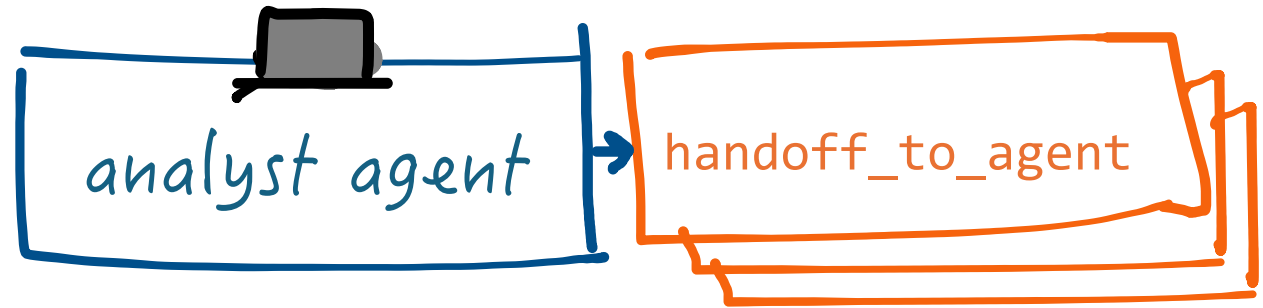
- ~~Add more nodes and edges~~

Unmaintainable — every flow must be decided beforehand. New features can impact the whole graph

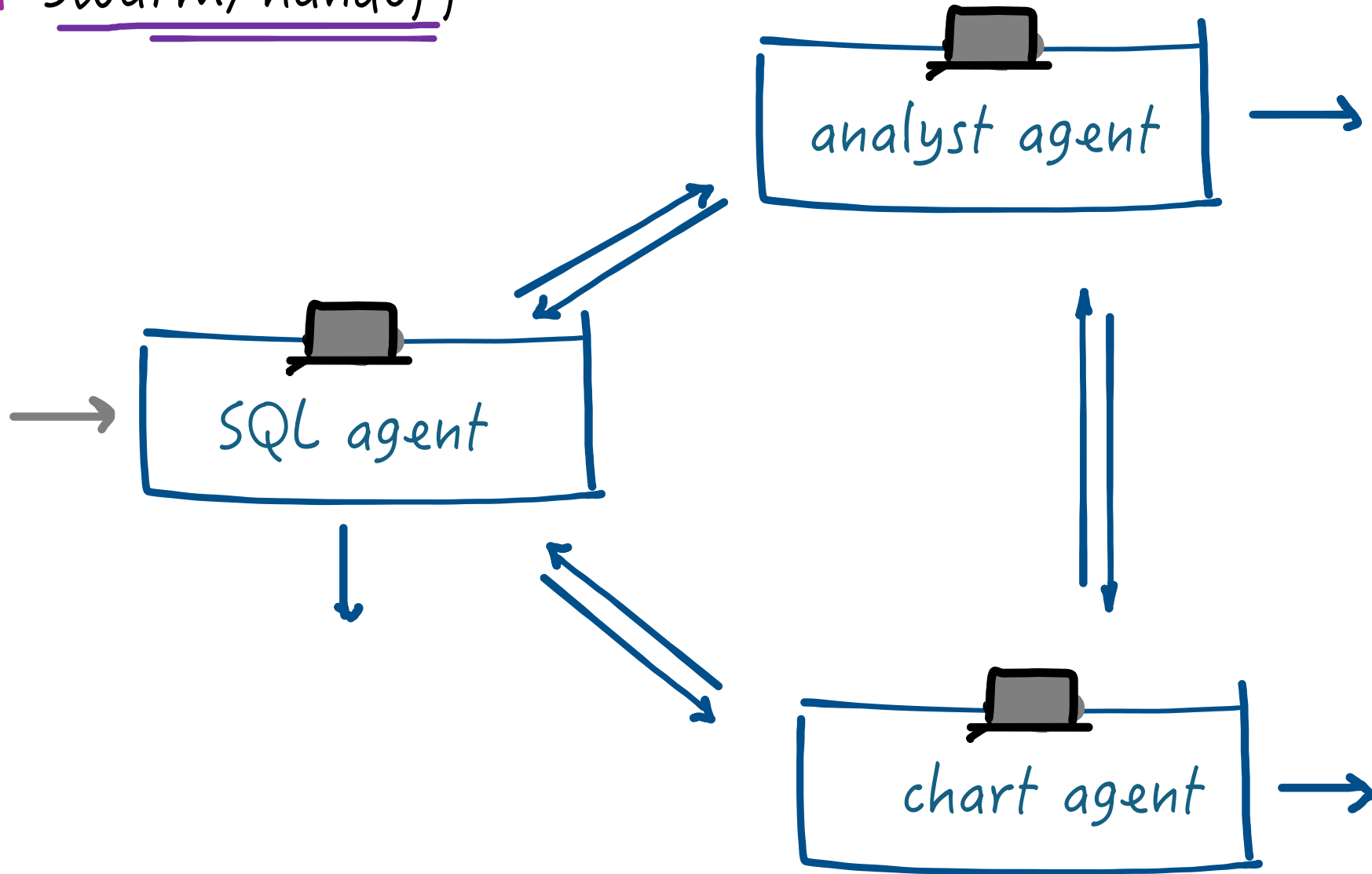
- Let's have a swarm of agents that work with each other

