IMEN891K / IDSC501 / AXGS-501 · DATA ENGINEERING · FALL 2026

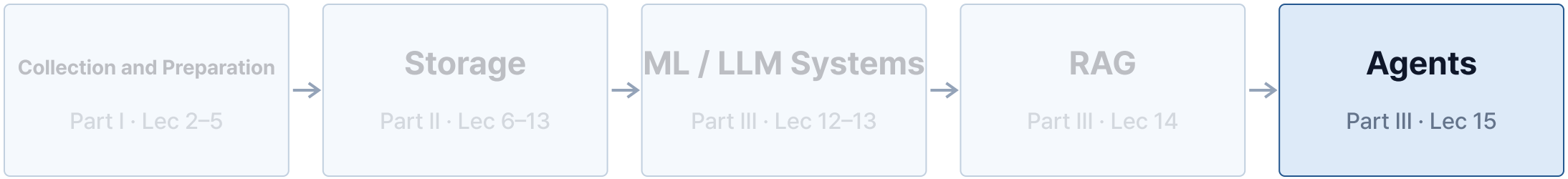
17.

Agents

Sangwoo Mo · POSTECH

Industrial & Management Engineering

Where We Are



Part III connects stored data to models, retrieval, and actions

Outline: Agents

01

How an Agent Takes Actions

choose and run tools to complete a task

02

Controlling Workflows and State

guide the steps and resume after failures

03

Connecting Agents to Data

query, change, and protect stored records

01 · How an Agent Takes Actions

01

How an Agent Takes Actions

choose and run tools to complete a task

02

Controlling Workflows and State

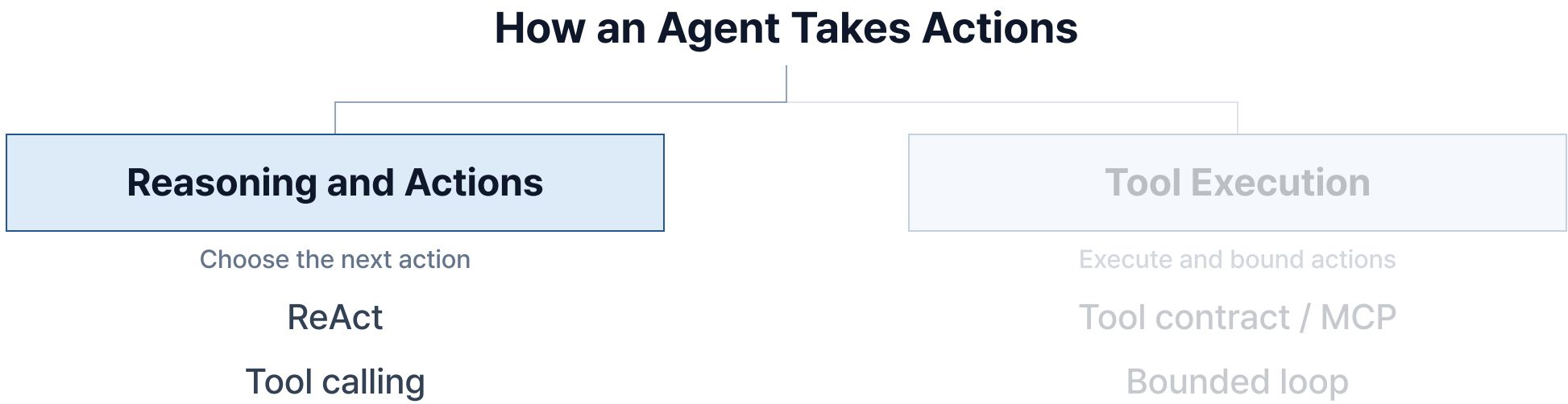
guide the steps and resume after failures

03

Connecting Agents to Data

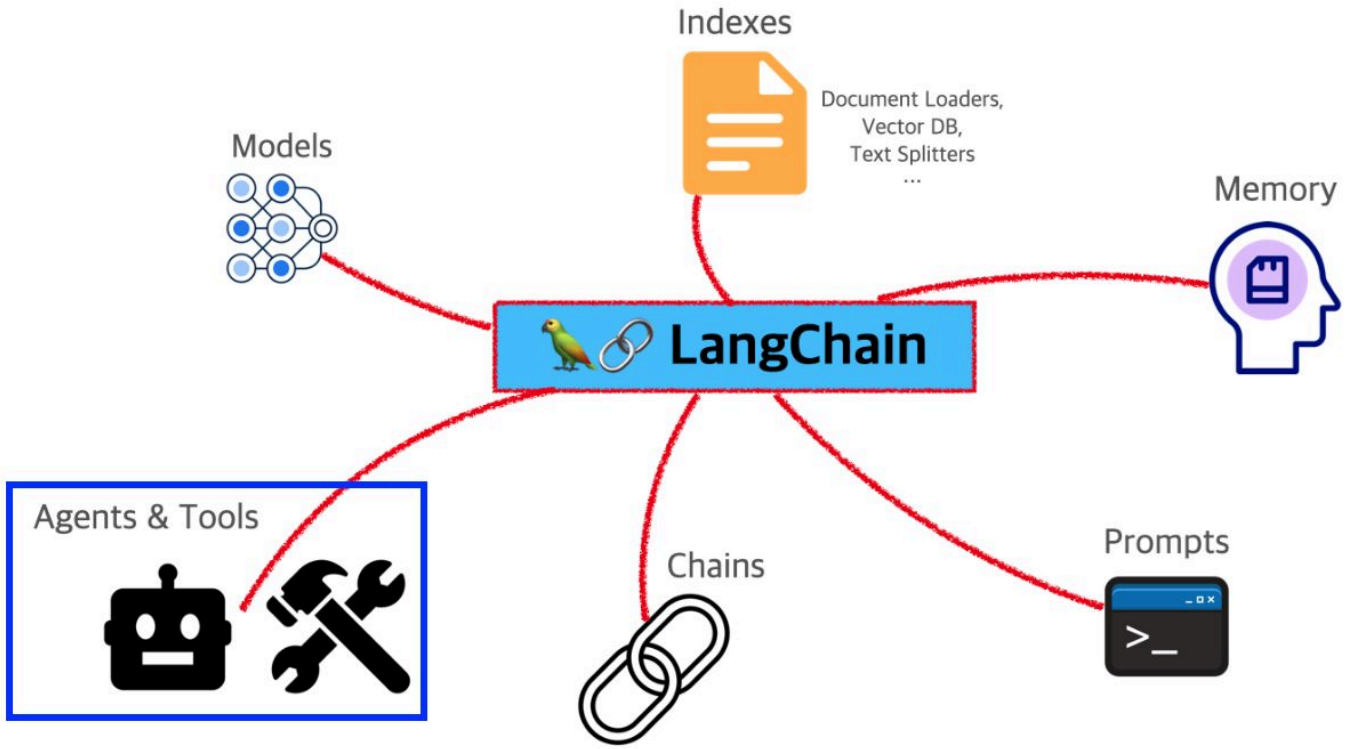
query, change, and protect stored records

§ Reasoning and Actions



Recap: Retrieval and Tools

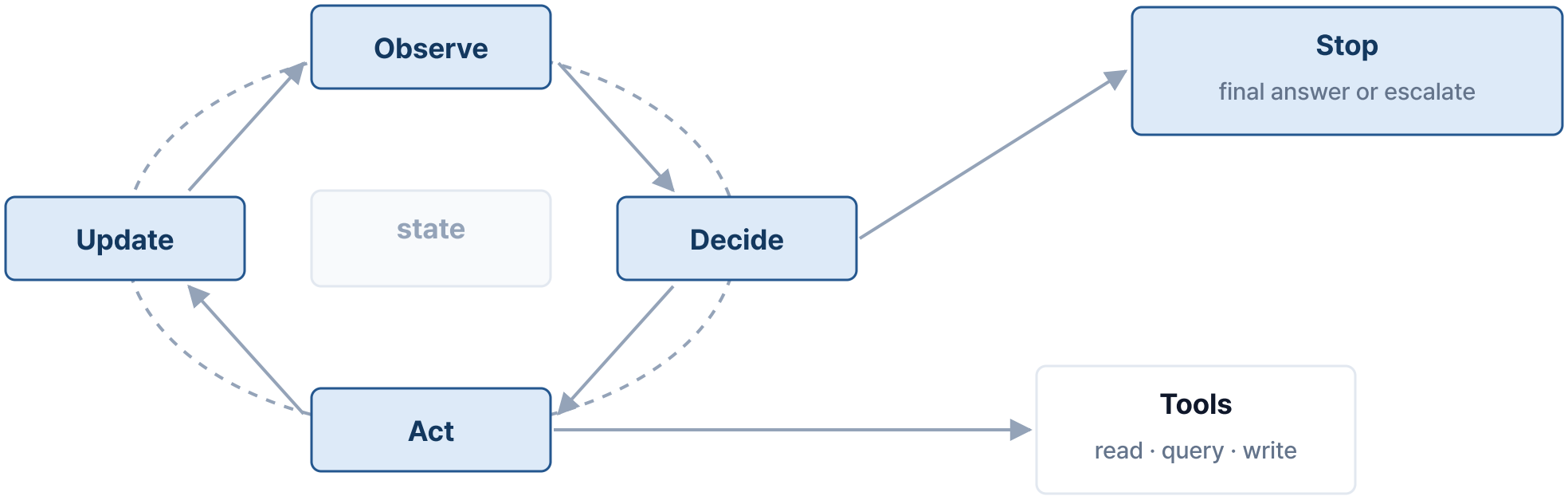
A retrieval pipeline fixes the search step before generation



Agents add **tool use** to the components seen so far

Agent

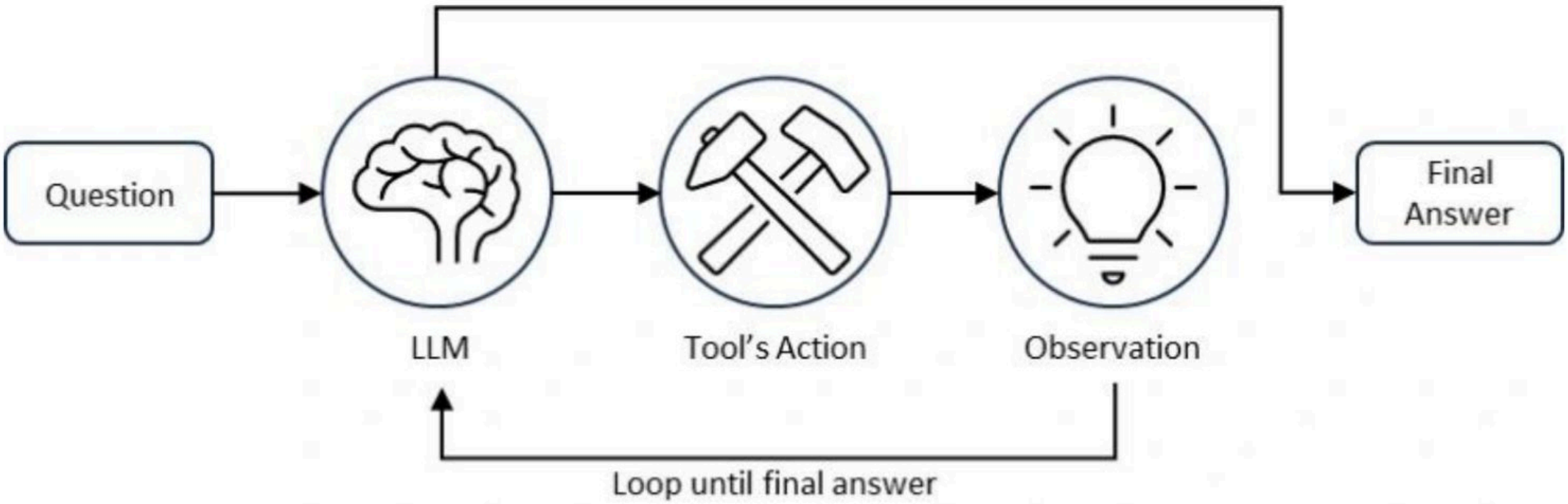
An agent uses a model to choose actions toward a task



