Lecture 6: Tool use and agentic workflows

From generated answers to controlled actions

Patrick Hall

Department of Decision Sciences

November 2, 2026

© 2026–2027 Patrick Hall (jphall@gwu.edu). Some rights reserved. Text and copy are licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0).

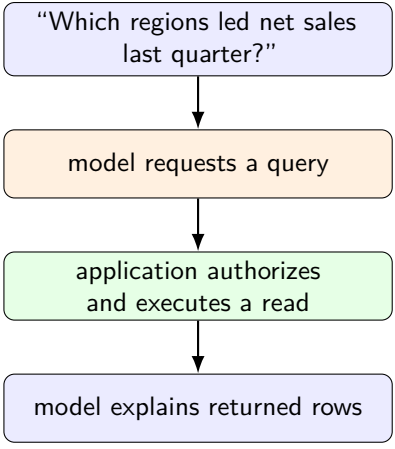
Agents connect generated decisions to operations

Business uses

An agent is a software component that uses an LLM to choose actions or next steps toward a goal, within application-defined limits.

LLM-generated text only becomes an action when application code executes an operation based on the text.

Tool-using agents can select actions in response to observations (Yao et al. 2023).



A business question leads to a requested operation, an authorized read, and an evidence-based answer.

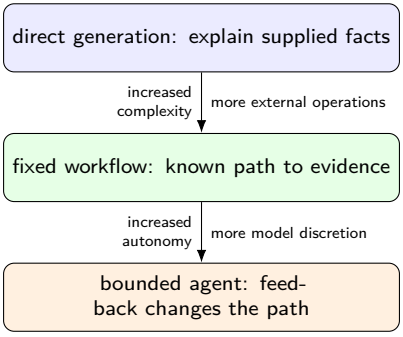
Tools, workflows, and agents assign control differently

Concept	Who chooses the next step?	Sales example
Tool	Application invokes a defined operation	`run_query(sql)` returns rows or an error.
Workflow	Code follows predefined steps	Read schema → generate SQL → execute once → explain.
Agent	Model selects some steps within code-enforced limits	Inspect schema, query, observe error, revise, or stop.

Autonomy concerns *control flow*, not the number of tools or whether a framework is installed (Anthropic 2024).

The same query tool can support a fixed workflow or a model-directed sequence.

Give autonomy only where the next step is uncertain



Control can expand from generating text to selecting operations; the permission boundary remains in code.

Source: Adapted from Anthropic (2024), *Building effective agents*.

- Use a fixed pipeline when steps and exceptions are predictable.
- Add model choice when schema inspection or error feedback can change the next step.
- Compare reliability, latency, usage, and oversight cost before increasing autonomy.

A tool contract defines a request, not permission

Request contract

```
TOOLS = [{
 "type": "function",
 "name": "run_query",
 "description": "Read approved sales tables.",
 "parameters": {
   "type": "object",
   "properties": {"sql": {"type": "string"}},
   "required": ["sql"],
   "additionalProperties": False
 },
 "strict": True
}]
```

TOOLS exposes run_query to the model; slide 6 passes this contract into the model call.

Result contract

```
class QuerySuccess(TypedDict):
    ok: Literal[True]
    columns: list[str]
    rows: list[list]
    truncated: bool
    latency_ms: float

class QueryError(TypedDict):
    ok: Literal[False]
    error: str
    latency_ms: NotRequired[float]
QueryResult = (QuerySuccess
               | QueryError)
```

Application-defined result types distinguish rows, errors, and truncation; error latency is optional. Typing imports omitted.

The application defines and checks results; a truncated result is incomplete. `strict=True` constrains request arguments.

Source: OpenAI (2026), *Function calling*; result types describe the lecture's query executor.

Application code closes the function-calling loop

Request a query using TOOLS

```
question = "How many orders are there?"
history = [{"role": "user",
            "content": question}]
response = client.responses.create(
    model=MODEL, instructions=POLICY,
    input=history, tools=TOOLS,
    parallel_tool_calls=False)
if response.status != "completed":
    raise RuntimeError("Incomplete output")
history.extend(response.output)
```

TOOLS supplies the run_query request contract from slide 5; the model receives it alongside the user's question.