Because why not, YOLO

4 Swarm/handoff



4 Swarm/handoff



```
from strands.multiagent import Swarm

agents = [sql_expert, analyst_expert, chart_expert]

swarm = Swarm(
    agents,
    entry_point=data_expert,
    max_handoffs=15
)

result = swarm(request)
```

(demo)

Evaluation of architecture 4:

✓ Results

Autonomy and adaptability
Very powerful



Evaluation of architecture 4:

BUT



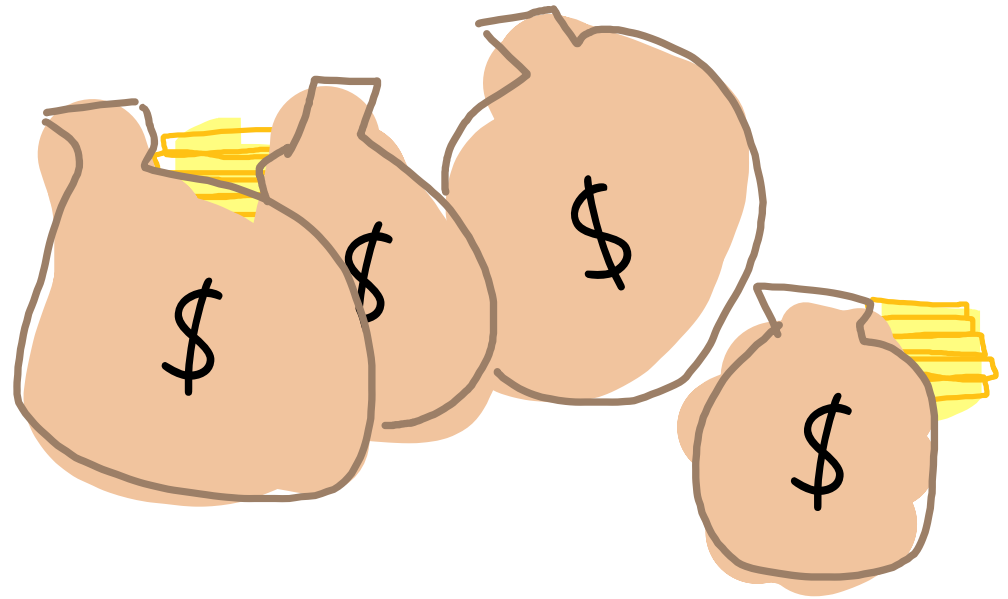
We notice that “thank you” can trigger a swarm of agents