The loop repeats until a **stop rule** fires

ReAct

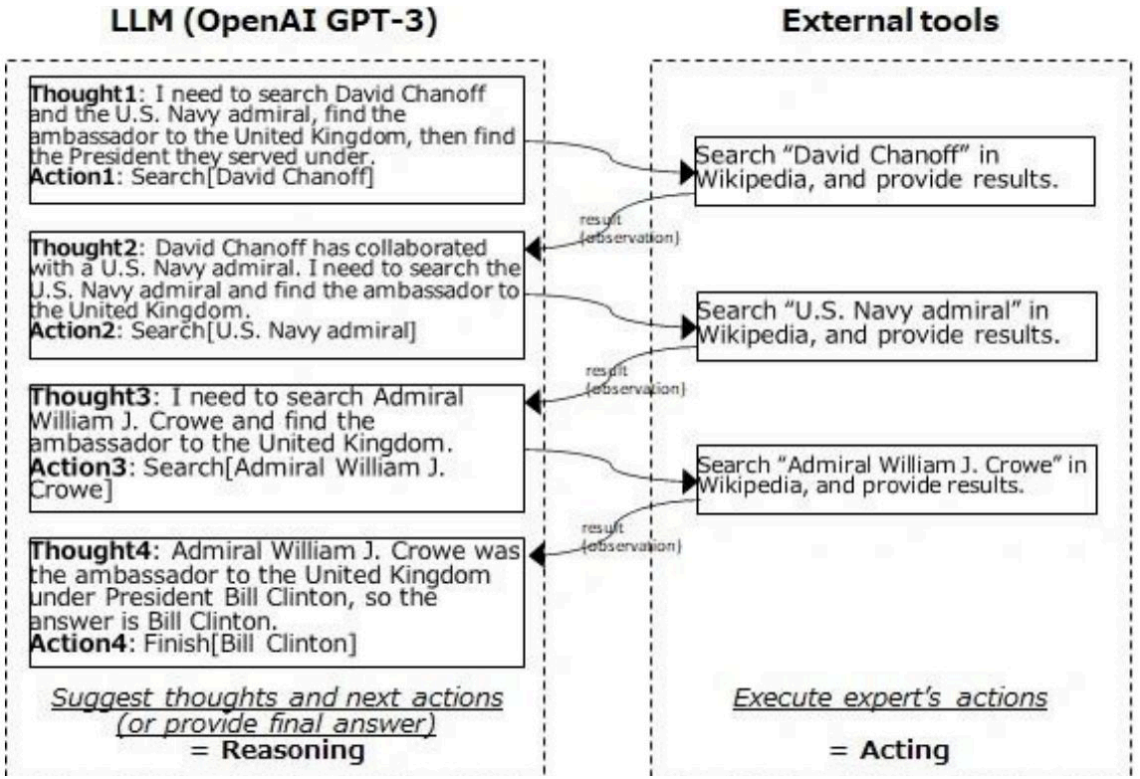
ReAct alternates reasoning, actions, and observations



An observation can change the next action

ReAct Example

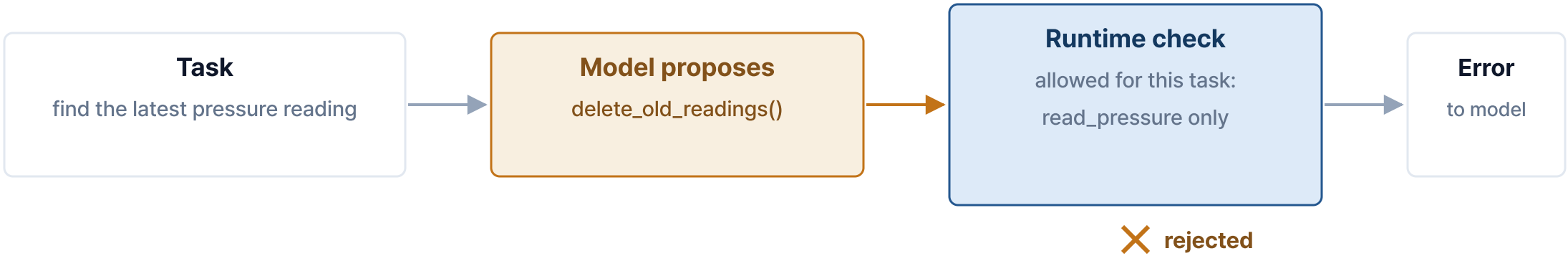
Tool observations provide facts that the model can use



The answer depends on observations from external tools

Caveat: Unsupported Action

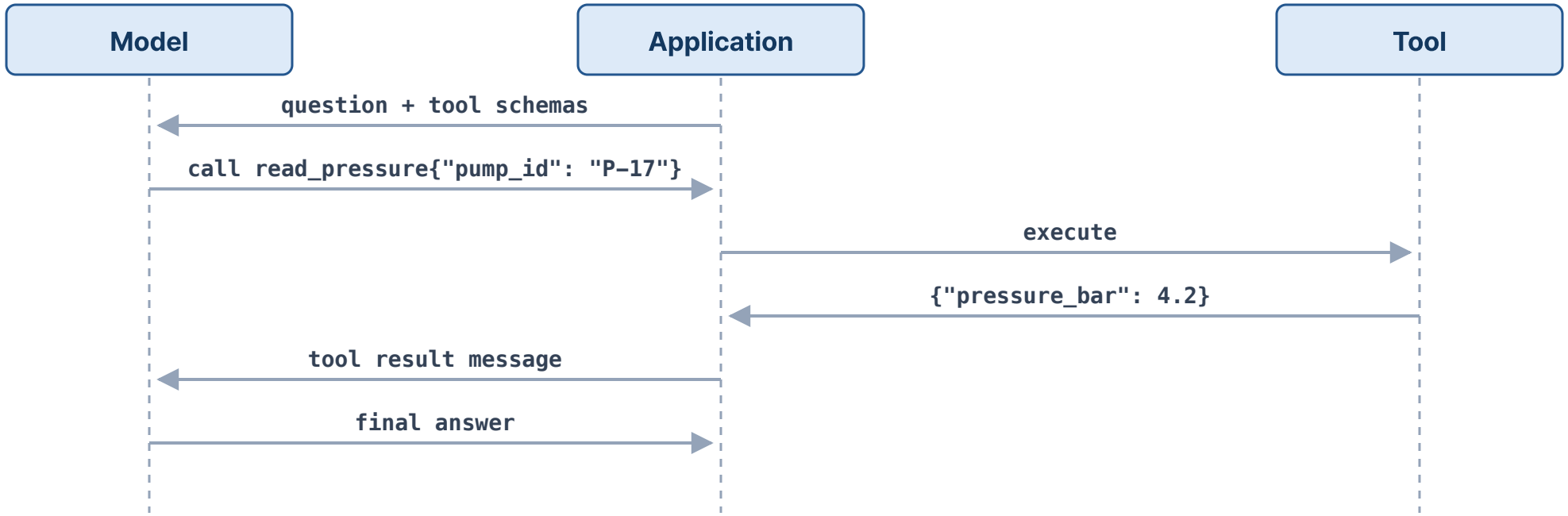
A model can request a tool that does not match the task



The runtime, not the model, decides what **may run**

Tool Calling

The model proposes a structured tool name and arguments



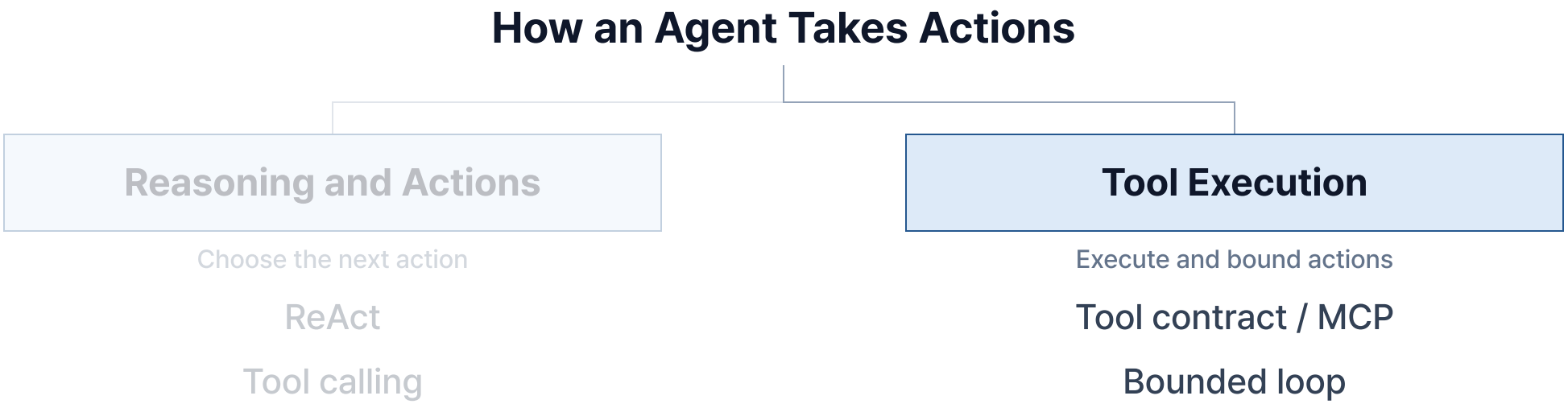
The model asks; the **application** executes

Tool Definition

A typed LangChain tool executes one bounded read

```
const readPressure = tool(({pump_id}) => {
  const row = readings[pump_id];
  if (!row) throw Error("Unknown pump");
  return {pump_id, ...row};
}, {name:"read_pressure", description:"Read one pump pressure in bar",
  schema:z.object({pump_id:z.enum(["P-17","P-18"])})});
return await readPressure.invoke({pump_id:"P-17"});
```

§ Tool Execution



Tool Contract

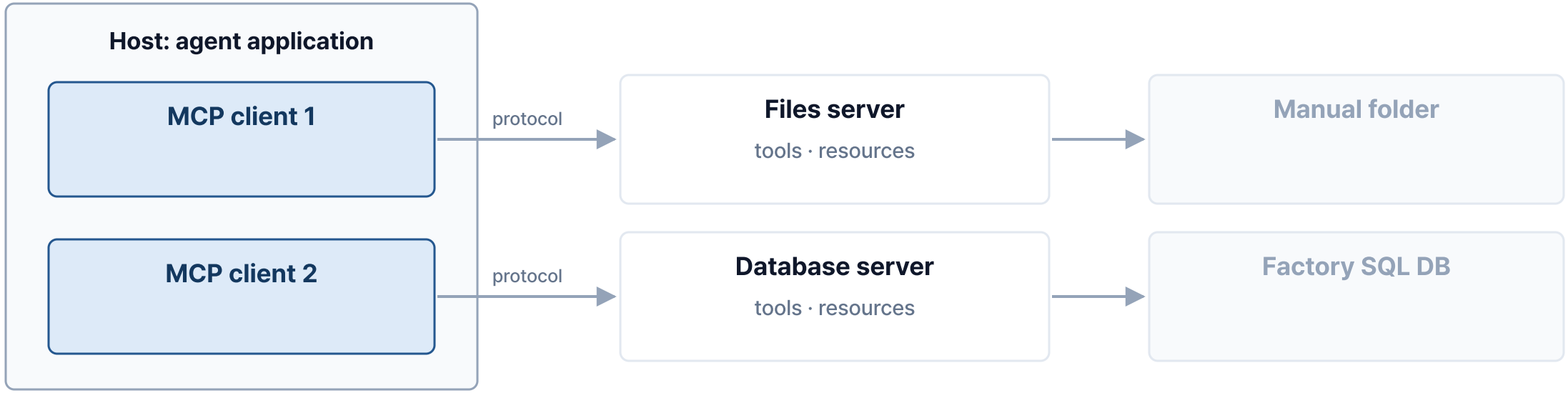
Specify what a tool accepts, changes, and returns



