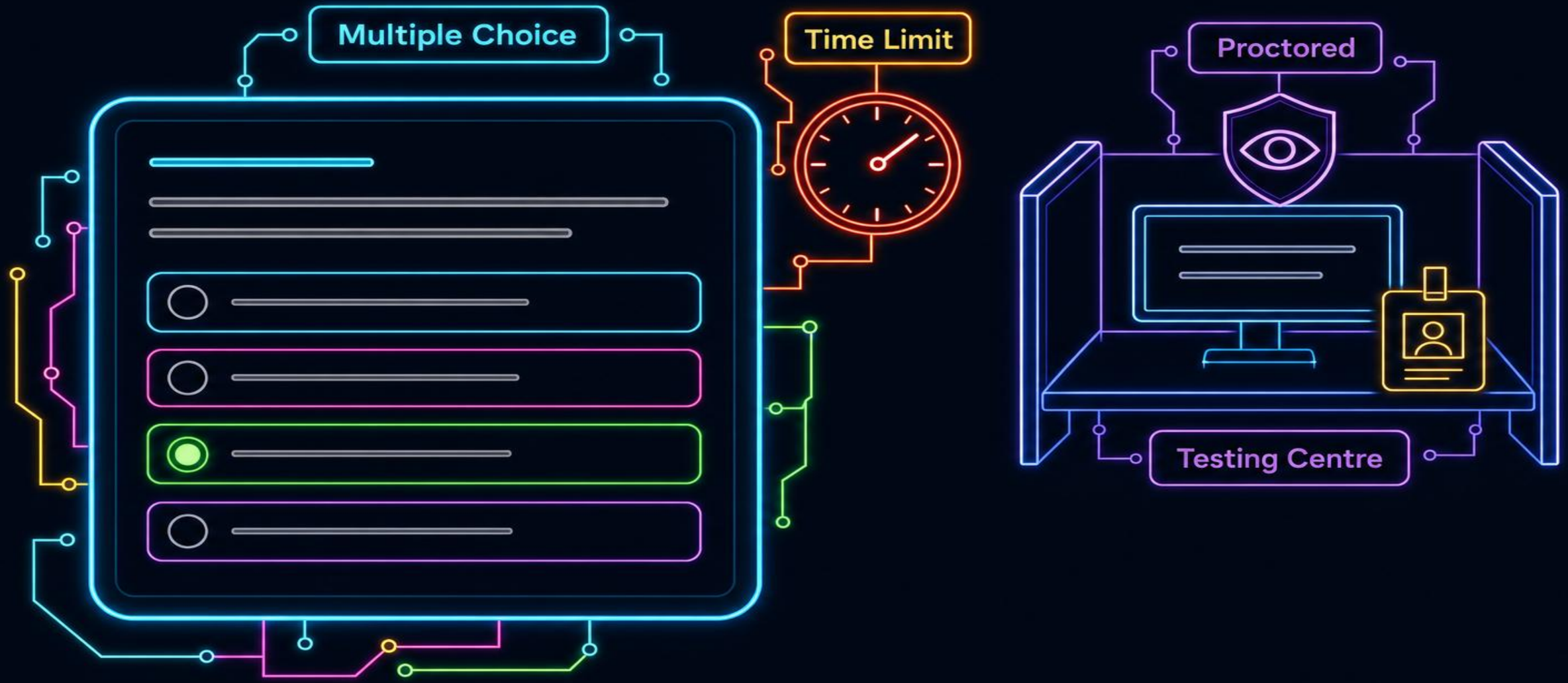
# Welcome to CCDV-F

How the Claude Certified Developer exam works, and why it is built the way it is.



# What CCDV-F Actually Is

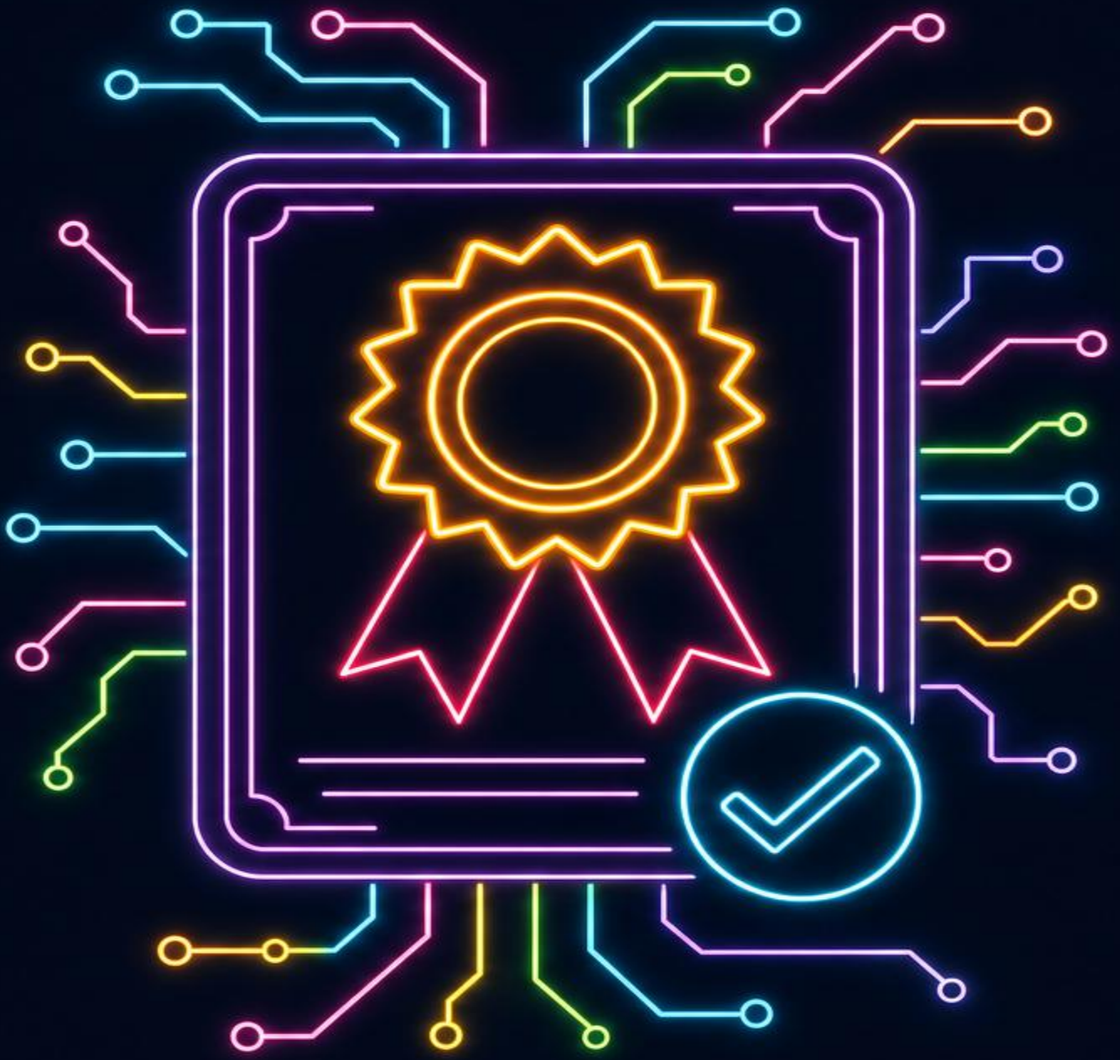
Anthropic's developer credential. It certifies that you can build, integrate, and ship Claude applications.