Extremely slow and expensive for simple queries

your POC:



your Prod:



Evaluation of architecture 4:

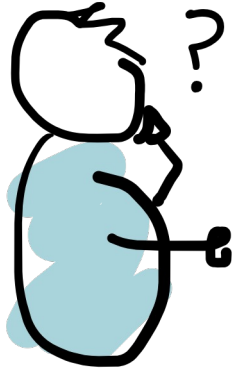
BUT



We notice that “thank you” can trigger a swarm of agents

Extremely slow and expensive for simple queries

Security and control stop being a property of the system
and can become a project of their own



What about security ?

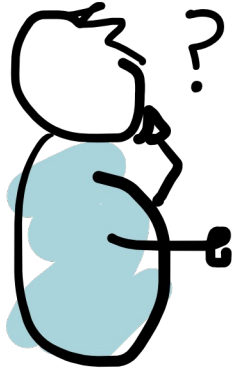


Prompt injection

Behavioral manipulation

System exposure

Unauthorized data access and modification



What about security ?



Prompt injection

Injection travels
through handoffs

Behavioral manipulation

System exposure

Unauthorized data access and modification

Blast radius scales with topology

Takeaways

Audit the handoffs, not
just the answer

Least privilege per agent

Pass references, not payloads

Guard-rails:
Hard limits for handoffs, loops, spend



Takeaways

Nothing raw goes in

Nothing runs forever

Nothing decides its own permissions

Nothing happens unlogged.





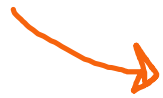
Ideas

- Use orchestrator + experts mode, but let the main agent spawn sub agents on demand
- Use a (virtual) file system for shared memory, and let agents write scripts and execute them
- ...