Four facts every tool must **declare**

Model Context Protocol

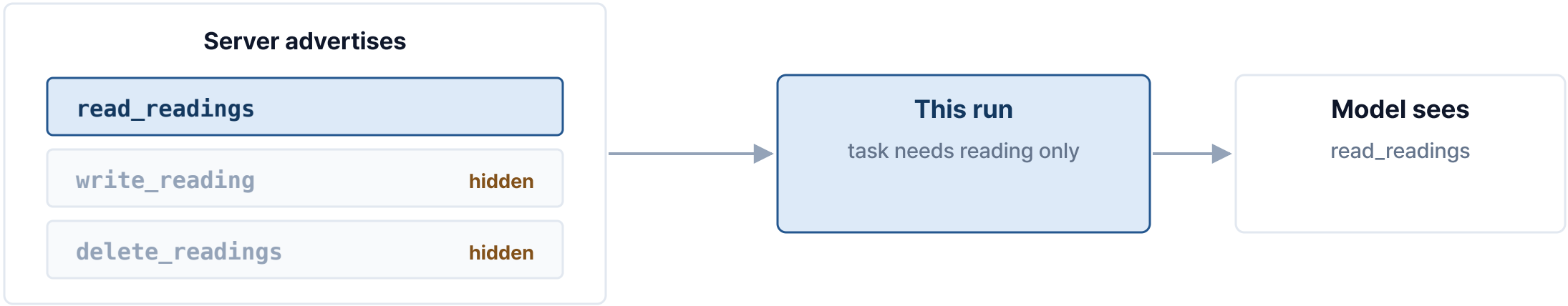
MCP standardizes how applications connect to capability servers



One protocol between the host and every **capability server**

Caveat: MCP and Permission

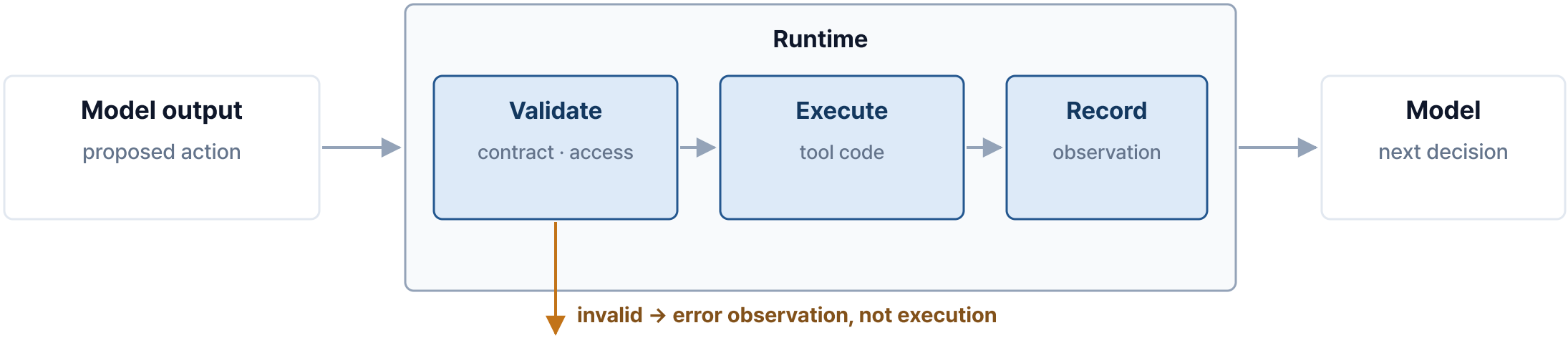
A connected server may expose more capability than a task needs



Expose only the tools **this task** needs

Agent Runtime

The runtime owns execution, validation, and state updates



Invalid actions return as errors, not **executions**

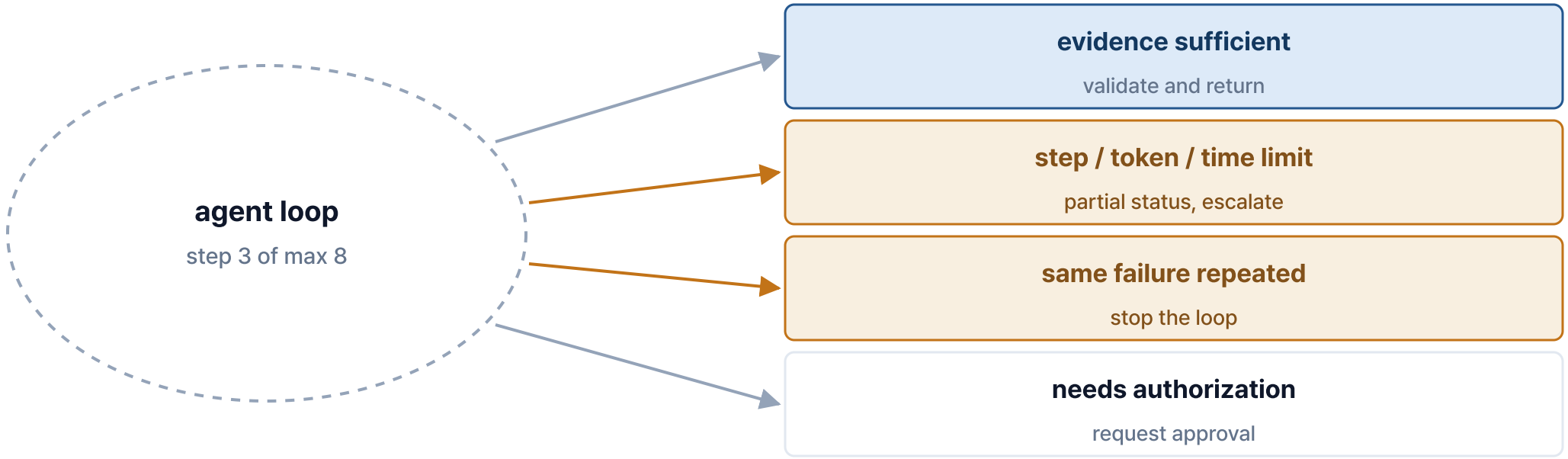
Bounded Agent Loop

Observe a model decision, tool execution, and the next decision

```
const model = makeModel([
  '{"tool":"read_pressure","pump_id":"P-17"}',
  '{"final":"P-17 pressure is 4.2 bar;
threshold 5 bar."}'
]);
const messages = [new HumanMessage("Check P-17
pressure. Return JSON tool or final.")];
const trace = [];
for (let step=0; step<3; step++) {
  const raw = await model.invoke(messages);
  const decision = await new
JsonOutputParser().invoke(raw);
  trace.push({step,decision});
  if (decision.final) return {mode:MODEL_MODE,
trace};
  if (decision.tool !== "read_pressure") throw
Error("Unknown tool");
  const observation = await
pressureTool.invoke({pump_id:decision.pump_id})
;
  trace.push({observation});
  messages.push(raw, new
HumanMessage(JSON.stringify(observation)));
}
throw Error("Step budget exceeded");
```

Stopping Rules

Stop on completion, limits, or an unresolved failure



Four ways a loop ends; only one is **success**

Demo: Bounded Agent Loop

Inspect the decision and observation sequence

1. Request P-18 instead of P-17. Compare the observation with the recorded final response.
2. Replace the final response with another tool request. Observe the step limit.

Execution Offline replay | Installed model na | http://127.0.0.1:11434

```
Type code or click Starter
```

```
Run code to inspect its output.
```

Run | Starter | Reset

Ready

LangChain.js · JavaScript · Ctrl/Cmd+Enter · Offline replay uses fixed responses; local mode calls your model

Answer: Bounded Agent Loop

The baseline output comes from executing the shown code


```
{"mode": "Recorded response (no inference)", "trace": [{"step": 0, "decision": {"tool":
"read_pressure", "pump_id": "P-17"}}, {"observation": {"pump_id": "P-17",
"pressure_bar": 4.2, "age_minutes": 2}}, {"step": 1, "decision": {"final": "P-17
pressure is 4.2 bar; threshold 5 bar."}}]}
```

02 · Controlling Workflows and State

01

How an Agent Takes Actions

choose and run tools to complete a task

02

Controlling Workflows and State

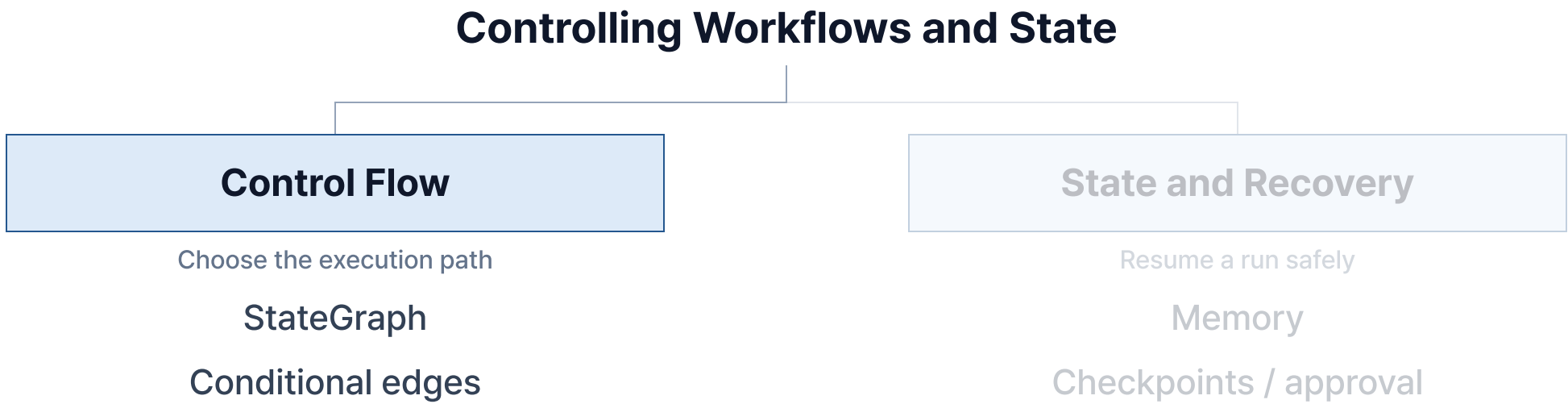
guide the steps and resume after failures

03

Connecting Agents to Data

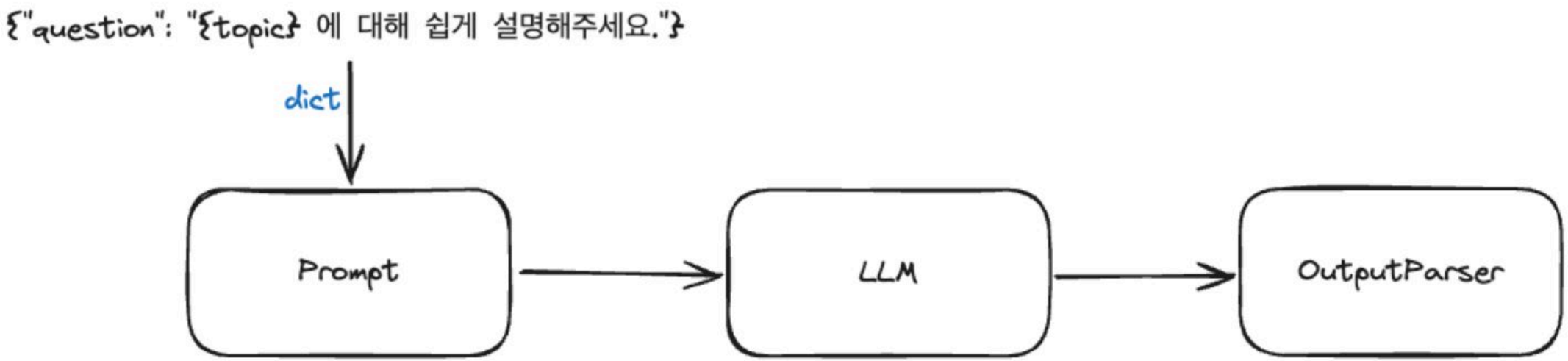
query, change, and protect stored records

