

AI Agents & Tool Use

Building LLM systems that take actions, use tools, and complete multi-step tasks autonomously.

Last reviewed 28 Sep 2026 ·

Prerequisites: [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) · [Prompt Engineering](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/prompt-engineering/)

Related: [LangChain & LlamaIndex](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/langchain-llamaindex/) · [Claude Code](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/claude-code/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/claude-code/) · [MCP & Text-to-SQL](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mcp-text-to-sql/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mcp-text-to-sql/) · [Evals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/eval-and-evals/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Overview

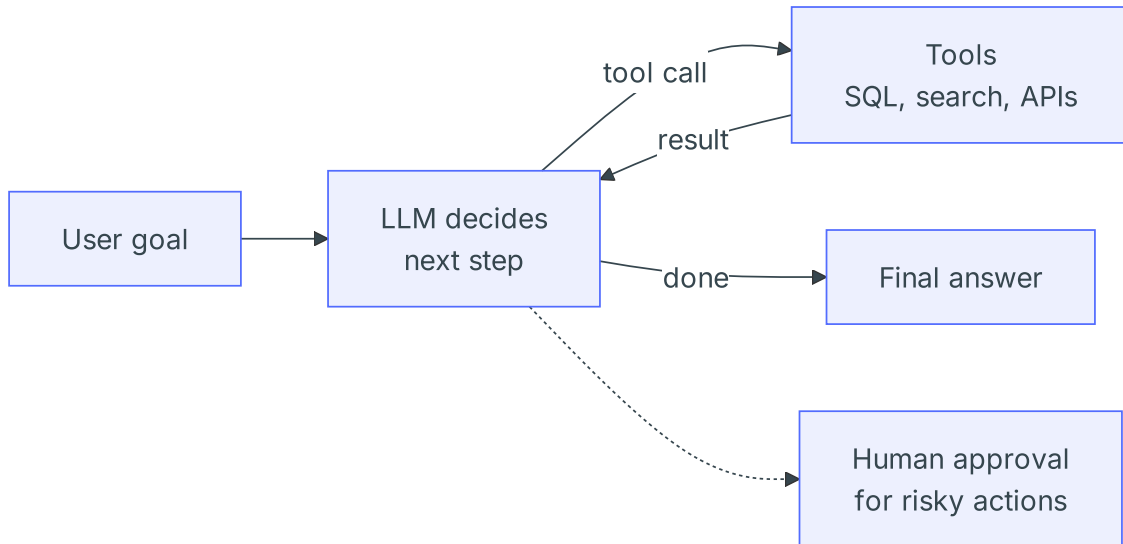
Challenge: A single LLM call produces a single response. It cannot look anything up, verify that generated SQL runs, or decide on a next step based on what it finds. Investigating a question such as "why is today's orders report wrong?" requires several dependent steps: checking freshness, reviewing the latest pipeline run, querying the table, and comparing it with the source.

Solution: an agent gives the model *tools* — functions it can request, such as `run_sql`, `get_pipeline_status`, or `search_docs` — and runs in a loop: the model chooses an action, the application executes it, the model reads the result, and the cycle repeats until the task is complete.

```
goal → model: "check freshness first" → run tool:
get_table_freshness("fct_orders")
      ↑                               |
      └── result: "last load 26h ago" ←─┘
      model: "check the DAG" →
get_dag_runs("orders_daily") → "failed at
```

```
load_to_snowflake"
    model: "Answer: yesterday's load failed at
load_to_snowflake; data is 26h stale."
```

Design principle: use the *least* autonomy that solves the problem. A fixed sequence of LLM calls (a workflow) is cheaper, faster, and easier to test than an agent. Agents are appropriate when the steps cannot be known in advance, and they require guardrails: limited tools, budgets, approvals, and logging.