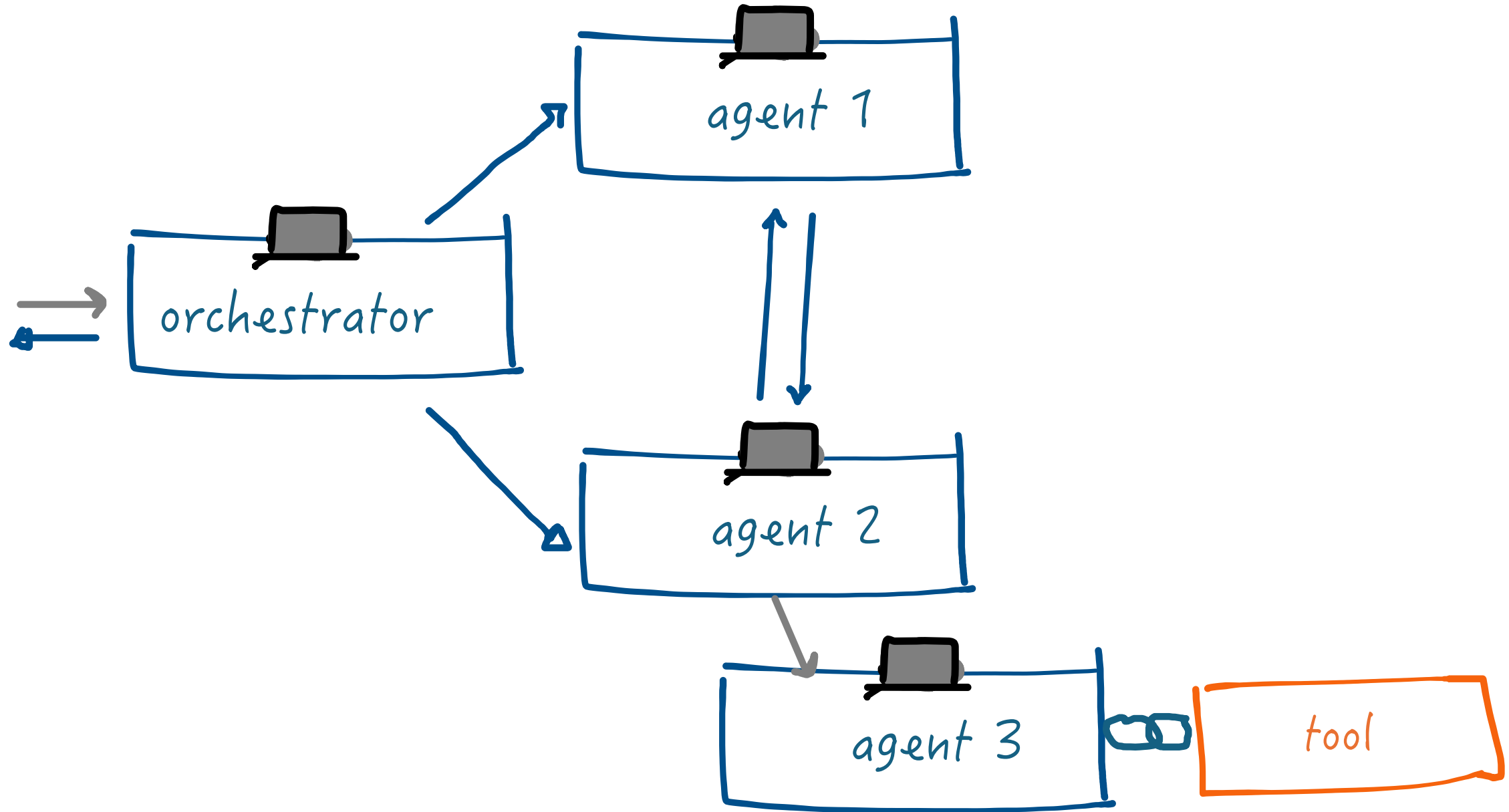
Ideas

- Let's sit down with our PM and understand better what are the actual needs



Because we should probably have started with that

? Hybrid architecture



Useful resources

LangGraph — Multi-agent systems

langchain-ai.github.io/langgraph/concepts/multi_agent/



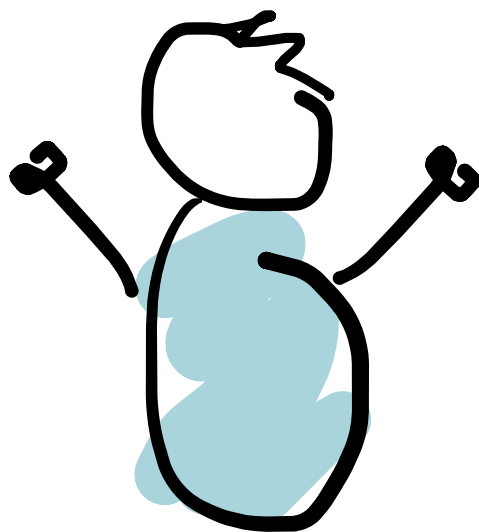
Anthropic — Building effective agents

www.anthropic.com/engineering/building-effective-agents

Strands — Multi-agents patterns

strandsagents.com/docs/user-guide/concepts/multi-agent/multi-agent-patterns/

Let's wrap it up!



more agents, more ...