§ Control Flow



Chain Composition

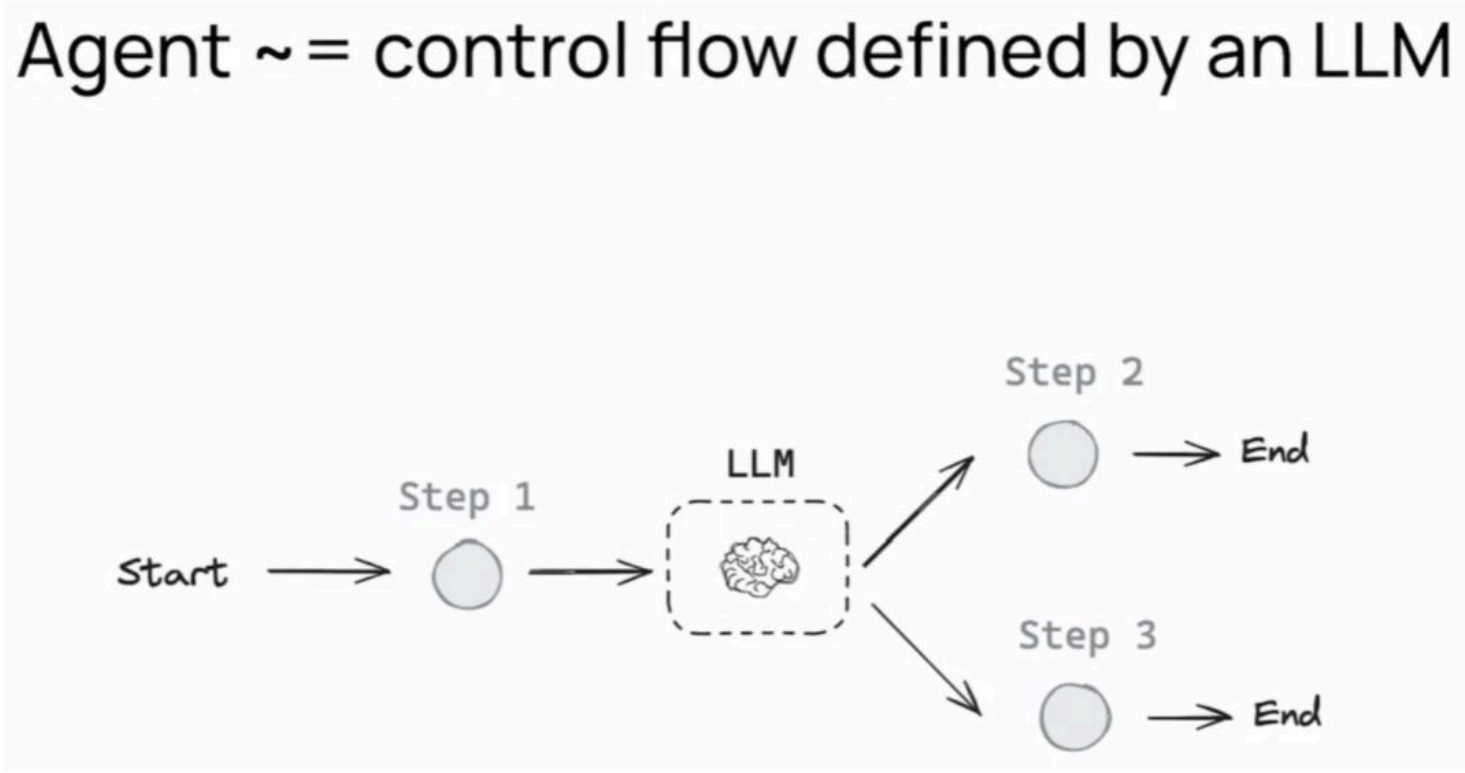
A chain fixes the order of prompt, model, and parsing stages



Fixed composition is suitable when the dependencies are known

Adaptive Control Flow

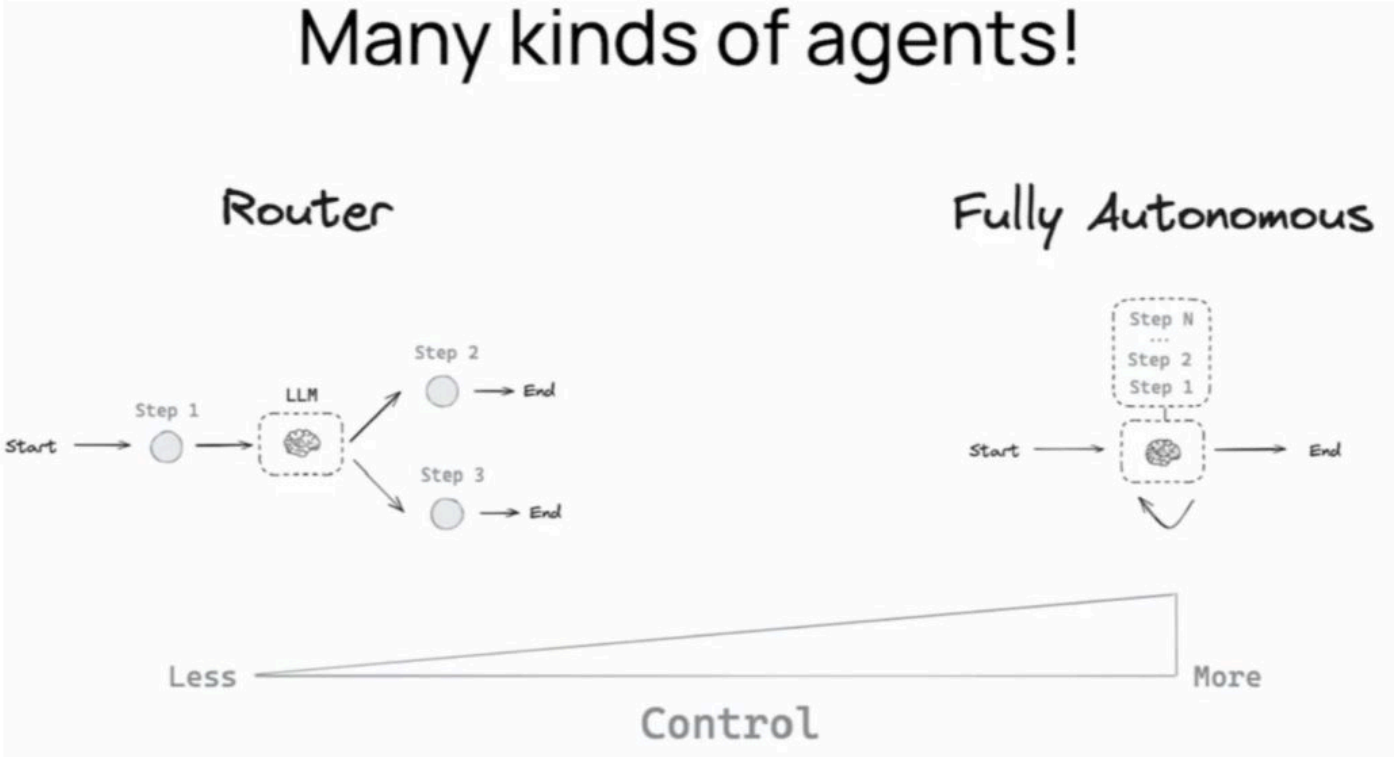
An agent may choose a different next step after observing a result



The observation changes which branch runs

Kinds of Agents

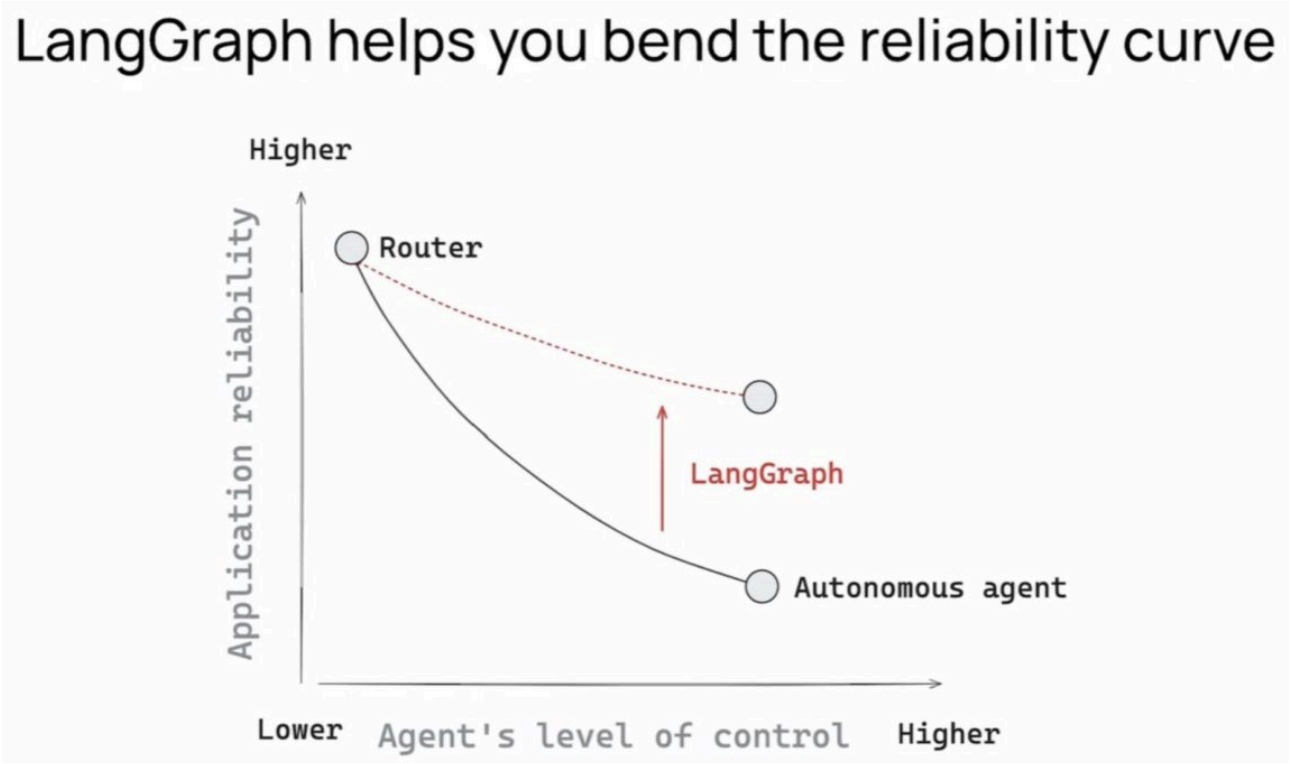
Agents differ in how much of the control flow the model decides



A router picks **one branch**; an autonomous agent picks every step

Control and Reliability

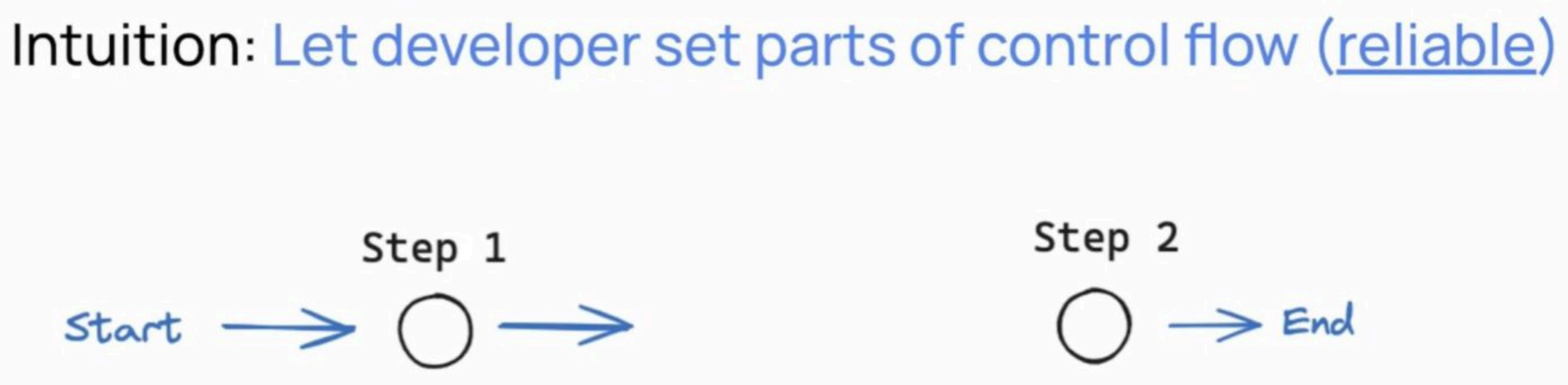
Giving the model more control tends to lower application reliability