On this page

Basic - [What Is an AI Agent](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#what-is-an-ai-agent) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#what-is-an-ai-agent) - [Tool Use Fundamentals](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#tool-use-fundamentals) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#tool-use-fundamentals) - [Single-Step Tool Use](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#single-step-tool-use) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#single-step-tool-use)

Intermediate - [Agentic Loop](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#agentic-loop) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#agentic-loop) - [Building a DE Agent](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#building-a-de-agent) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#building-a-de-agent) - [ReAct Pattern](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#react-pattern) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#react-pattern) - [Error Handling in Agents](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#error-handling-in-agents) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#error-handling-in-agents)

Advanced - [Multi-Agent Systems](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#multi-agent-systems) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#multi-agent-systems) - [Agent Memory](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#agent-memory) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#agent-memory) - [Human-in-the-Loop](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#human-in-the-loop) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#human-in-the-loop) - [Production Agent Patterns](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#production-agent-patterns) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#production-agent-patterns)

Reference - Common Pitfalls (<https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#common-pitfalls>) - **Cheat Sheet** (<https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#cheat-sheet>) - **Interview Questions** (<https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#interview-questions>) - **Further Reading** (<https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/ai-agents/#further-reading>)

What Is an AI Agent

An agent is an LLM that can take actions — it calls tools, observes results, and decides what to do next, repeating until it completes the task.

Chatbot:

User asks → LLM responds (one step)

Agent:

User asks → LLM thinks → calls tool → observes result

→ LLM thinks → calls tool → observes result

→ LLM thinks → answers (many steps)

Key components:

Component	Role
LLM	The brain — decides what action to take next
Tools	Functions the LLM can call (search, SQL, API calls, file ops)
Memory	Context from previous steps and conversations
Planner	(optional) Breaks big tasks into subtasks
Executor	Runs the tool calls and returns results

Tool Use Fundamentals

The LLM doesn't directly execute code — it generates a structured "call this function with these arguments" request. Your code executes it and returns the result.

LLM output: {"tool": "run_sql", "args": {"query": "SELECT COUNT(*) FROM orders"}}

```
Your code:    runs the query → returns {"count": 15234}
LLM input:    "The query returned 15234 rows"
LLM output:   "There are 15,234 orders in the table."
```

Tool definition (Anthropic):

```
tool = {
    "name": "run_sql",
    "description": "Execute a SQL query and return results.
    Use for counting, aggregating, or looking up data.",
    "input_schema": {
        "type": "object",
        "properties": {
            "query": {
                "type": "string",
                "description": "SQL query to execute. Must
                be a SELECT statement."
            },
            "limit": {
                "type": "integer",
                "description": "Max rows to return (default
                100)",
                "default": 100
            }
        },
        "required": ["query"]
    }
}
```

What makes a good tool: 1. **Clear description** — the LLM uses this to decide when to call it
2. **Narrow scope** — one tool does one thing 3. **Safe by default** — read-only where possible; destructive actions need confirmation 4. **Handles errors gracefully** — return error as a string, not an exception

Single-Step Tool Use

```
import anthropic
import json

client = anthropic.Anthropic()
```

```

tools = [
    {
        "name": "get_table_row_count",
        "description": "Get the number of rows in a
database table",
        "input_schema": {
            "type": "object",
            "properties": {
                "table_name": {"type": "string",
"description": "Name of the table"}
            },
            "required": ["table_name"]
        }
    }
]

def get_table_row_count(table_name: str) -> dict:
    # Simulated – in production, this runs a real query
    counts = {"orders": 152340, "customers": 48921,
"products": 5420}
    if table_name in counts:
        return {"table": table_name, "row_count":
counts[table_name]}
    return {"error": f"Table '{table_name}' not found"}

# First turn: LLM decides to use the tool
response = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=1024,
    tools=tools,
    messages=[{"role": "user", "content": "How many orders
do we have?"}]
)

# Extract tool call
for block in response.content:
    if block.type == "tool_use":
        print(f"Tool: {block.name}, Args: {block.input}")
        result = get_table_row_count(**block.input)
        print(f"Result: {result}")

```

```

# Second turn: give result back to LLM
final = client.messages.create(
    model="claude-sonnet-5-5",
    max_tokens=512,
    tools=tools,
    messages=[
        {"role": "user", "content": "How many
orders do we have?"},
        {"role": "assistant", "content":
response.content},
        {"role": "user", "content": [{"type":
"tool_result",

"tool_use_id": block.id,

"content": json.dumps(result)}}]}
    ]
)
print(next(b.text for b in final.content if b.type
== "text"))
# "We currently have 152,340 orders in the
database."