**Fifty-three questions**, two hours, fully proctored through Pearson VUE

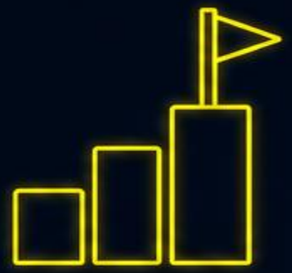
**Eight domains**, from agents and the API through tools, MCP, security, and model selection

**Applied exam.** Realistic build-and-integrate decisions, not memorization recall

**Pass** at seven hundred twenty out of a thousand



# Before You Start - Some Things To Know



## The Bar

This is a developer exam. Anthropic recommends around six months of hands-on Claude experience plus real software engineering time. I assume you already write code and have shipped against the Claude API.



## Not In Scope

No setup walkthroughs. No environment tutorials. No language primers. You should be comfortable in Python or TypeScript, REST, and the command line before you press play.



## If You Are Not There Yet

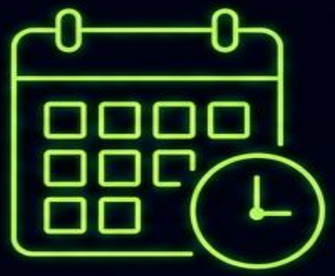
Build a small Claude app first. Get reps with the API and an SDK. Come back when you want sharp, exam-focused preparation, not a getting-started course.

# Why This Cert Is Time-Limited



## Most Certs

Pass once, hold for years. Three to five-year cycles, maintained with continuing-education credits.



## CCDV-F

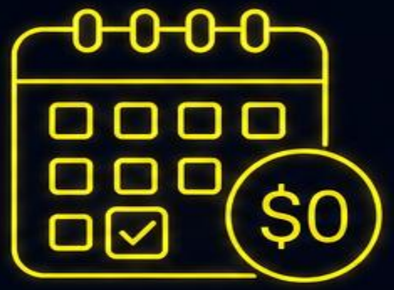
Valid for twelve months. It is a statement that you build with current Claude, not a lifetime badge.



## Why It Fits

The platform moves fast. A developer cert that never expired would vouch for skills that may be a version or two out of date.

# The Part That Makes It Easy




## Free Annual Renewal

Staying certified is a free, non-proctored renewal assessment each year. No exam center, no fee, as long as you keep current.



## Only If You Lapse

You only pay for and re-sit the full proctored exam if you let the credential lapse. Keep up and renewal stays free.



## The Deal

You pay the full exam once. After that, keeping the credential current costs you time, not money.

# Why Anthropic Built It This Way

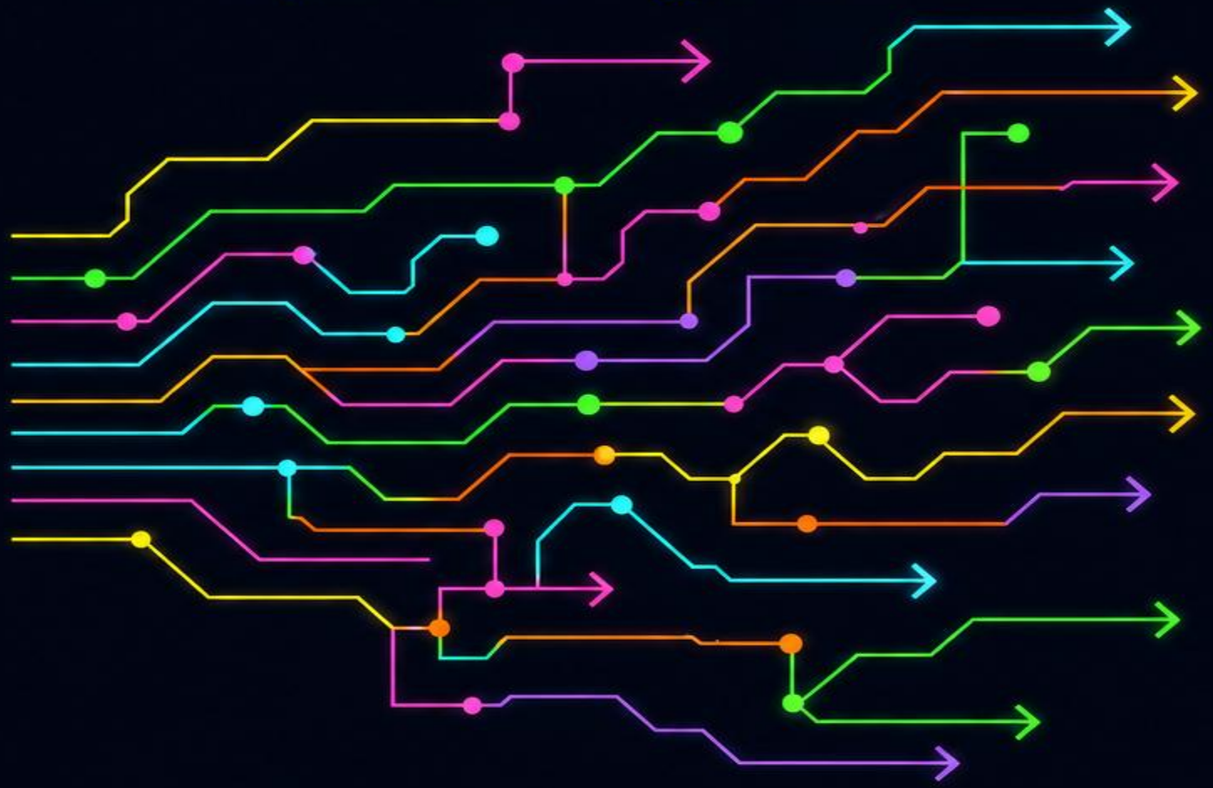
The Claude developer surface changes faster than any multi-year cert cycle could track.

**New model versions** every few months, with different defaults and tradeoffs

**New A.P.I. capabilities**, new S.D.K. surfaces, new M.C.P. patterns shipping

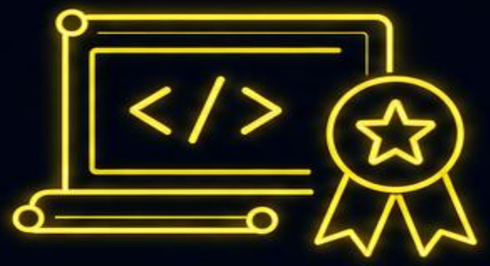
**The right way to build a feature** last year is sometimes the wrong way today

**Rapid Technology Evolution**



A static developer credential here would certify someone who knows old Claude. That helps no one.

# What That Means For You



## As a Candidate

Plan for one focused exam, then a quick free renewal each year. The exam frame stays stable while the content tracks Claude.



## As a Holder

Your CCDV-F is current-tense. It says I build with current Claude, which separates you from people who used it once and stopped.