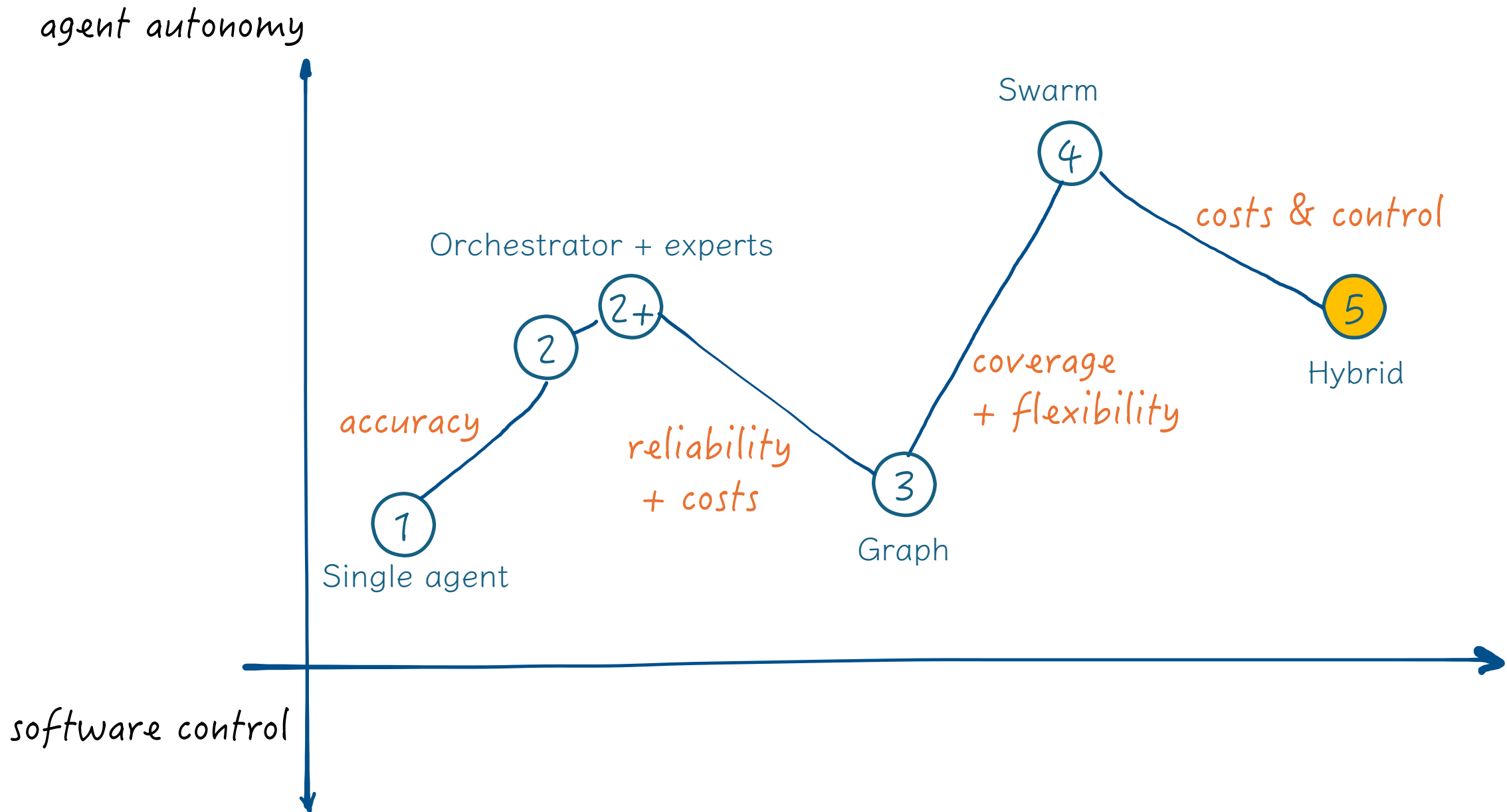
capabilities
modularity
flexibility
accuracy

problems

with cost

with context management

with control



Key takeaways

latency
context = the crux of it all
cost



Reduce context bloat -pass references, use ephemeral subagents, have a shared memory, ...