```

Agentic Loop

A loop that continues until the model stops calling tools.

```

import anthropic
import json

client = anthropic.Anthropic()

def run_agent(user_message: str, tools: list,
tool_executor: dict,
               system: str = "", max_iterations: int = 10) -
> str:
    """
    Generic agentic loop.
    tool_executor: {"tool_name": callable}
    """

```

```

messages = [{"role": "user", "content": user_message}]

for i in range(max_iterations):
    response = client.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=16000,
        tools=tools,
        messages=messages,
        **({"system": system} if system else {}),  #
omit an empty system prompt
    )

    # Agent is done
    if response.stop_reason == "end_turn":
        return next((b.text for b in response.content
if b.type == "text"), "")

    # Server paused a long turn – send the conversation
back to let it continue
    if response.stop_reason == "pause_turn":
        messages.append({"role": "assistant",
"content": response.content})
        continue

    # Anything else (max_tokens, refusal, ...) – stop
instead of re-sending the same request
    if response.stop_reason != "tool_use":
        raise RuntimeError(f"Agent stopped early:
stop_reason={response.stop_reason}")

    # Agent wants to use tools
    if response.stop_reason == "tool_use":
        messages.append({"role": "assistant",
"content": response.content})
        tool_results = []

        for block in response.content:
            if block.type != "tool_use":
                continue

            print(f"[Tool call] {block.name}
({json.dumps(block.input)})")

```

```

        is_error = False
        if block.name in tool_executor:
            try:
                result = tool_executor[block.name]
            (**block.input)
                result_str = json.dumps(result) if
isinstance(result, dict) else str(result)
                print(f"[Tool result]
{result_str[:200]}")
            except Exception as e:
                result_str, is_error = f"ERROR:
{e}", True
                print(f"[Tool error] {result_str}")
        else:
            result_str, is_error = f"ERROR: Tool
'{block.name}' not available", True

        tool_results.append({
            "type": "tool_result",
            "tool_use_id": block.id,
            "content": result_str,
            "is_error": is_error,      # lets
the model know the call failed
        })

        messages.append({"role": "user", "content":
tool_results})

    return "Max iterations reached without completing the
task."

```

Building a DE Agent

A practical data engineering agent that can query schemas, run SQL, and check pipeline status.

```

import anthropic
import json
from datetime import datetime

```



```

client = anthropic.Anthropic()

# — Tool definitions


---


—
DE_TOOLS = [
    {
        "name": "list_tables",
        "description": "List all available tables in the
data warehouse",
        "input_schema": {
            "type": "object",
            "properties": {
                "schema": {"type": "string", "description":
"Schema/database name (optional)"}
            }
        },
    },
    {
        "name": "get_table_schema",
        "description": "Get column names and types for a
specific table",
        "input_schema": {
            "type": "object",
            "properties": {
                "table_name": {"type": "string"}
            },
            "required": ["table_name"]
        },
    },
    {
        "name": "run_sql",
        "description": "Execute a read-only SQL query. Use
for counts, aggregations, sampling data.",
        "input_schema": {
            "type": "object",
            "properties": {
                "query": {"type": "string", "description":
"Must be a SELECT statement"},
                "limit": {"type": "integer", "default": 20}
            },
        },
    },

```

```

        "required": ["query"]
    },
    {
        "name": "get_pipeline_status",
        "description": "Get the recent run status of a data
pipeline",
        "input_schema": {
            "type": "object",
            "properties": {
                "pipeline_name": {"type": "string"}
            },
            "required": ["pipeline_name"]
        }
    },
    {
        "name": "check_data_freshness",
        "description": "Check how recent the data is in a
table",
        "input_schema": {
            "type": "object",
            "properties": {
                "table_name": {"type": "string"},
                "timestamp_column": {"type": "string"},
                "default": "updated_at"
            },
            "required": ["table_name"]
        }
    }
]