## As a Job Seeker

Anyone can claim A.I. experience. A current developer credential is verifiable proof you build with the platform as it is today.

# How This Course Tracks the Cert



## Traditional Cert Course

Buy once. Watch through. Pass.
The course is done. Years later when the exam changes, you buy a new course or piece together updates from blog posts.



## This Course

Access does not expire, and the course updates as the platform evolves.
At each renewal window, the current version is already here.

# When New Material Drops



## The Announcement

When content drops, you get an announcement through Udemy. What is new, what got reshot, and what to revisit before your renewal.



## The Practice Tests

Every update ships paired with refreshed, exam-current practice questions, on the same cadence as the lessons.



## Your Renewal Loop

Watch the new lessons, run the refreshed practice tests, take the free renewal assessment. Same loop each year.

# Why No Domain Weights Here

On a typical cert welcome, this is where I'd list every domain percentage.

**Not here.** Weights and emphasis shift as the platform evolves

**Recording time** goes into actual lesson content instead

**The current breakdown** lives on Anthropic's certification page

Trust the modules over this welcome video.
The modules are the artifact that gets updated.



# How To Use This Course

## First-Time Candidate

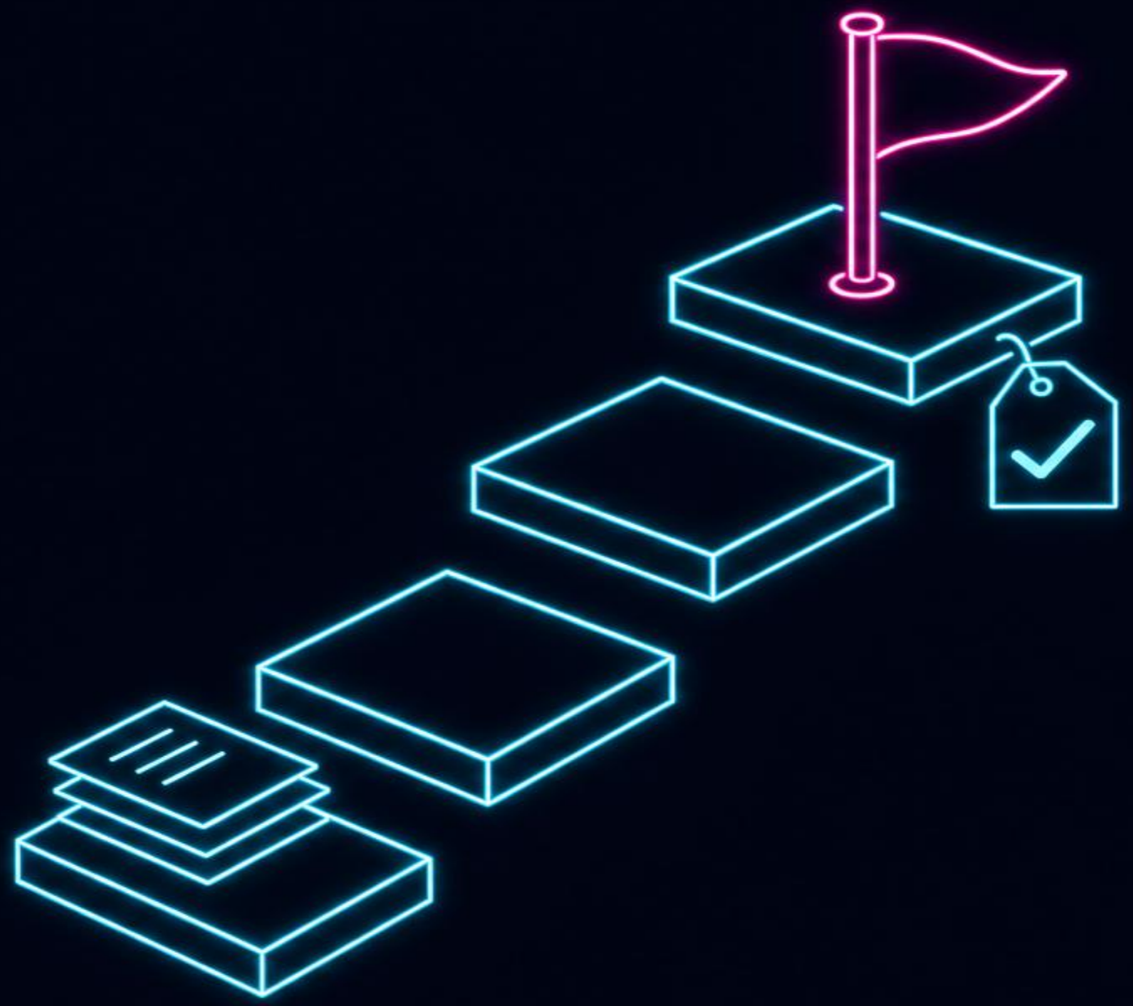
Watch the modules in order across all eight domains. Run the practice tests until you are consistently above the passing line. Then take the exam.

## Returning to Renew

Read the announcement. Watch the new and updated lessons. Run the refreshed practice tests, then take the free renewal assessment.

## Practice Exams

A full set of graduated practice exams is available here, updated on the same cadence as this course.