Bound the loops -hard limits on handoffs, loops, spend
Audit the loops -trace every call, not just the final answer

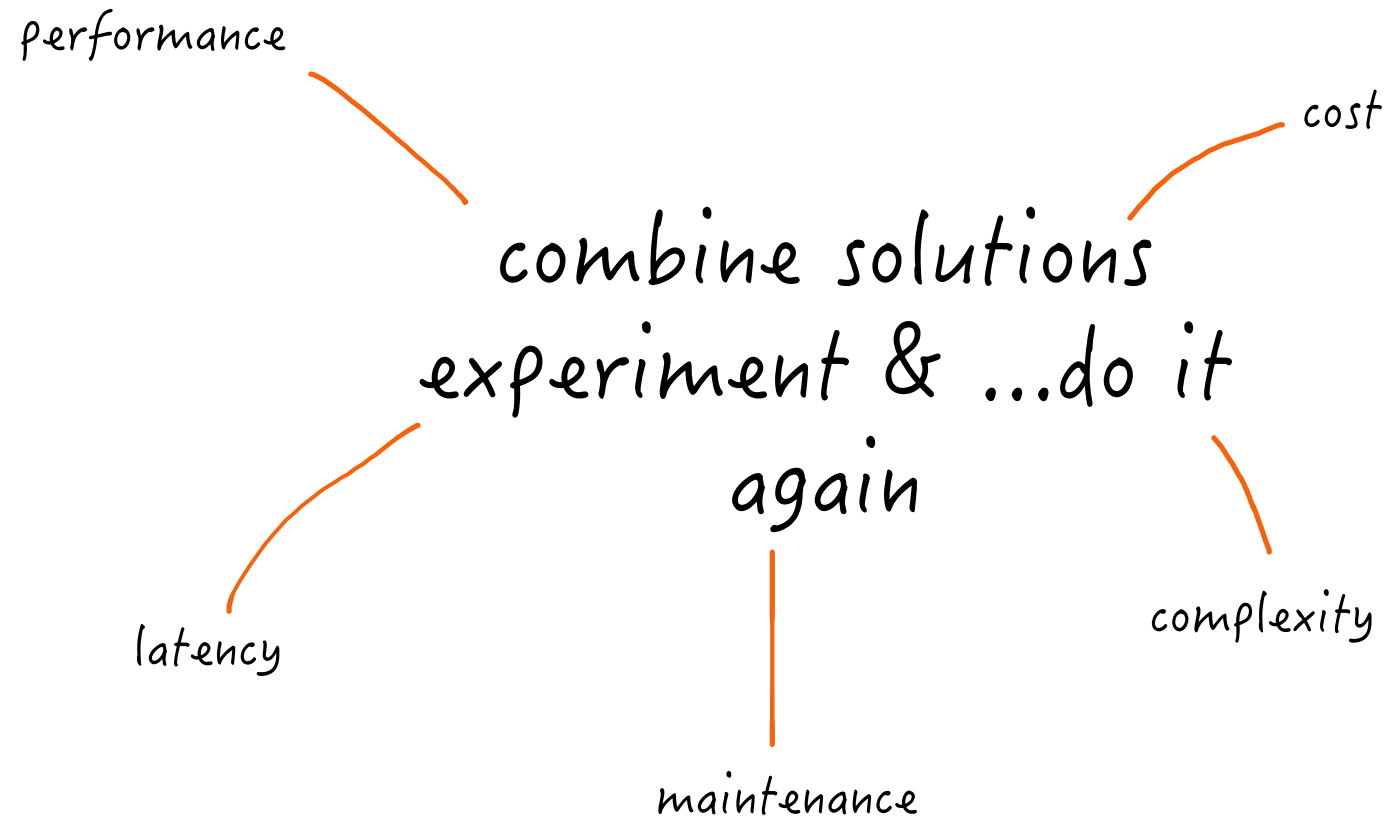
Key takeaways

Spend autonomy where judgment is actually needed

Easier to move from a controlled system to an autonomous one than the reverse



Key takeaways



linkedin.com/in/mblete
mariealiceblete.com

AI engineering landscape
malywut.github.io/ai-engineering-landscape/

contact

QUESTIONS?

ASK ME IN THE APP



ressources

Free access to O'Reilly platform (trial)
learning.oreilly.com/get-learning/?code=DEVGPT24