```

— Tool implementations (simulated)

```

def list_tables(schema=None):
    return {"tables": ["orders", "customers", "products",
"dim_date", "fct_revenue"]}

def get_table_schema(table_name):
    schemas = {
        "orders": [
            {"column": "order_id", "type": "VARCHAR",
"nullable": False},

```

```

        {"column": "customer_id", "type": "INTEGER",
"nullable": False},
        {"column": "amount", "type":
"DECIMAL(10,2)", "nullable": False},
        {"column": "status", "type": "VARCHAR",
"nullable": False},
        {"column": "created_at", "type": "TIMESTAMP",
"nullable": False},
    ]
}

return schemas.get(table_name, {"error": f"Table
{table_name} not found"})

def run_sql(query, limit=20):
    # Safety check – only allow SELECT
    if not query.strip().upper().startswith("SELECT"):
        return {"error": "Only SELECT statements are
allowed"}
    # Simulated results
    return {"rows": [{"count": 152340}], "columns":
["count"], "elapsed_ms": 87}

def get_pipeline_status(pipeline_name):
    return {
        "pipeline": pipeline_name,
        "last_run": "2024-03-15 02:15:43",
        "status": "success",
        "duration_minutes": 23,
        "rows_processed": 15234
    }

def check_data_freshness(table_name,
timestamp_column="updated_at"):
    return {
        "table": table_name,
        "latest_record": "2024-03-15 02:08:31",
        "hours_old": 0.5,
        "sla_hours": 2,
        "status": "FRESH"
    }

TOOL_MAP = {

```

```

    "list_tables":          list_tables,
    "get_table_schema":     get_table_schema,
    "run_sql":              run_sql,
    "get_pipeline_status":  get_pipeline_status,
    "check_data_freshness": check_data_freshness,
}

SYSTEM = """You are a data engineering assistant with
access to the data warehouse.
Use the provided tools to answer questions accurately.
Always check the actual data rather than guessing.
Format numbers with commas. Be concise."""

# — Run the agent

```

```

answer = run_agent(
    user_message="Is the orders data up to date? And how
many orders came in today?",
    tools=DE_TOOLS,
    tool_executor=TOOL_MAP,
    system=SYSTEM
)
print(answer)

```

ReAct Pattern

Reason + Act — the model alternates between thinking and acting. Makes reasoning transparent and easier to debug.

```

# ReAct via chain-of-thought in the system prompt
REACT_SYSTEM = """You are an analytical assistant. Think
through problems step by step.

```

```

When solving a task, use this format:
Thought: What do I know? What do I need to find out?
Action: Which tool to call and why
Observation: What the tool returned
... (repeat until you have enough information)
Thought: I now have everything I need

```

Answer: Final answer to the user

Always show your reasoning before calling tools."""

```
# The model will naturally produce:
# Thought: The user wants to know if orders data is fresh.
# I should check the freshness.
# Action: check_data_freshness(table_name="orders")
# Observation: {"hours_old": 0.5, "status": "FRESH"}
# Thought: Data is fresh. Now I need today's order count.
# Action: run_sql(query="SELECT COUNT(*) FROM orders WHERE
# DATE(created_at) = CURRENT_DATE")
# Observation: {"rows": [{"count": 8423}]}
# Answer: The orders data is fresh (last updated 30 minutes
# ago). Today we've received 8,423 orders.
```

Error Handling in Agents

```
def safe_tool_call(tool_fn, **kwargs) -> str:
    """Wrap every tool call – return errors as strings so
    the agent can recover."""
    try:
        result = tool_fn(**kwargs)
        return json.dumps(result)
    except PermissionError as e:
        return json.dumps({"error": "permission_denied",
        "detail": str(e)})
    except TimeoutError:
        return json.dumps({"error": "timeout", "detail":
        "Query timed out after 30s. Try a more specific query."})
    except Exception as e:
        return json.dumps({"error": "tool_failed",
        "detail": str(e)})

# Agents can recover from errors – give them the error
# message as context
# Example:
# Tool result: {"error": "timeout", "detail": "Query timed
# out after 30s"}
# Agent: "I see the query timed out. Let me try with a date
```

```
filter to reduce the data scanned."
# Next action: run_sql(query="SELECT COUNT(*) FROM orders
WHERE created_at >= CURRENT_DATE")
```