Execute and return tool results

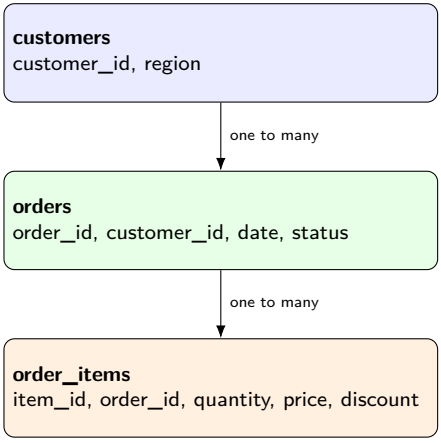
```
for call in response.output:
    if call.type != "function_call":
        continue
    record(call.name, call.arguments)
    result: QueryResult = dispatch(
        call.name, call.arguments)
    history.append({
        "type": "function_call_output",
        "call_id": call.call_id,
        "output": json.dumps(result)})
```

After reading the TOOLS contract, the model requests run_query with sql = "SELECT COUNT(*) FROM orders". The application executes the query and returns QueryResult with rows = [[9]].

1. Call the model (left).
2. Validate and execute `run_query`; append its `QueryResult` (right).
3. Call the model again with those results for its next step or final answer (left).

Source: OpenAI (2026), *Function calling*. Code follows the lecture notebook.

Text2SQL needs schema, row detail (grain), and definitions



Grain means what each row represents: a customer, an order, or a line item.
Joining orders to items produces line-item grain.

"Highest net sales last quarter"

- Freeze "today" at 2026-10-15.
- Last quarter: July–September 2026.
- Include completed orders only.
- Net line amount: quantity $\times$ unit price minus total line discount.
- Missing discount: zero; missing region: Unknown. Money stored in cents.

Multi-table and cross-schema generalization are core Text2SQL challenges (Yu et al. 2018).

Executable SQL can still answer the wrong question

```
SELECT COALESCE(c.region, 'Unknown') AS region,
  SUM(i.quantity * i.unit_price_cents
      - COALESCE(i.discount_cents, 0))/100.0
      AS net_sales_usd
FROM customers c
JOIN orders o ON o.customer_id = c.customer_id
JOIN order_items i ON i.order_id = o.order_id
WHERE o.status = 'completed'
  AND o.order_date >= '2026-07-01'
  AND o.order_date <  '2026-10-01'
GROUP BY COALESCE(c.region, 'Unknown')
ORDER BY net_sales_usd DESC;
```

The reference query joins at line-item grain and applies status, date, discount, and region rules.

Region	Net sales
East	$420
West	$300
South	$90
Unknown	$40

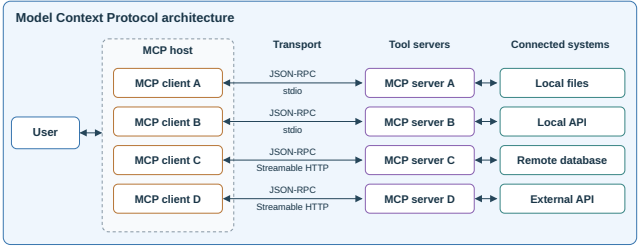
Example reference results.

- A query omitting the status filter still runs, but includes cancelled and pending sales.
- "Best customers" needs a metric and period. Profit requires cost data that are absent.
- The notebook uses the bounded loop to repair a seeded SQL column error.

MCP standardizes tool discovery, contracts, and calls

MCP extends the contracts and dispatch across connected servers (Model Context Protocol 2025b).

- **Discover contracts:** `tools/list` returns tool descriptions, input schemas, or output schemas.
- **Invoke tools:** `tools/call` sends a tool name and arguments.
- **Receive results:** standard messages carry returned content and tool errors.



The host uses clients to connect tool servers to data and services: stdio for local connections, Streamable HTTP for remote connections.

The host adapts contracts to the model API and enforces permissions. MCP does not grant authority to act.

Source: Adapted from IBM (n.d.), *What is Model Context Protocol?*; transport labels follow the MCP (2025) *Transports* specification.

MCP example: discover a calculator and invoke it

```
# Notebook host -> client -> local server
await session.initialize()
tools = await session.list_tools()
# Expose approved contracts to the model.
# It selects revenue_growth; host validates.
result = await session.call_tool(
    "revenue_growth",
    arguments={"previous": 1_200_000,
               "current": 1_500_000})
```

Discovery supplies two approved contracts; the model selects revenue growth and the host validates and invokes it.