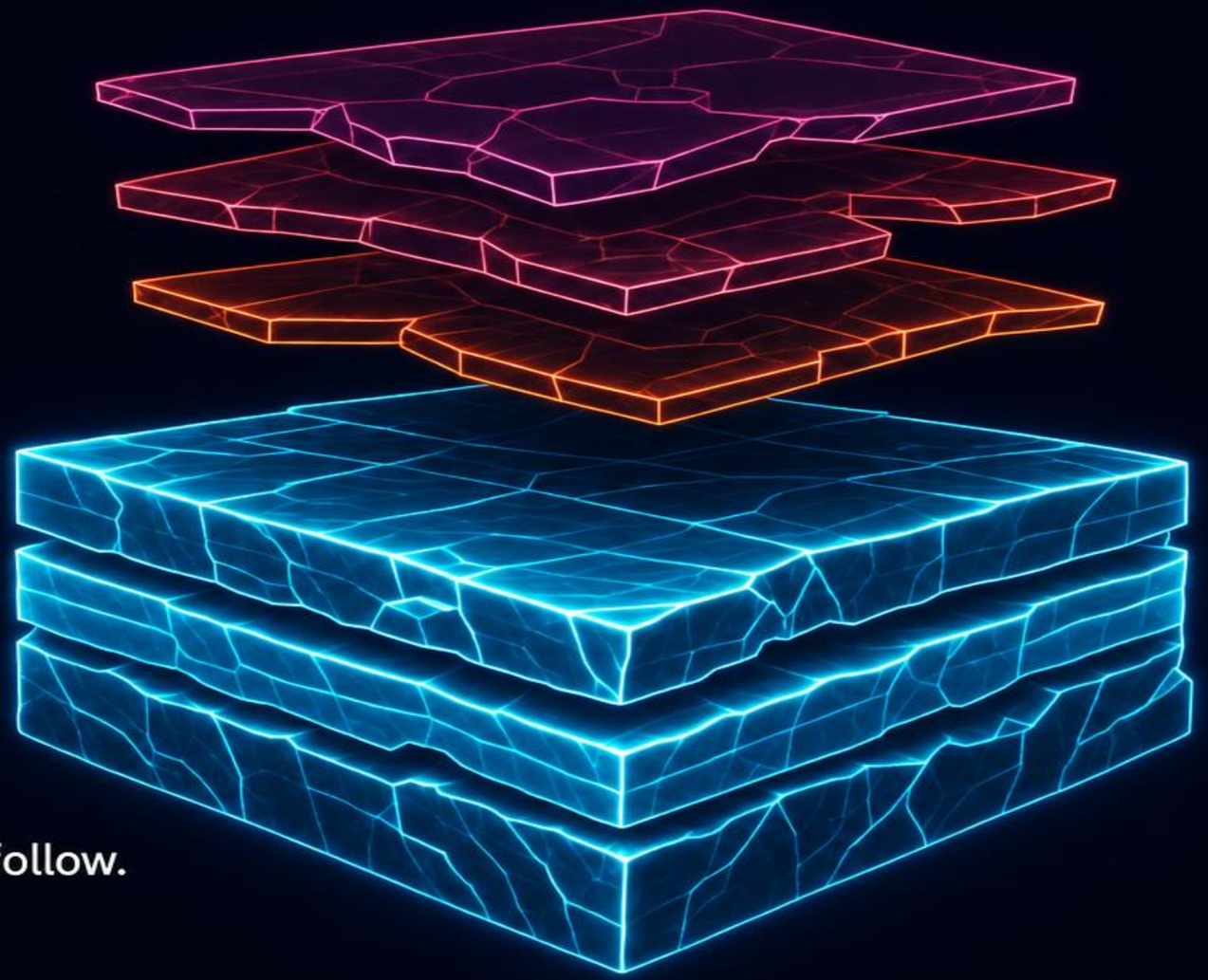
# Patterns Over Signatures

The single study habit that pays off every renewal on this credential.

**Learn the reasoning**
behind a design, not the exact A.P.I. call signature

**Method names change.**
Parameters change. Limits change. The why is durable

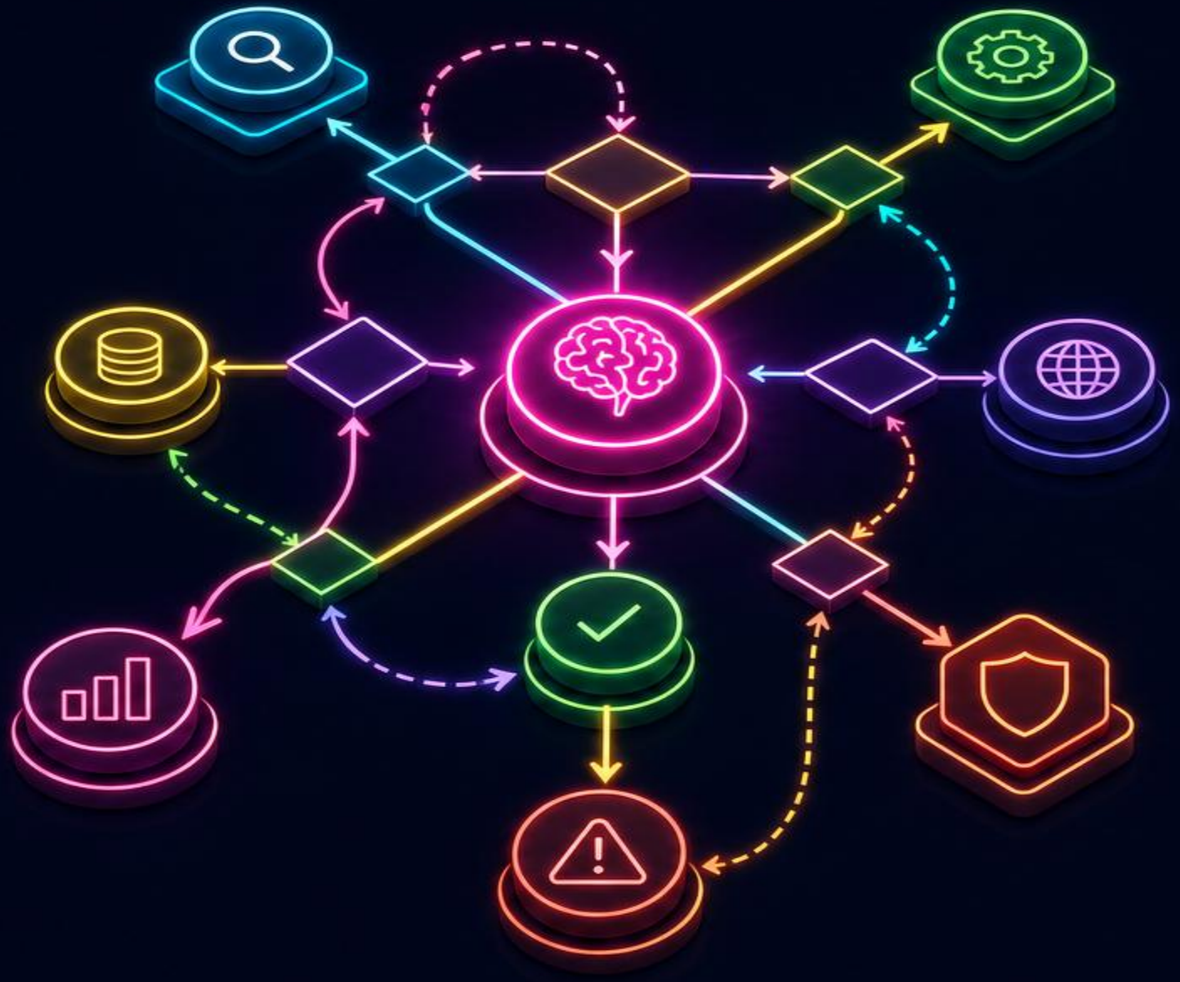
**Each topic flags**
what is stable and what is volatile



Internalize the reasoning. The signatures will follow.

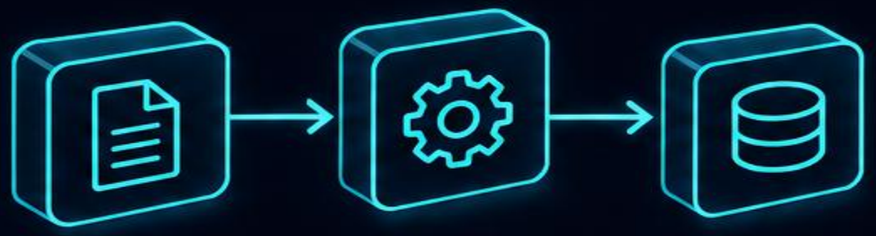
# Agent or Workflow

Not every task that calls a model needs an agent.