A developer-defined graph keeps **model choices** inside fixed structure

Controlled Workflow

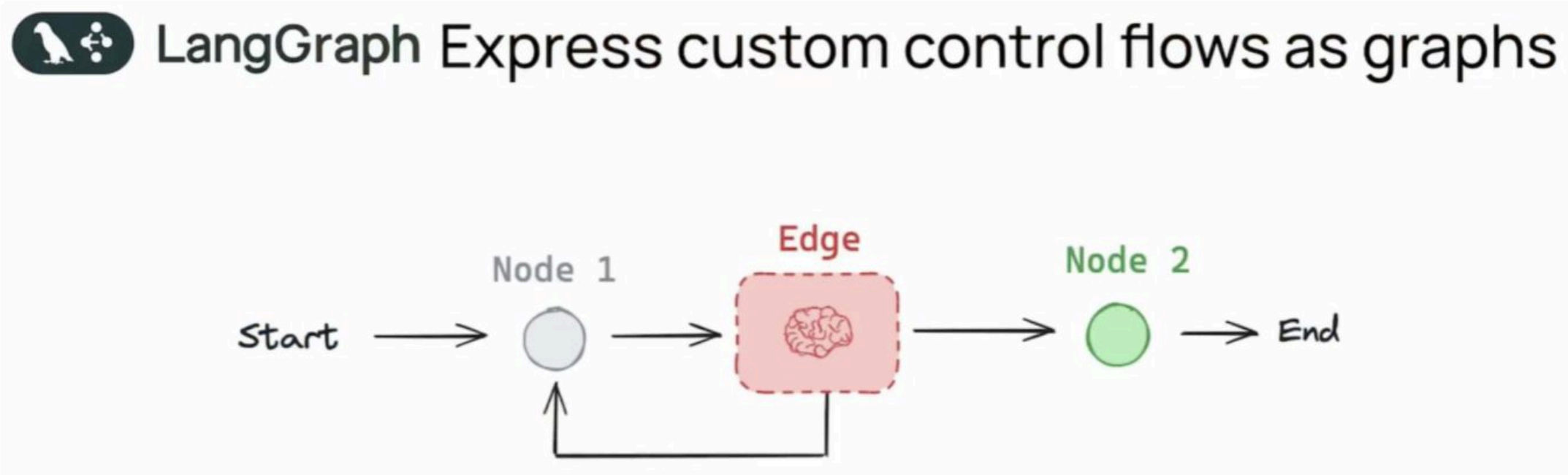
Developers define parts of the control flow explicitly



Some steps follow a fixed path

LangGraph

LangGraph represents stateful workflows with nodes and edges



Nodes update state; edges select the next node

Workflow Graph

Use conditional edges to send stale readings to review

```
const State = Annotation.Root({pump_id:Annotation(), reading:Annotation(),
  outcome:Annotation()});
const graph = new StateGraph(State)
 .addNode("read", async s => ({reading:await pressureTool.invoke({pump_id:s.pump_id})}))
 .addNode("report", s => ({outcome:s.reading.pressure_bar > 5 ? "alarm" : "normal"}))
 .addNode("review", () => ({outcome:"stale; review required"}))
 .addEdge(START,"read")
 .addConditionalEdges("read", s => s.reading.age_minutes > 10 ? "review" : "report")
 .addEdge("report",END).addEdge("review",END)
 .compile();
const result = await graph.invoke({pump_id:"P-17"});
return {reading:result.reading, outcome:result.outcome};
```

Demo: Stateful Inspection Graph

Change the pump and inspect the selected state transition

1. Run P-18. Which branch handles the stale reading?
2. Change the freshness limit from 10 to 60. Does the outcome change?

```
Type code or click Starter
```

```
Run code to inspect its output.
```

Run | Starter | Reset

Ready

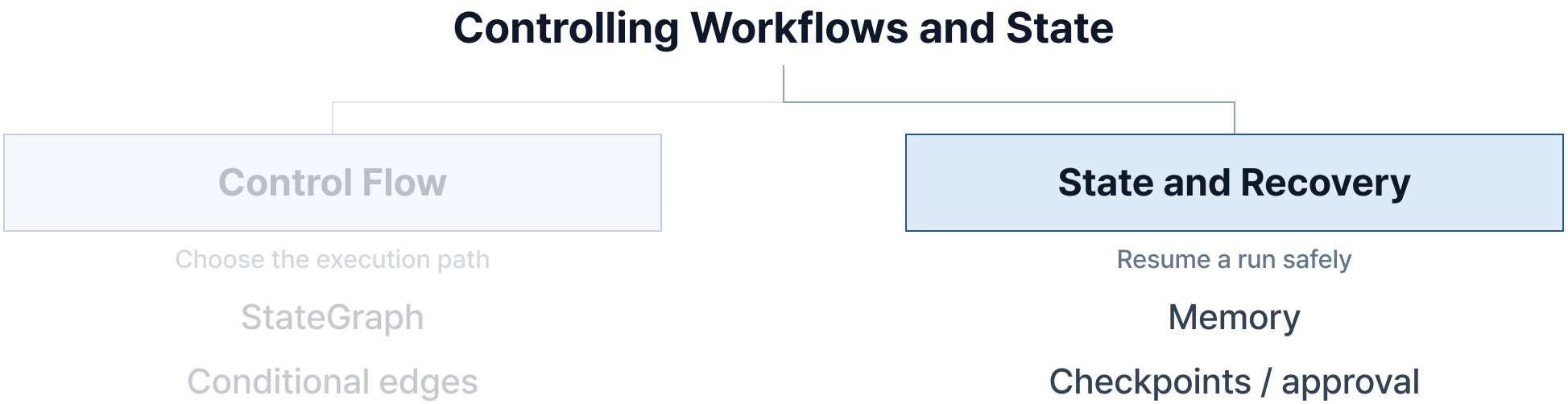
LangChain.js · JavaScript · Ctrl/Cmd+Enter · Real library execution with local data

Answer: Stateful Inspection Graph

The baseline output comes from executing the shown code

```
{
  "reading": {
    "pump_id": "P-17",
    "pressure_bar": 4.2,
    "age_minutes": 2
  },
  "outcome": "normal"
}
```

§ State and Recovery



Agent State

State contains the information needed to continue the current run

Run state

`messages`	request, tool observations
`task`	pump_id P-17, window 24 h, pending action
`progress`	read done; compare pending
`control`	budget 5 of 8, retries 1, approval none

model context for this call

the runtime keeps the rest

The model sees a **slice** of the run state

Memory Types

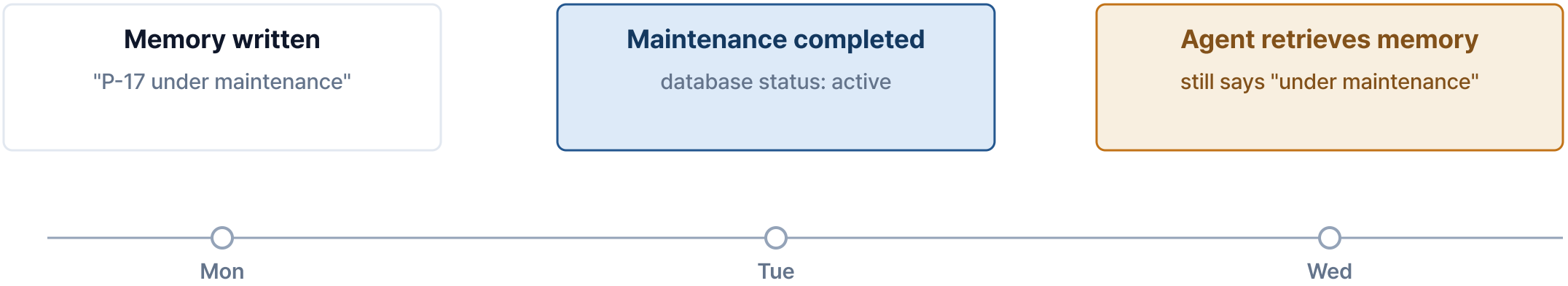
Different retained information supports different decisions

Type	Contains	Example
Working state	Current task progress	A pending tool call
Semantic memory	Stored facts	Preferred report format
Episodic memory	Past events and outcomes	A resolved maintenance case
Procedural memory	Reusable instructions	A validated inspection procedure

Choose what to remember before choosing where to store it

Caveat: Stale Memory

A stored fact can outlive the condition that made it true

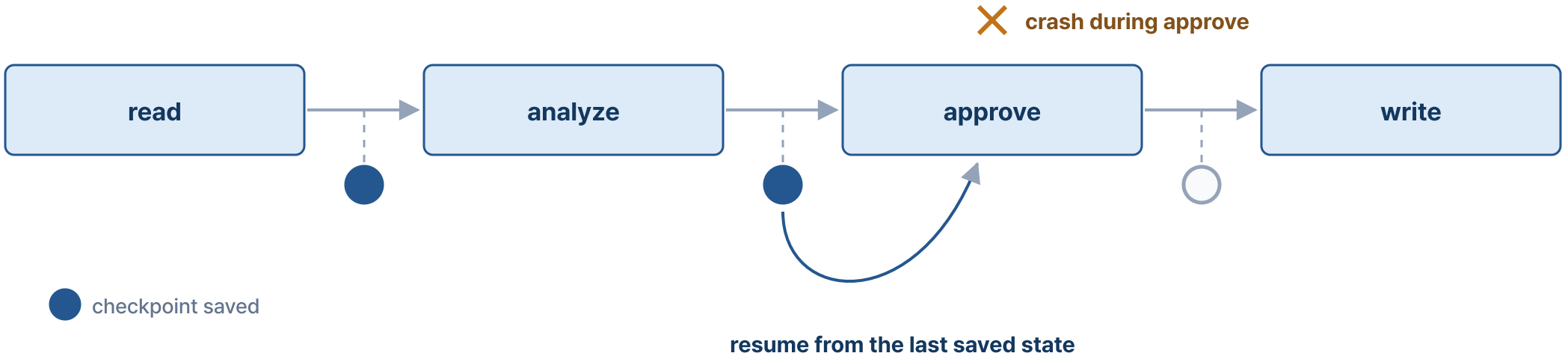


Store time and source with the fact; recheck the system of record

A remembered fact outlived the **condition**

Checkpoint

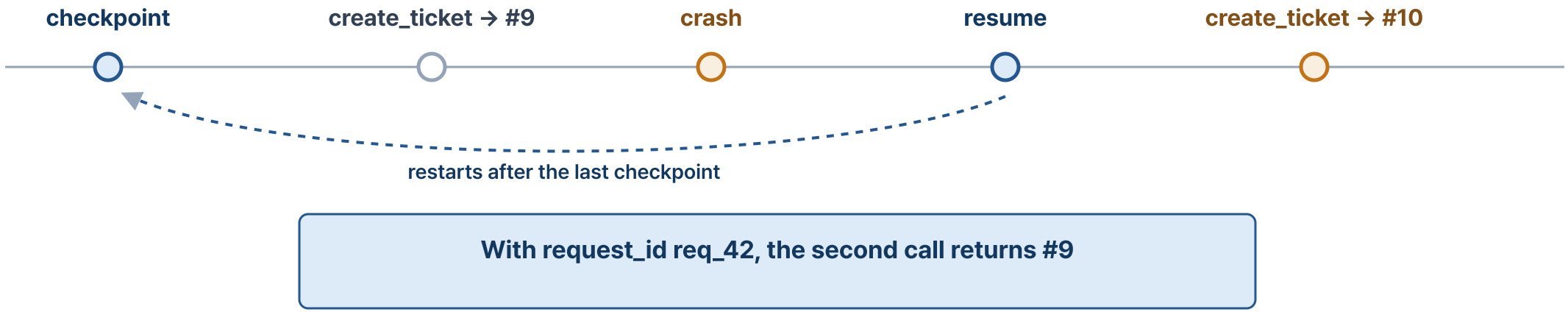
A checkpoint saves the state needed to resume a workflow