Multi-Agent Systems

Multiple specialized agents working together.

```
# Orchestrator → Subagents pattern
# Orchestrator decides which specialist to call;
specialists do the work

ORCHESTRATOR_TOOLS = [
    {
        "name": "call_data_agent",
        "description": "Call the data agent to query the
warehouse or check schemas",
        "input_schema": {
            "type": "object",
            "properties": {"task": {"type": "string"}},
            "required": ["task"]
        }
    },
    {
        "name": "call_pipeline_agent",
        "description": "Call the pipeline agent to check
DAG status or pipeline health",
        "input_schema": {
            "type": "object",
            "properties": {"task": {"type": "string"}},
            "required": ["task"]
        }
    }
]

def call_data_agent(task: str) -> str:
    return run_agent(task, tools=DE_TOOLS,
tool_executor=TOOL_MAP,
                    system="You are a data warehouse
expert.")
```

```

def call_pipeline_agent(task: str) -> str:
    # Different system prompt, different tools focused on
    pipelines
    return run_agent(task, tools=DE_TOOLS,
                      tool_executor=TOOL_MAP,
                      system="You are a pipeline reliability
engineer.")

orchestrator_result = run_agent(
    user_message="Give me a morning report: data freshness
for all key tables and pipeline status.",
    tools=ORCHESTRATOR_TOOLS,
    tool_executor={
        "call_data_agent": call_data_agent,
        "call_pipeline_agent": call_pipeline_agent,
    },
    system="You are an orchestrator. Delegate tasks to
specialist agents."
)

```

Agent Memory

Agents are stateless by default — each call starts fresh. Add memory explicitly.

```

class AgentWithMemory:
    def __init__(self, tools, tool_executor, system,
max_history=20):
        self.tools = tools
        self.tool_executor = tool_executor
        self.system = system
        self.max_history = max_history
        self.history = []

    def _trim(self):
        # Rolling window — but the conversation must always
        start with a user turn
        while len(self.history) > self.max_history or
(self.history and self.history[0]["role"] != "user"):
            self.history.pop(0)

```

```

def chat(self, user_message: str) -> str:
    self.history.append({"role": "user", "content":
user_message})
    self._trim()
    messages = list(self.history)

    # Run agent with full history
    response = client.messages.create(
        model="claude-sonnet-5-5",
        max_tokens=2048,
        system=self.system,
        tools=self.tools,
        messages=messages
    )

    # Handle tool use
    if response.stop_reason == "tool_use":
        # ... (same agentic loop logic)
        pass

    # Save response to history
    answer = next(b.text for b in response.content if
b.type == "text")
    self.history.append({"role": "assistant",
"content": answer})
    return answer

agent = AgentWithMemory(tools=DE_TOOLS,
tool_executor=TOOL_MAP,
                        system=SYSTEM)

print(agent.chat("How many orders do we have?"))
print(agent.chat("And how many of those are from today?"))
# remembers previous context
print(agent.chat("What's the average order value?"))

```

Human-in-the-Loop

For destructive or high-stakes actions, require human approval before execution.

```

DESTRUCTIVE_TOOLS = {"delete_table", "truncate_table",
"run_write_sql", "deploy_pipeline"}

def tool_executor_with_approval(tool_name: str, tool_fn,
**kwargs) -> str:
    if tool_name in DESTRUCTIVE_TOOLS:
        print(f"\n[Approval required] Agent wants to call:
{tool_name}")
        print(f"    Arguments: {json.dumps(kwargs,
indent=2)}")
        confirm = input("    Approve? (yes/no):
").strip().lower()
        if confirm != "yes":
            return json.dumps({"status": "cancelled",
"reason": "User did not approve this action"})

    return safe_tool_call(tool_fn, **kwargs)