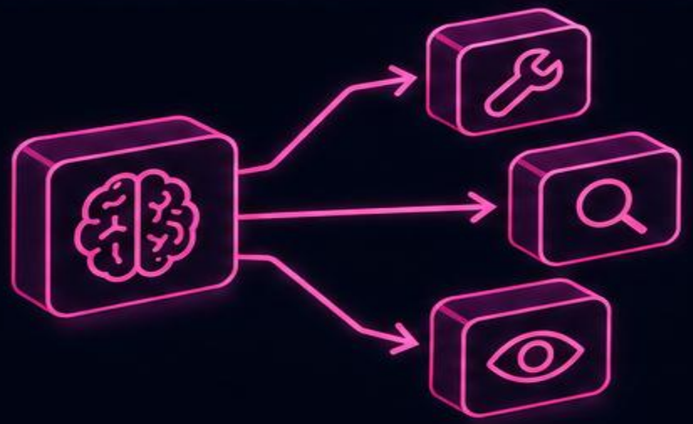


# Workflow vs Agent



## Workflow

A developer-authored sequence where control flow is fixed at design time. Your code decides the order, and the model just fills in each step.



## Agent

A system where the model decides the next action at runtime, driving a loop of reason, act, and observe until it hits a stopping condition.

# The Augmented LLM



## Retrieval

Pulling relevant context into the prompt so the model answers from real information instead of guessing.

```python
from anthropic import Anthropic
client = Anthropic()

docs = store.search(question, k=3)
prompt = f"{docs}\n\n{question}"
```



## Tools

Letting the model take actions in the world, like querying a database or calling a service.

```python
response = client.messages.create(
  model="claude-opus-5",
  max_tokens=1024,
  tools=[get_weather, query_db],
  messages=[{"role":"user","content":q}],
)
```



## Memory

Carrying state across a task so the model can build on what it has already done.

```python
client.messages.create(
  model="claude-opus-5",
  max_tokens=1024,
  messages=history,
  metadata={"user_id": "abc123"},
)
```

# Who Decides the Next Step

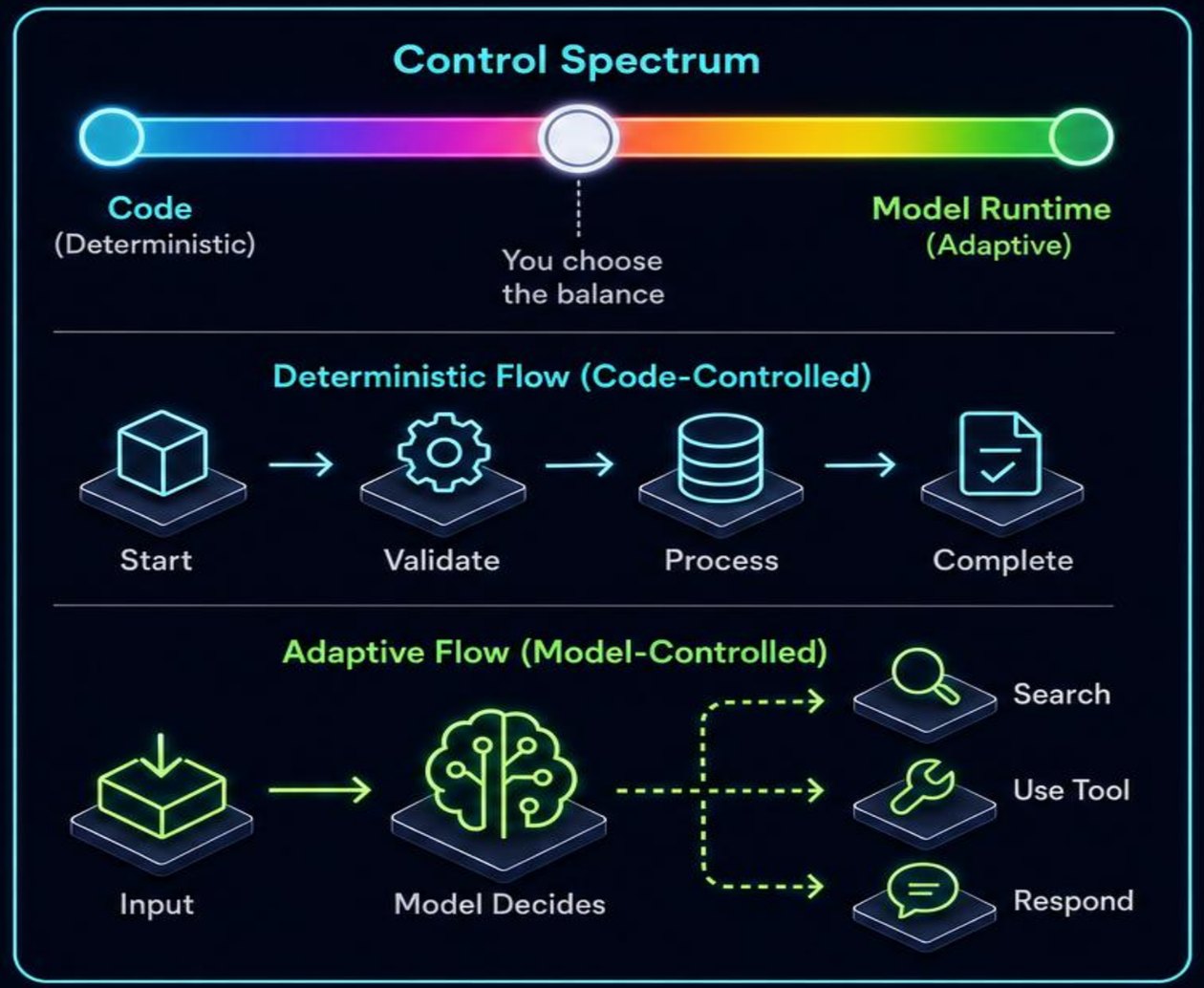
Every agentic system sits on a single axis, and that axis is control.

**At one end,**
your code owns the order of operations

**At the other,**
the model owns it at runtime

**Fixed paths**
are predictable and easy to test

**Model-driven paths**
adapt to inputs you didn't foresee



# Start With the Simplest Thing

Anthropic's guidance is blunt. Reach for the simplest design that works.

**SIMPLE**



**COMPLEX**



**Add complexity** only when it clearly improves the outcome

**Agents are the expensive** option, so justify the loop

**Many tasks** need just one call with retrieval and examples

**Measure the simple** version before you build more

# Three Signals for an Agent



## Open-Ended Space

The task can unfold in many directions you simply can’t list out up front.




## Unknown Steps

You can’t predict how many actions the job will really take to finish.



## Model Chooses Tools

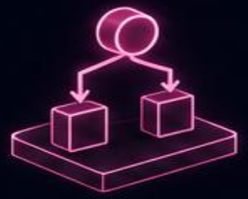
Part of the value is letting the model itself pick which tool and when.

# Is a Workflow Enough?

Run the cheap test before you reach for a loop.