A checkpoint after each node limits **repeated work**

Caveat: Resume and Side Effects

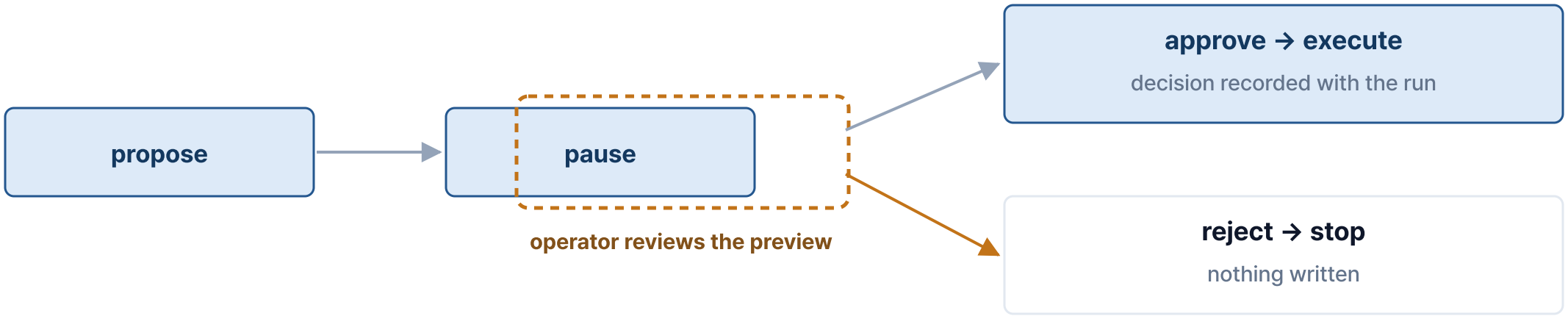
Resuming a workflow can repeat an external write



Resuming repeated a **write**

Approval Boundary

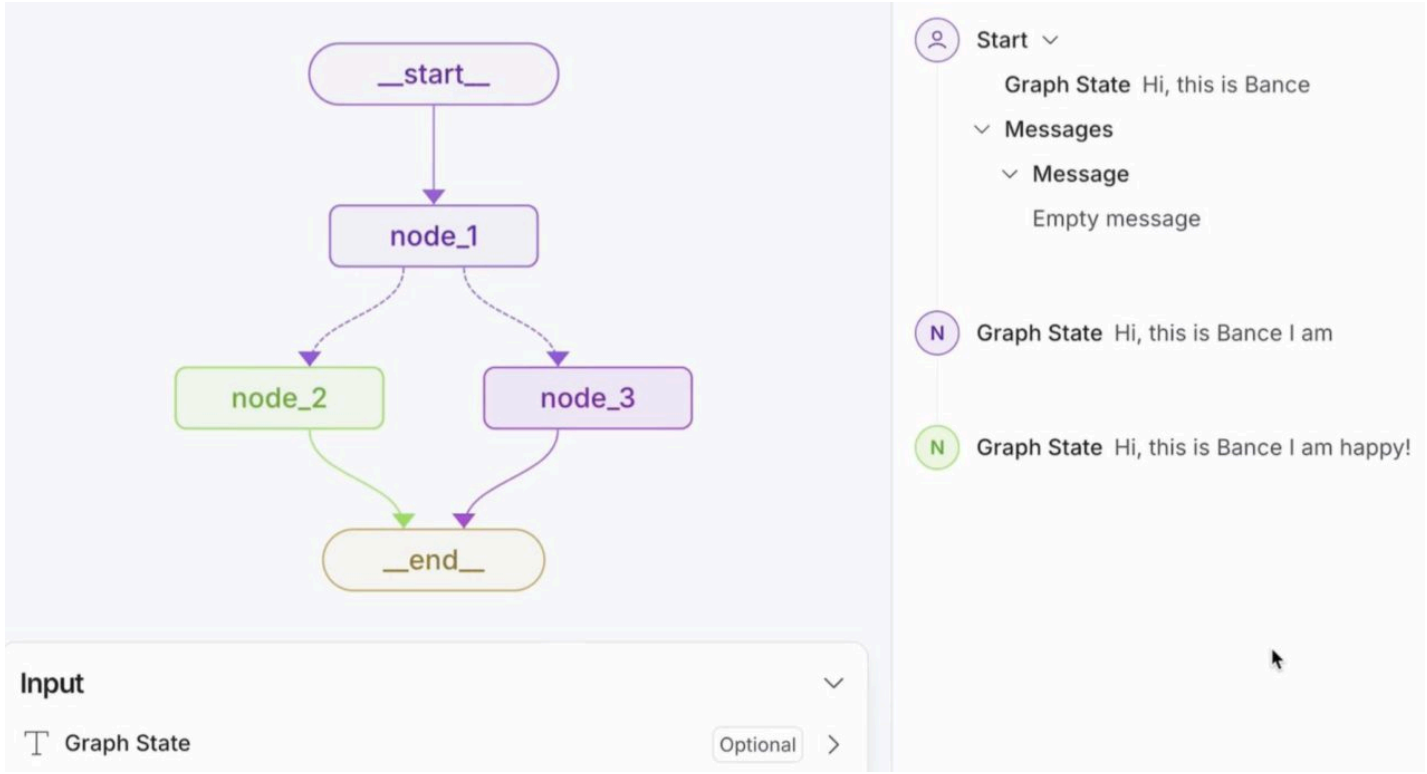
Pause before a consequential action and resume with a recorded decision



The run waits for a **recorded** decision

Agent Trace

Trace actions and observations instead of relying on the final narrative



The trace shows each node, **state** update and message

03 · Connecting Agents to Data

01

How an Agent Takes Actions

choose and run tools to complete a task

02

Controlling Workflows and State

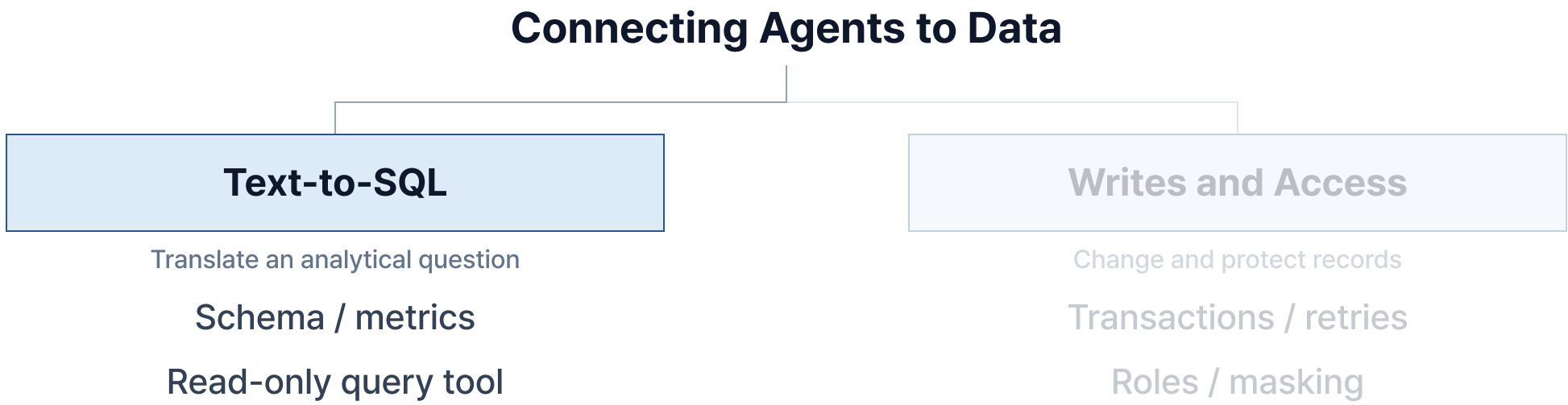
guide the steps and resume after failures

03

Connecting Agents to Data

query, change, and protect stored records

§ Text-to-SQL


Connecting Agents to Data
Text-to-SQL
Translate an analytical question
Schema / metrics
Read-only query tool
Writes and Access
Change and protect records
Transactions / retries
Roles / masking

Data Agents

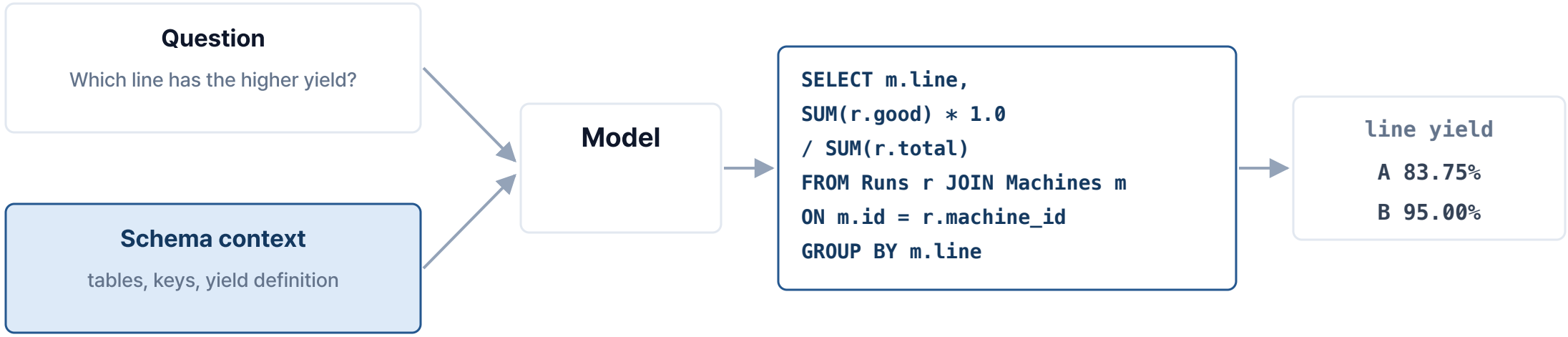
Here, a data agent is an agent that reads and writes data systems



Reads and **writes** go through separate tools

Text-to-SQL

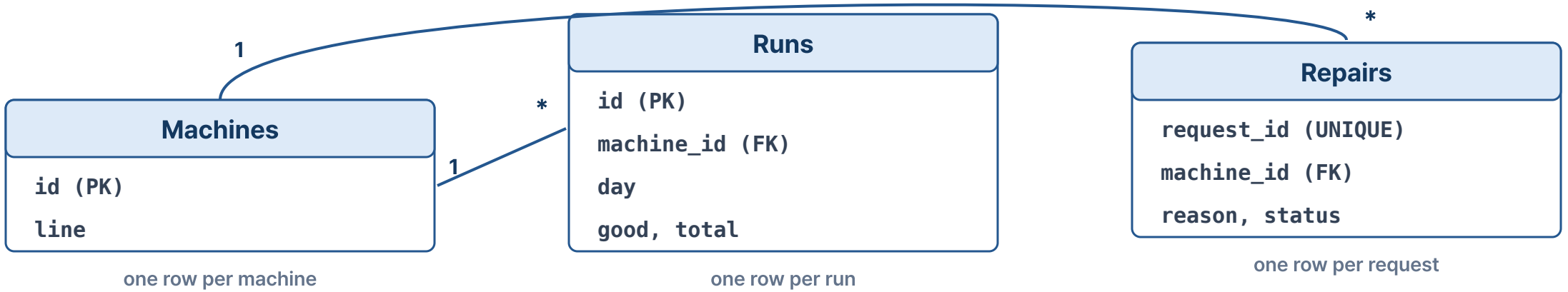
Text-to-SQL translates a natural-language question into a database query