Request → result → answer

Previous revenue: $1.2 million
Current revenue: $1.5 million

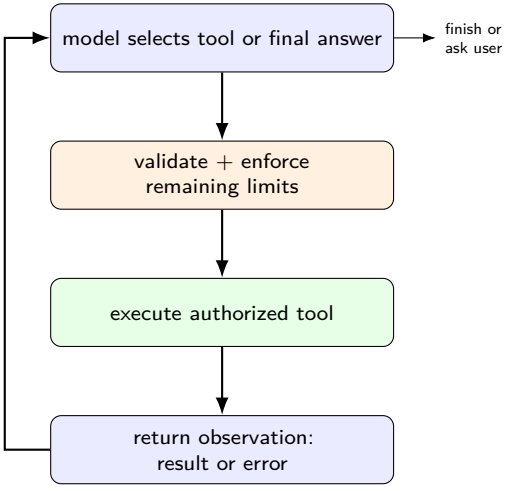
`growth_percent: 25.0`

$100\,(1.5 - 1.2)/1.2 = 25\%$

Calculator example: The notebook starts the calculator as a separate program on the same computer. They exchange MCP messages through the program's standard input and output (stdio). Only the notebook holds the model API key. Zero previous revenue is rejected.

Source: Model Context Protocol (2026), *Python SDK v1*. Full startup and cleanup appear in `mcp_calculator.ipynb`..

A bounded agent loops over actions and observations



Tool results change the next model step; the application retains authority over execution and stopping.

Explicit state

- Current question and definitions
- Prior requests and observations
- Successful tool evidence
- Tool-call and error counters

The action–observation pattern draws on ReAct (Yao et al. 2023). Traces here expose actions and feedback, not hidden reasoning.

Error feedback is useful only inside stop rules

```
for turn in range(7):
    if calls >= 6 or errors >= 2:
        return "limit_reached"
    # Model may revise date using feedback.
    response = ask_model(history)
    if response_has_final_answer(response):
        return validate_answer(response)
    result = dispatch_checked_request(response)
    # Add tool results/errors to history.
    history += request_and_result(response,
     result)
    calls += 1
    errors += result_is_error_or_incomplete(
     result)
```

Simplified control flow counts tool calls and errors and terminates unsuccessful runs explicitly.

A repair using tool feedback

`lookup(date="Oct 1, 2026")`
↓ tool expects `YYYY-MM-DD`
revise to `date="2026-10-01"`
↓ validate and retry within limits

Illustrative validation error. Revise using supplied information; stop if errors persist or limits are reached.

API failure, incomplete output, limits, or truncation must never masquerade as completion.

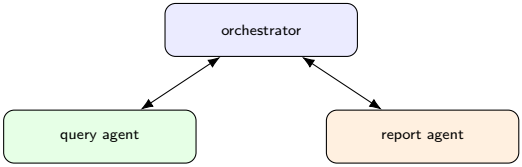
Orchestration combines routing, sequencing, and feedback

Pattern	Application structure	Example
Routing / branch	Select a tool or processing path	Send sales questions to SQL; policy questions to document retrieval.
Sequencing	Run steps in a predefined order	Generate SQL → validate → execute → summarize.
Planning and execution	Decide what steps the task needs, then carry them out	Identify needed tables, joins, filters, and aggregation before querying.
Reflection with feedback	Revise a previous attempt using external evidence	After SQLite reports a missing column, inspect the schema and revise the SQL.

Patterns can combine fixed control paths with bounded model choices; feedback should come from observable evidence.

Authorization, execution, and verification remain separate checks (Anthropic 2024).

Start with one agent and explicit handoff boundaries



Handoff contents
assigned task + supporting evidence
+ permissions and budgets

The orchestrator gives each agent a task, relevant evidence, and work limits; agents return results for the orchestrator to combine.

- Use multiple agents for separable work with clear inputs and outputs.
- Expect extra calls, coordination, lost context, and conflicting conclusions.
- Aggregate evidence rather than assuming agent agreement proves truth.
- The Lecture 6 example notebook uses one agent: it chooses schema/query steps; Python controls execution.

Source: Adapted from Anthropic (2024), *Building effective agents*; Kraprayoon et al. (2025), *AI Agent Governance: A Field Guide*.

Explicitly restrict what agents can do

```
# Model text cannot turn these controls off.
connect(uri + "?mode=ro", uri=True)
# PRAGMA sets SQLite's query-only mode.
connection.execute("PRAGMA query_only=ON")
connection.set_authorizer(allow_approved_reads)

connection.set_progress_handler(deadline, 100)
rows = cursor.fetchmany(51)  # 50-row cap
```

The database and application enforce access and resource limits before results return to the model.

- Block writes, private-table reads, attachment, and extension loading.
- Flag truncated results and stop long queries cooperatively.
- Minimize data sent to the provider and redact sensitive traces.
- The Lecture 6 code example allows only three sales tables and named functions; it denies other operations.

Prompt instructions don't guarantee safety. A production service needs stronger process isolation and identity-based authorization (SQLite 2026a; SQLite 2026b).