**Can you write**
the steps out in a fixed order?



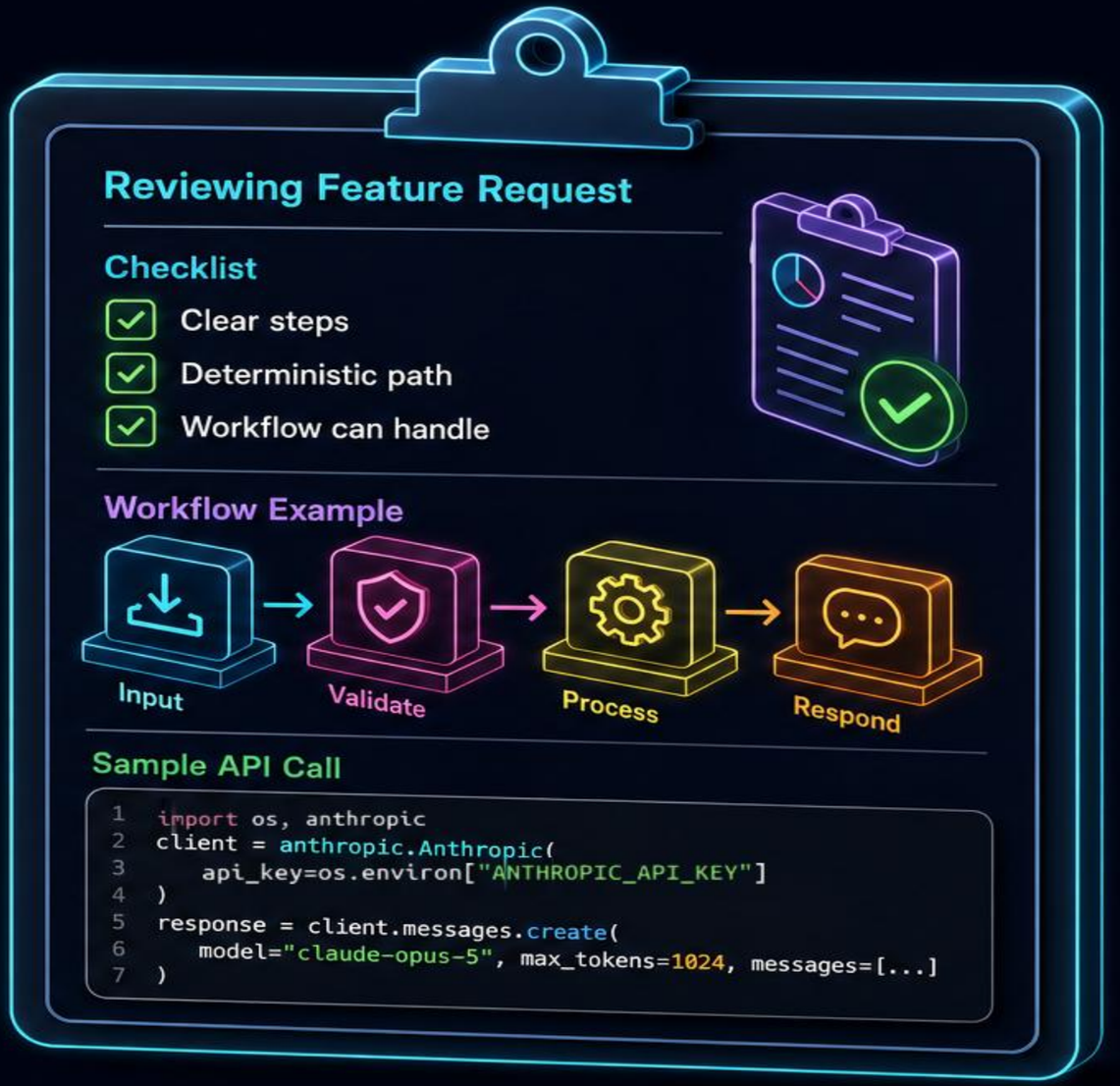
**Does the same input**
always take the same path?



**If yes,**
a deterministic workflow
is enough



**Prove the workflow**
can't cope before paying
for an agent



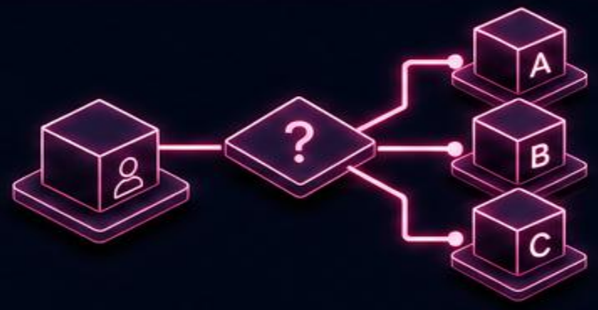
# Three Workflow Shapes




## Prompt Chaining

Break a task into a fixed sequence where each call works on the previous output.

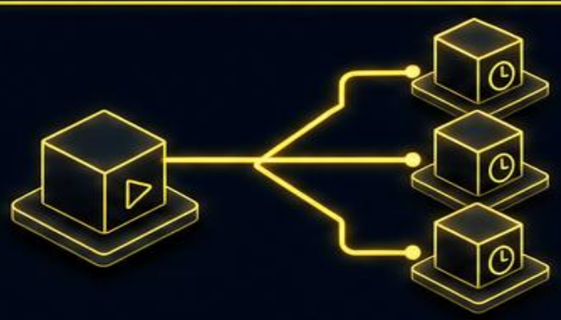
```python
import anthropic
client = anthropic.Anthropic()
r1 = client.messages.create(
  model="claude-opus-5", max_tokens=1024,
  messages=[{"role": "user","content":"Draft outline"}]
)
r2 = client.messages.create(
  model="claude-opus-5", max_tokens=1024,
  messages=[{"role": "user","content":r1.content[0].text}]
)
```




## Routing

Classify an incoming request and send it to the handler built for that case.

```python
route = client.messages.create(
  model="claude-opus-5", max_tokens=1024,
  messages=[{"role":"user","content":request_text}],
  tools=[router_tool]
)
target = route.content[0].name
result = handlers[target](request_text)
```



## Parallelization

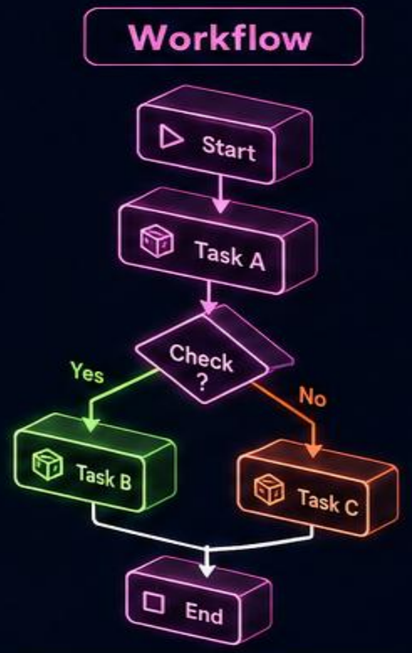
Run independent pieces at once, or run the same task several times and compare.

```python
prompts = [p1, p2, p3]
batch = client.messages.batches.create(requests=[
  {"custom_id": f"p{i}", "params": {"model": "claude-opus-5",
   "max_tokens": 1024,
   "messages": [{"role": "user", "content": p}]}}
  for i, p in enumerate(prompts)
])
results = client.messages.batches.results(batch.id)
```

# Match the Design to the Task

Turn the decision into one clean habit.