```

Production Agent Patterns

Timeout and cost guard

```

import time

class AgentGuard:
    def __init__(self, max_tokens: int = 50_000,
max_seconds: int = 120):
        self.max_tokens = max_tokens
        self.max_seconds = max_seconds
        self.total_tokens = 0
        self.start_time = time.time()

    def check(self, response):
        self.total_tokens += response.usage.input_tokens +
response.usage.output_tokens
        if self.total_tokens > self.max_tokens:
            raise RuntimeError(f"Token budget exceeded:
{self.total_tokens} > {self.max_tokens}")
        if time.time() - self.start_time >

```

```
self.max_seconds:
    raise RuntimeError(f"Time budget exceeded:
{self.max_seconds}s")
```

Structured output at the end

```
FINAL_ANSWER_TOOL = {
    "name": "final_answer",
    "description": "Call this when you have a complete
answer. Use it to return structured results.",
    "input_schema": {
        "type": "object",
        "properties": {
            "summary": {"type": "string"},
            "data": {"type": "object"},
            "confidence": {"type": "string", "enum":
["high", "medium", "low"]},
            "caveats": {"type": "array", "items":
{"type": "string"}}
        },
        "required": ["summary"]
    }
}
# Force a structured output even from a free-form agent
# The agent uses its tools then calls final_answer with the
structured result
```

Common Pitfalls

Pitfall	Symptom	Fix
Building an agent where a fixed workflow would do	Slow, expensive, unpredictable results	Start with single calls or chained steps; use an agent only for open-ended tasks
Loop handles only <code>end_turn</code> and <code>tool_use</code>	Infinite retries on <code>max_tokens</code> , <code>pause_turn</code> , or <code>refusal</code>	Handle every <code>stop_reason</code> explicitly, or use the SDK's tool runner

Pitfall	Symptom	Fix
Vague tool names and descriptions	Wrong tool chosen, malformed arguments	Clear names, descriptions that say <i>when</i> to use the tool, strict input schemas, examples in descriptions
Too many overlapping tools	The model gets confused and calls the wrong one	A small, orthogonal toolset; tool search for large catalogs
Tools that return huge payloads	Context fills up; cost soars; quality drops	Paginate, summarize, and cap tool outputs; return IDs and let the agent fetch details
Raising exceptions out of tools	The whole run crashes on one bad query	Return errors as <code>tool_result</code> with <code>is_error: true</code> and a helpful message
Write-capable tools with no guardrails	An agent drops a table or emails a customer	Read-only by default, least-privilege credentials, human approval for destructive actions
No budget limits	A runaway loop burns hundreds of dollars	Caps on iterations, tokens, cost, and wall-clock time
Trusting tool output as instructions	Prompt injection from web pages, tickets, or documents	Treat tool results as data; restrict what a single run can do
No traces	Impossible to debug why the agent did something	Log every step: prompt, tool call, arguments, result, tokens, latency

Cheat Sheet

Manual loop skeleton (Claude)

```

messages = [{"role": "user", "content": task}]
while True:
    r = client.messages.create(model="claude-sonnet-5-5",
max_tokens=16000, tools=tools, messages=messages)
    messages.append({"role": "assistant", "content":
r.content})      # keep all blocks
    if r.stop_reason == "end_turn":
        break
    if r.stop_reason == "pause_turn":

```

```

        continue
    if r.stop_reason != "tool_use":
        raise RuntimeError(r.stop_reason)
    results = [{"type": "tool_result", "tool_use_id": b.id,
"content": run_tool(b.name, b.input)}
        for b in r.content if b.type == "tool_use"]
    messages.append({"role": "user", "content": results})
# all results in ONE message

```

Or let the SDK run the loop (Tool Runner)

```

from anthropic import beta_tool

@beta_tool
def get_table_freshness(table: str) -> str:
    """Return the last load time for a warehouse table.

    Args:
        table: Fully qualified table name, e.g.
analytics.marts.fct_orders.
    """
    ...

runner = client.beta.messages.tool_runner(
    model="claude-sonnet-5-5", max_tokens=16000,
    tools=[get_table_freshness],
    messages=[{"role": "user", "content": "Is fct_orders
fresh?"}],
)
for message in runner:          # one message per turn; stops
    when Claude is done
        print(message)