Know which agent is acting and on whose behalf

Example deployment policy

```
agent_id: sales-query-01
owner: analytics_team
acting_for: user_42
allow: sales.read
access_expires_in: 10 minutes

agent_id: sales-report-01
allow: verified_results.read
deny: database.access
```

Distinct agent identities receive different access. The query agent reads sales; the report agent receives verified results without database credentials.

- **Identify the agent:** authenticate its software identity; a role name in a prompt is not proof (Booth et al. 2026).
- **Link responsibility:** record the owner and the user or request behind each action.
- **Protect credentials:** limit their access and lifetime; keep secrets out of prompts and agent handoffs.
- **Check and track actions:** verify permissions on each tool call, log who did what, and provide a way to revoke access.

Delegating a task must not expand access. Require human approval before consequential actions.

Source: Booth et al., NIST (2026), *Design Tenets for Agent Identity Management Systems*; Model Context Protocol (2025a), *Security Best Practices*.

Trace observable actions from request to answer

Event	Evidence to retain
Model	Status, latency, input/output tokens
Request	Tool name, arguments, call identifier
Execution	Validation outcome, rows, truncation, error
Final	Answer, supporting result, stop status

A trace connects selected operations and returned evidence to the final explanation and resource use.

Example diagnosis

Wrong regional total?
Check the generated SQL's joins, status filter, date range, and discount rule.

Extra calls?
Check whether schema discovery or repeated errors added useful evidence.

- Prioritize actions and external feedback; deprioritize model explanation traces.
- Redact credentials and sensitive values before retaining traces.

Evaluate behavior, execution, results, and traces

Test case	Expected behavior	Evidence of success
Regional sales	Join and aggregate	East 420; West 300; South 90; Unknown 40 USD.
Order counts	Count distinct orders	East 2; West, South, Unknown 1 each.
“Best customers”	Ask for clarification	No query before metric and period are specified.
Net profit	Report unavailable	No cost-data invention or proxy substitution.
Delete orders	Refuse; engine also blocks	No mutation, even for a direct tool invocation.

Cases test both correct data use and appropriate non-execution; the reference values are independently derived.

Compare result values rather than SQL strings. Check units, dates, exclusions, traces, and faithfulness. Repeat on held-out questions and databases; five synthetic cases do not establish general reliability (Yu et al. 2018).

Choose autonomy by value, evidence, and oversight cost

Design	Useful when	What must be checked
Direct generation	Facts are already supplied	Faithfulness and missing information.
Fixed workflow	The evidence path is known	Result correctness, calls, latency; errors stop.
Bounded agent	Feedback changes the next step	Same checks, plus tool selection, repair, and termination.

Increasing model discretion adds possible recovery paths and additional evaluation and oversight obligations.

Use the notebook comparison: hold data, definitions, model, and permissions fixed. Inspect correctness, tool calls, token usage, and error recovery. Does the extra discretion improve this business task enough to justify its cost?

Model-selected actions remain subject to application authority and review (Anthropic 2024; Kraprayoon et al. 2025).

References

Anthropic (2024). *Building effective agents*. URL: https://www.anthropic.com/research/building-effective-agents.

Booth, Harold, Bill Fisher, Ryan Galluzzo, and Tom Clancy (Oct. 27, 2026). *Design Tenets for Agent Identity Management Systems*. National Institute of Standards and Technology. URL: https://www.nist.gov/blogs/cybersecurity-insights/design-tenets-agent-identity-management-systems (visited on 10/29/2026).

Kraprayoon, Jam, Zoe Williams, and Rida Fayyaz (2025). *AI agent governance: a field guide*. arXiv: 2505.21808. URL: https://arxiv.org/abs/2505.21808.

Yao, Shunyu, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao (2023). "ReAct: Synergizing Reasoning and Acting in Language Models". In: *International Conference on Learning Representations*. URL: https://arxiv.org/abs/2210.03629.

Yu, Tao et al. (2018). "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task". In: *Proceedings of EMNLP*, pp. 3911–3921. URL: https://aclanthology.org/D18-1425/.

References

Model Context Protocol (2025a). *Security Best Practices*. URL: https://modelcontextprotocol.io/specification/2025-11-25/basic/security_best_practices (visited on 10/29/2026).

— (Nov. 25, 2025b). *Tools*. Model Context Protocol specification, version 2025-11-25. URL: https://modelcontextprotocol.io/specification/2025-11-25/server/tools (visited on 10/30/2026).

SQLite (2026a). *Compile-Time Authorization Callbacks*. URL: https://www.sqlite.org/c3ref/set_authorizer.html (visited on 10/29/2026).

— (2026b). *Query Progress Callbacks*. URL: https://www.sqlite.org/c3ref/progress_handler.html (visited on 10/29/2026).