Schema context turns a question into a **checkable** query

Recap: Factory Tables

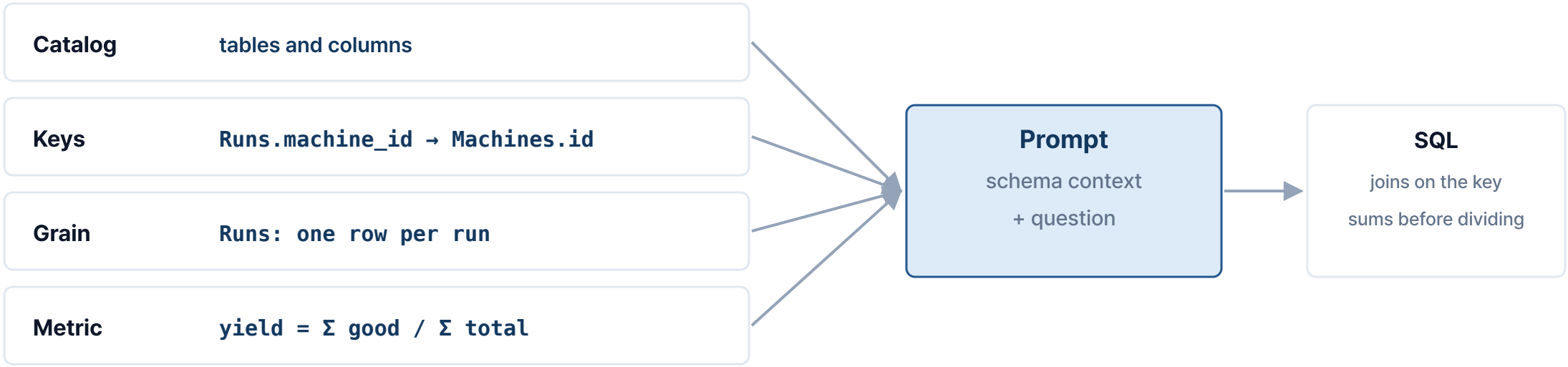
Machine 1 is P-17; each row in Runs records one production run



Keys and grain decide which **joins** are valid

Schema Context

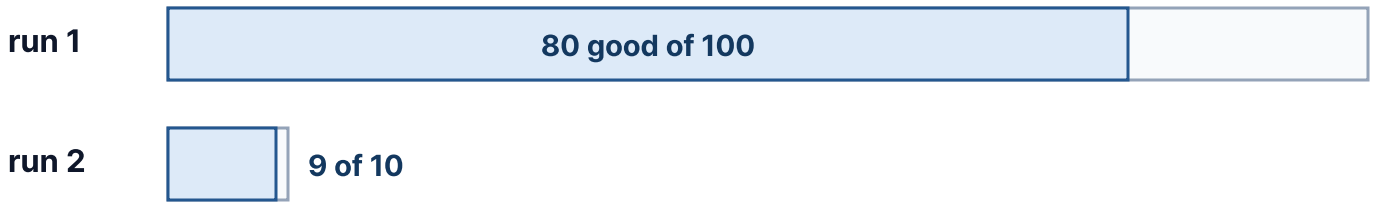
Expose names, types, keys, and grain before generating SQL



The prompt carries keys, grain and the **metric**

Weighted Yield

Aggregate counts before dividing when run sizes differ



Σ good / Σ total = 89 / 110

80.9%: weights runs by size

mean of ratios = (80% + 90%) / 2

85%: the 10-unit run counts as much

Sum the counts, **then** divide

Line Yield Query

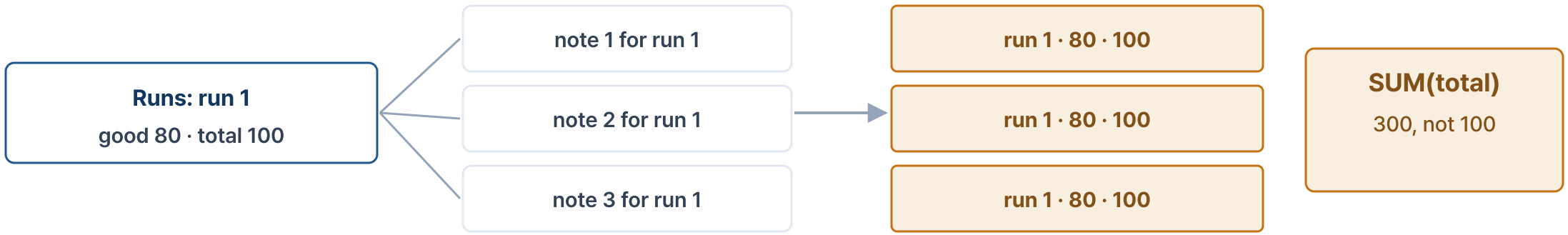
Join by the declared key and divide the aggregated counts

```
SELECT m.line, SUM(r.good) AS good, SUM(r.total) AS
total,
       ROUND(100.0 * SUM(r.good) /
NULLIF(SUM(r.total),0),2) AS yield_pct
FROM Runs r JOIN Machines m ON m.id=r.machine_id
GROUP BY m.line ORDER BY m.line;
```

The result uses production counts rather than an average of run ratios

Caveat: Join Multiplication

An extra one-to-many join can inflate both counts and rankings

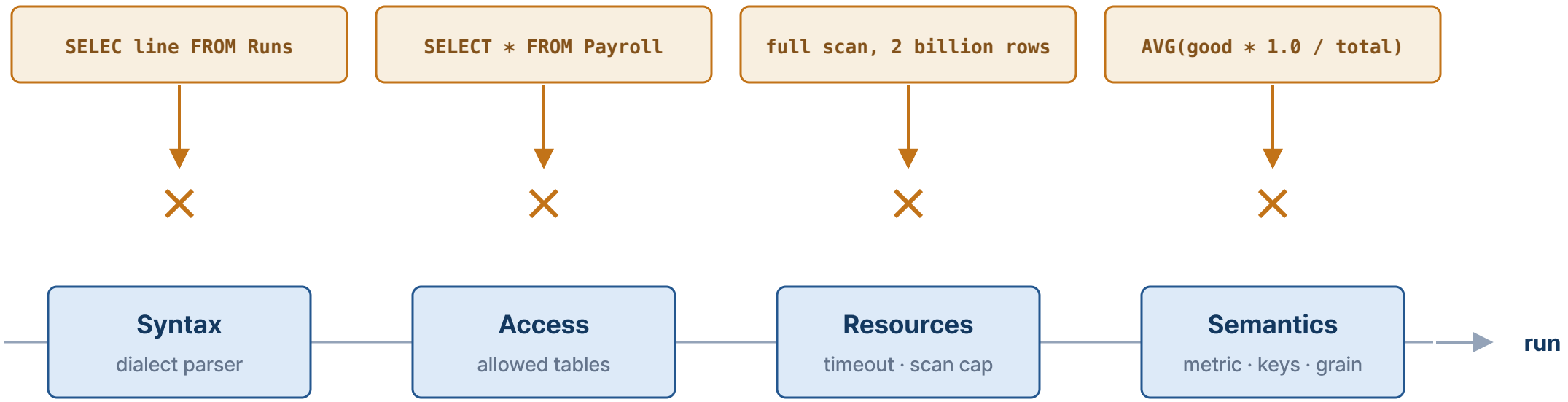


Aggregate notes per run first, then join

Three notes **triple** the production count

Query Validation Layers

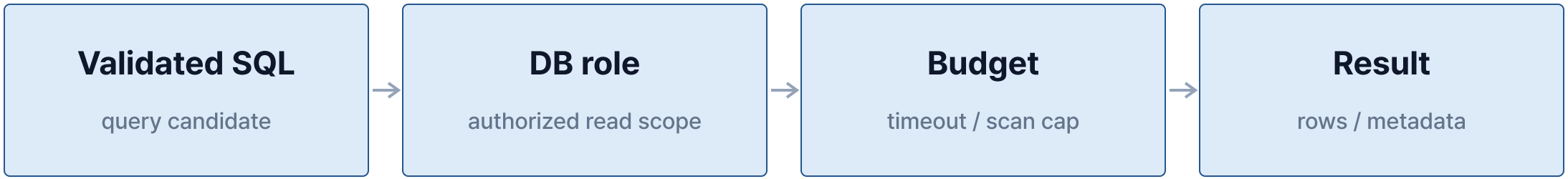
Check syntax, access, resources, and semantics separately



A query that runs can still be **wrong**

Read-Only Query Tool

Execute queries under a constrained database identity



Validated SQL → DB role → Budget → Result

Demo: Factory Yield

Run the calculation in the local SQLite database

1. Return total good and total output for each line
2. Calculate weighted yield as a percentage

```
-- write your query
here, then Run
```

Run ▶ Reset Ctrl/⌘ + Enter

Answer: Factory Yield

Line A yields 83.75%; line B yields 95%

```
SELECT m.line, SUM(r.good) AS good, SUM(r.total) AS
total,

ROUND(100.0*SUM(r.good)/NULLIF(SUM(r.total),0),2) AS
yield_pct
FROM Runs r JOIN Machines m ON m.id=r.machine_id
GROUP BY m.line ORDER BY m.line;
```

The answer follows the stated metric and joins

Text-to-SQL Execution

Pass schema and metric meaning into the model, then execute the query

```
const prompt = ChatPromptTemplate.fromMessages([
 ["system", "Return JSON with sql. Only read Machines and Runs. " + DB_SCHEMA],
 ["human", "{question}"]
]);
const model = makeModel([JSON.stringify({sql:YIELD_SQL})]);
const proposed = await prompt.pipe(model).pipe(new JsonOutputParser())
 .invoke({question:"Yield by line over all runs?"});
const rows = await dbQuery(proposed.sql); // SQLite enforces query-only mode
return {mode:MODEL_MODE, sql:proposed.sql, rows};
```

Demo: Text-to-SQL Execution

Inspect the proposed SQL and the rows returned by SQLite

1. Replace SUM(good)/SUM(total) with AVG(1.0*good/total). Compare line A.
2. Change the proposed query to DELETE FROM Runs. Inspect query-only enforcement.