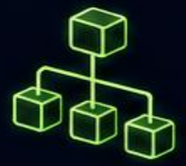
```
{
  "type": "sequence",
  "steps": [
    "step_1",
    "step_2",
    "step_3",
    "step_4"
  ]
}
```

```
{
  "type": "workflow",
  "nodes": ["start","task_a",
    "check","task_b","task_c",
    "end"],
  "edges": "start->task_a->check",
  "branches": {"check": [
              ["task_b","task_c"]}
}
```

```
{
  "type": "agent",
  "model": "claude-opus-5",
  "tools": ["search","calc","lookup"],
  "memory": "enabled",
  "max_steps": 15
}
```



**Write the task**
as a sequence of steps first



**If the path is stable,**
build the workflow and stop



**If steps won't enumerate,**
an agent earns its keep



**Hybrid designs**
wrap an agent inside
one workflow stage

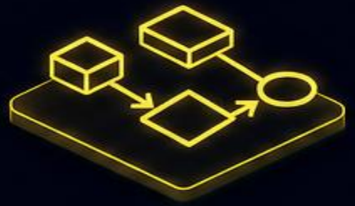
# When Workflows Win



**Prefer the simplest design** that solves the task



**Three signals:** open space, unknown steps, model-chosen tools



**Workflow shapes:** prompt chaining, routing, parallelization



**Hybrids** wrap an agent inside a workflow stage

# Counting the Cost of a Loop

Agents are powerful, but every extra loop has a price.





# Three Tradeoff Axes



## Latency

A loop makes several calls in sequence, so users wait longer than for one response.




## Token Cost

Every turn re-sends the growing context, so the bill climbs with each step.



## Task Performance

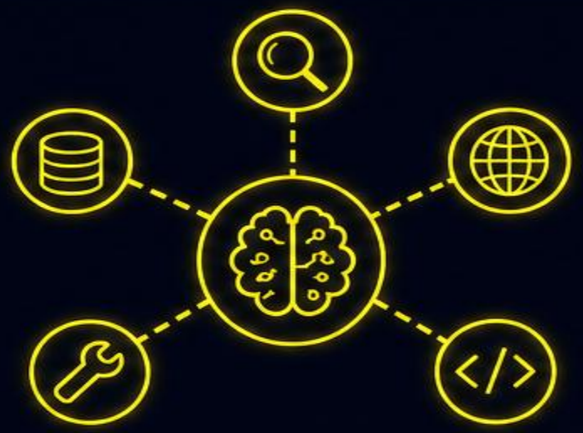
What you buy with the other two: the model's ability to adapt to messy inputs.

# The Core Tradeoff



## Deterministic Workflow

Predictable, cheaper, and faster.
It does the same thing on every run,
which makes it easy to test and easy to trust.



## Model-Driven Agent

Flexible and adaptive,
but it costs more, runs slower,
and opens the door to mistakes
a script would never make.

# Why a Loop Costs More

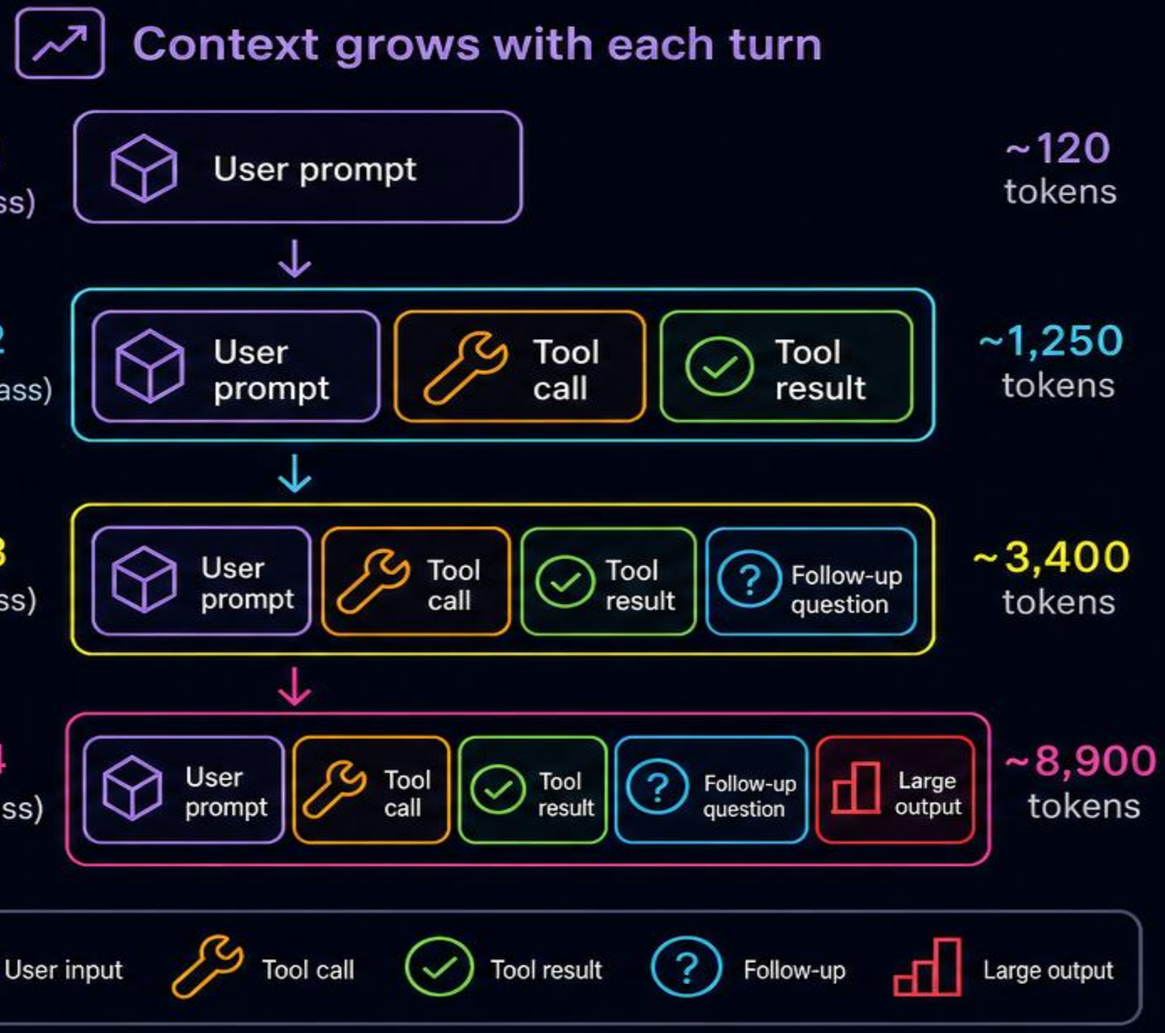
In a loop, the context doesn't reset between turns.

**Everything accumulates:** instructions, tool calls, and results

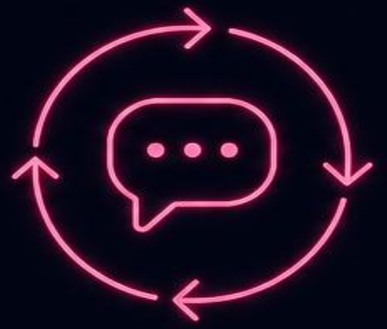
**You pay for the whole history again** on each pass

**Large tool outputs** are the biggest cost driver

**Latency stacks,** since each turn is another round trip



# Where Agents Add Cost



## More Turns

The loop keeps calling the model until the task is done, and each call bills.