```

Pattern	Use when
Single call with tools	One lookup, then answer
Workflow (fixed chain / routing / parallel calls)	Steps are known in advance
Agent loop	Steps depend on what's discovered along the way

Pattern	Use when
Orchestrator + sub-agents	Broad tasks that split into independent parts (each sub-agent gets its own context)
Human in the loop	Irreversible, costly, or customer-facing actions

Tool definition checklist: verb-noun name (`get_dag_runs`) · a description of *when* to use it · a strict JSON schema · small, structured output · actionable error messages · idempotent where possible

Where the loop runs: your own loop or the SDK tool runner (you host it) · the Claude Agent SDK (the Claude Code harness as a library, with built-in file and shell tools) · Claude Managed Agents (Anthropic hosts the loop and a sandbox) · frameworks like LangGraph

Interview Questions

Q: What is the difference between an LLM workflow and an agent? A: In a workflow, your code defines the steps — call the model to classify, then to extract, then to summarize — and the model fills in each step. In an agent, the model decides the steps itself, choosing which tools to call and when to stop, based on results as it goes. Workflows are more predictable, cheaper, and easier to test; agents handle open-ended tasks where the path can't be known up front. Start with the simplest option that works.

Q: How does tool calling actually work under the hood? A: You send tool definitions (name, description, JSON schema) with the request. When the model wants a tool, it stops with `stop_reason: "tool_use"` and returns a `tool_use` block with an ID and arguments. Your code runs the function and sends back a `tool_result` block with the same ID in the next user message. The model then continues. It never executes code itself — it only proposes calls, so your code controls permissions and side effects.

Q: How do you keep an agent safe when it can modify production systems? A: Defense in depth: give tools least-privilege credentials (read-only unless writes are essential); make destructive tools require human approval; validate arguments in code (allow-listed tables, `LIMIT` on queries, dry-run modes); cap iterations, tokens, cost, and time; treat all tool output as untrusted data (prompt injection); and log every action for audit. Test against adversarial scenarios before giving it more autonomy.

Q: How do you evaluate an agent? A: At two levels. Outcome: does it complete realistic tasks correctly? Build a task suite with verifiable end states (the right answer, the right rows changed) and measure success rate, cost, and steps taken. Trajectory: are the

intermediate steps sensible — right tools, valid arguments, no wasted or dangerous calls? Review traces, use LLM graders for qualitative checks, and re-run the suite on every prompt, tool, or model change. Agent runs vary, so run each task several times.

Q: How do agents manage context over long tasks? A: The conversation grows with every tool call and result, so long tasks can hit context limits and get slower and pricier. Techniques: keep tool outputs small, clear old tool results that are no longer needed (context editing), summarize earlier history (compaction), persist notes to external memory (files or a store the agent reads back), and delegate sub-tasks to sub-agents so each has a clean context and returns only a summary.

Further Reading

- [Anthropic: Building effective agents](https://www.anthropic.com/engineering/building-effective-agents) (https://www.anthropic.com/engineering/building-effective-agents) — workflows vs agents, and common patterns
 - [Claude tool use documentation](https://platform.claude.com/docs/en/agents-and-tools/tool-use/overview) (https://platform.claude.com/docs/en/agents-and-tools/tool-use/overview)
 - [Anthropic: Writing effective tools for agents](https://www.anthropic.com/engineering/writing-tools-for-agents) (https://www.anthropic.com/engineering/writing-tools-for-agents)
 - [Model Context Protocol](https://modelcontextprotocol.io/) (https://modelcontextprotocol.io/) — the open standard for connecting tools and data sources to agents
 - [Claude Agent SDK](https://code.claude.com/docs/en/agent-sdk/overview) (https://code.claude.com/docs/en/agent-sdk/overview)
 - *ReAct: Synergizing Reasoning and Acting in Language Models* — Yao et al., 2022
-

Previous: [Vector Databases](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) · **Next:** [MCP and Text-to-SQL](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mcp-text-to-sql/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mcp-text-to-sql/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026