Execution Offline replay | Installed model na | http://127.0.0.1:11434

```
Type code or click Starter
```

```
Run code to inspect its output.
```

Run | Starter | Reset

Ready

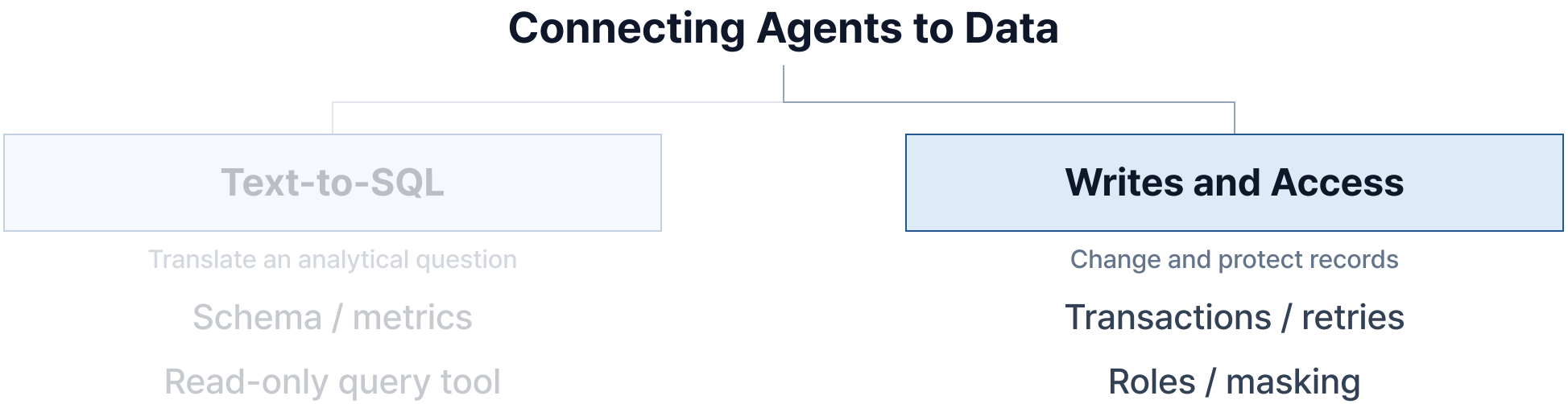
LangChain.js · JavaScript · Ctrl/Cmd+Enter · Offline replay uses fixed responses; local mode calls your model

Answer: Text-to-SQL Execution

The baseline output comes from executing the shown code


```
{"mode": "Recorded response (no inference)", "sql": "SELECT m.line,
ROUND(100.0*SUM(r.good)/SUM(r.total),2) yield_pct FROM Runs r JOIN Machines m ON
m.id=r.machine_id GROUP BY m.line ORDER BY m.line", "rows": [{"line": "A", "yield_pct":
83.75}, {"line": "B", "yield_pct": 95}]}
```

§ Writes and Access



Case study: Repair Request Workflow

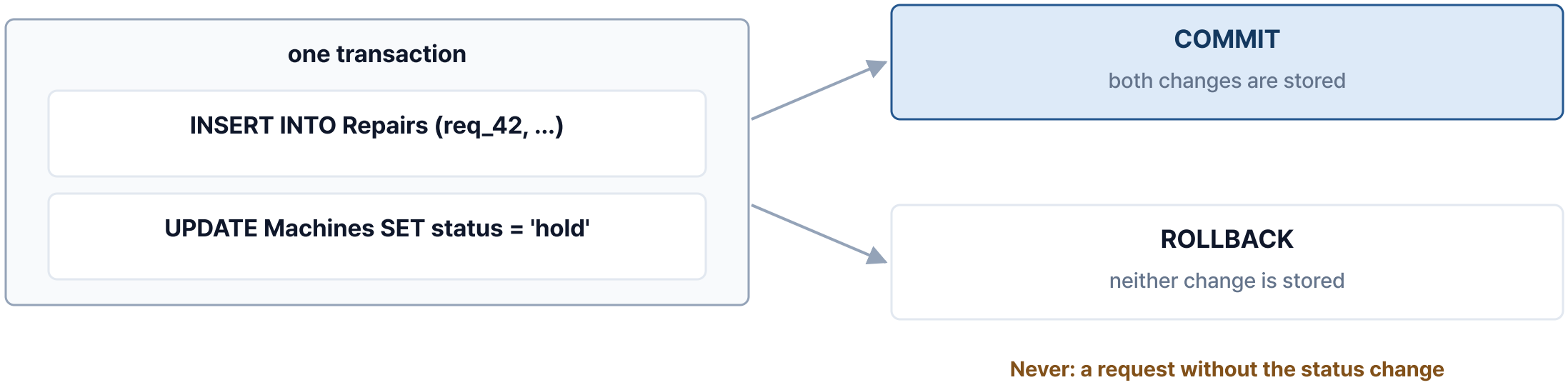
A repair task crosses query, permission, and write boundaries

Stage	Contract	Evidence
Find machine	Authorized lookup	Machine ID + current state
Check readings	Freshness + units	Timestamped measurements
Propose repair	Bounded write + review	Target + reason + request ID
Create request	Transaction + idempotency	Stored row + receipt

The final answer follows verified reads and a confirmed write

Transaction Boundary

A transaction groups changes that must succeed or fail together



Never: a request without the status change

Both writes happen, or **neither**

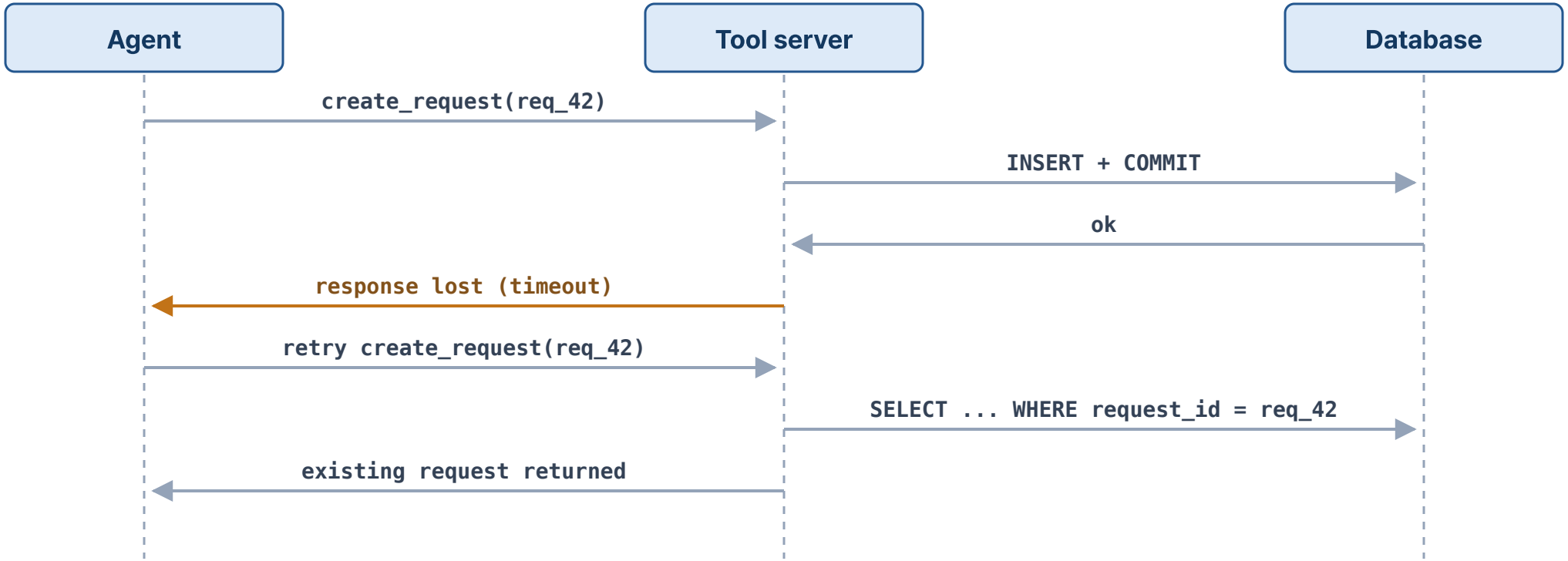
Idempotent Request

A uniqueness constraint can prevent duplicate requests

```
CREATE TABLE Repairs (
  request_id TEXT UNIQUE,
  machine_id INTEGER,
  reason TEXT,
  status TEXT
);
-- The server also verifies that repeated IDs carry the same payload.
```

Caveat: Timeout After Commit

A missing response does not reveal whether the write succeeded



The retry finds the **committed** request

Demo: Duplicate Repair Request

Use SQLite to observe a unique request key

1. Insert req_42 twice in one execution using INSERT OR IGNORE
2. Select Repairs and verify that one row remains

```
-- write your query here, then Run
```

Run ▶ Reset Ctrl/⌘ + Enter

Answer: Duplicate Repair Request

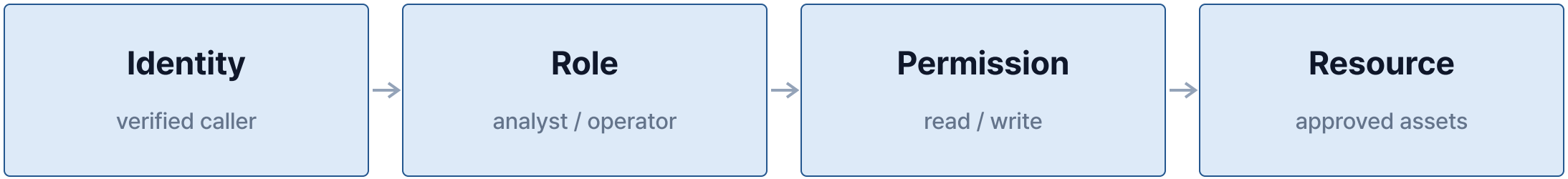
The unique key leaves one record after two identical inserts

```
INSERT OR IGNORE INTO Repairs VALUES
('req_42',1,'vibration','open');
INSERT OR IGNORE INTO Repairs VALUES
('req_42',1,'vibration','open');
SELECT * FROM Repairs;
```

This example demonstrates uniqueness, not the entire write policy

Role-Based Access Control

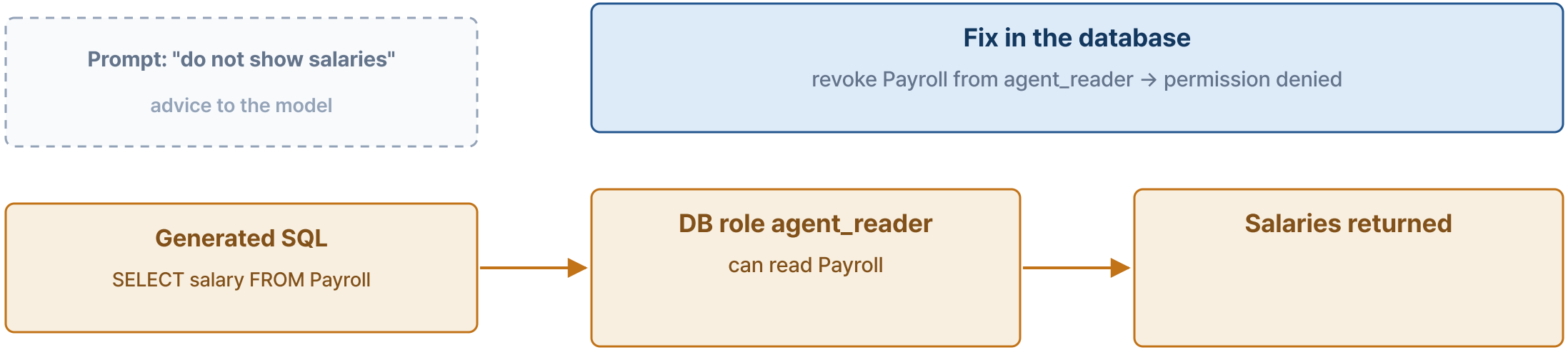
RBAC grants capabilities to roles assigned to authenticated identities



Identity → Role → Permission → Resource

Caveat: Prompt-Based Access

A prompt cannot replace a database permission check



Only the **database permission** stops the query

Row Filters and Column Masks

Filter records and redact sensitive values at query time

Contacts (source)

machine	line	email
1	A	op1@example.test
2	A	op2@example.test
3	B	op3@example.test

Operator, line A

machine	line	email
1	A	op1@example.test
2	A	op2@example.test
filtered out		

Analyst

machine	line	email
1	A	[masked]
2	A	[masked]
3	B	[masked]

Row filter by line; column mask by role; the source row is unchanged

Same table, different **visibility** per caller

Column Masking Example

The same row can expose different values to different roles

machine_id	Role	email visible
1	Operator with approved access	operator1@example.test
1	Analyst	[masked]

Masking changes visibility without changing the source record

Agent Evaluation

Test the outcome, the action path, and the database state

Dimension	Check
Task success	Required result verified by evidence
Tool behavior	Correct tool, arguments, and access scope
Recovery	Timeouts and duplicate requests handled
Efficiency	Calls, tokens, and elapsed time

An agent test observes both the final answer and the changed state

Takeaway: Bound actions, recover state, govern data

1. Validate tool requests and bound the agent loop
2. Keep workflow state outside the model so runs can pause and resume
3. Connect agents to data through read-only queries, transactions, and roles