## Bigger Context

Every turn carries the full history, so later turns are the priciest ones.



## Compounding Errors

A wrong step early feeds every step after it, and the model builds on the mistake.

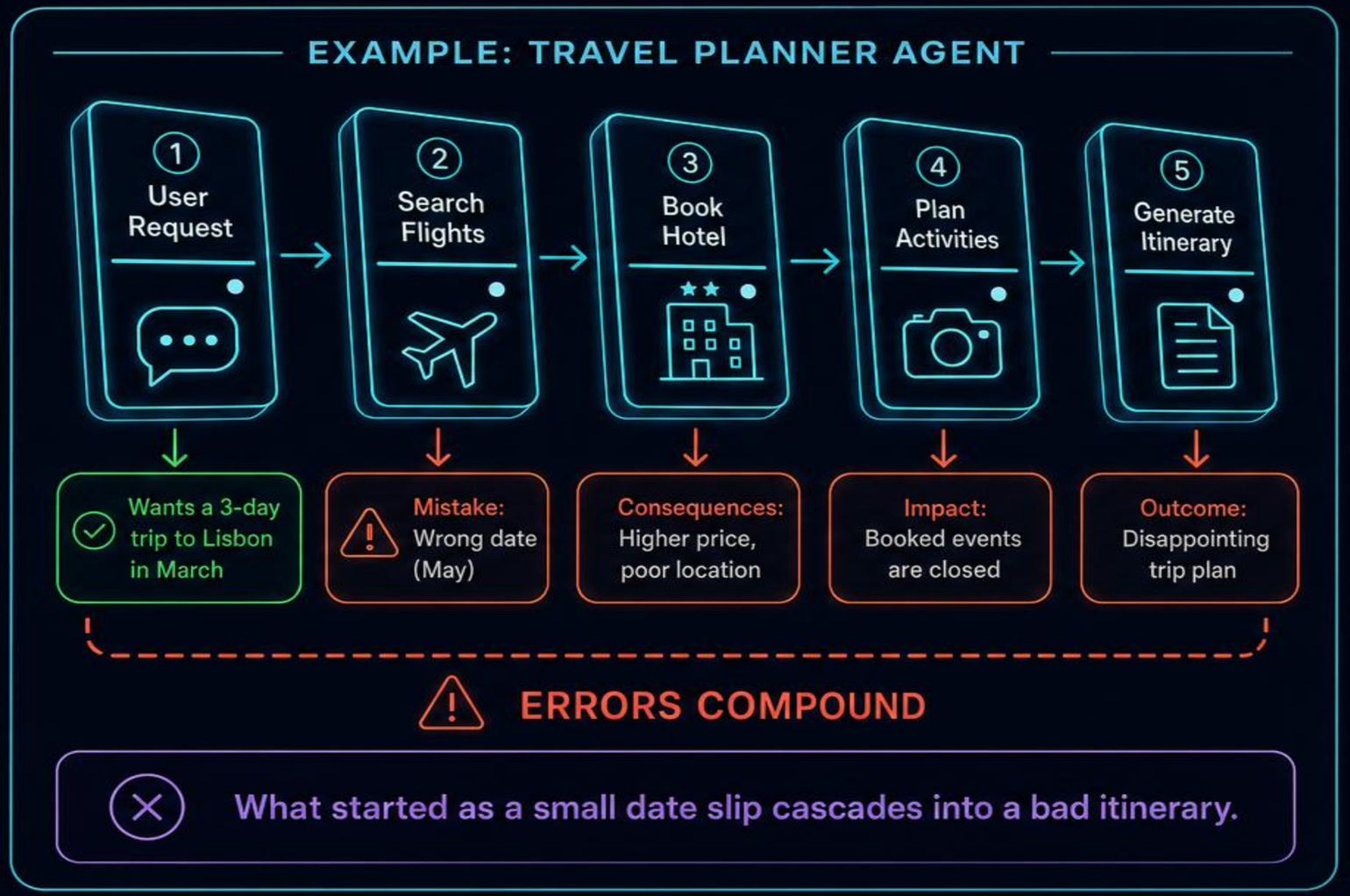


## Slower Responses

Sequential round trips add up into real wall-clock time your user feels.

# How Errors Compound

Compounding errors are the sneakiest cost of a loop.



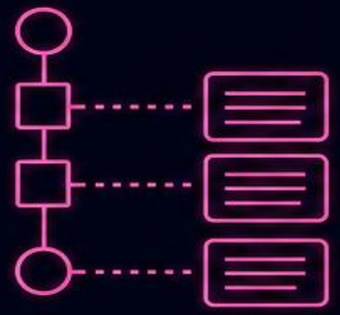

**A small early mistake**
becomes input to the next step

**The model treats**
its own flawed output as fact

**A few turns later**
it can be confidently off track

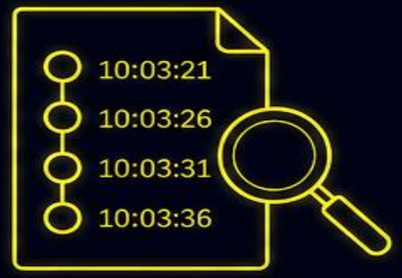
**Longer autonomous runs**
carry more risk than short ones

# Why Workflows Stay Reliable



## Predictability

A fixed path produces the same behavior every run, so you can reason about it in advance.




## Auditability

You can trace exactly which step ran, in what order, and why.



## Consistency

No runtime improvisation, so the behavior doesn't drift from one call to the next.

# Validate the Workflow First

When a problem could be either, build the workflow version first.



**It's faster to stand up** and easier to test



**Run it against real inputs** and see how far it gets

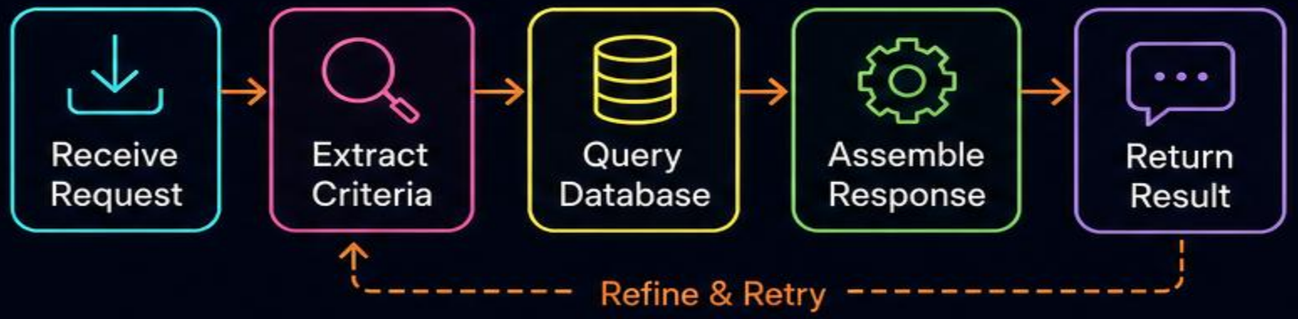


**Often it gets far enough,** saving an agent you didn't need



**Where it falls short** tells you what needs flexibility

## Example Workflow Prototype



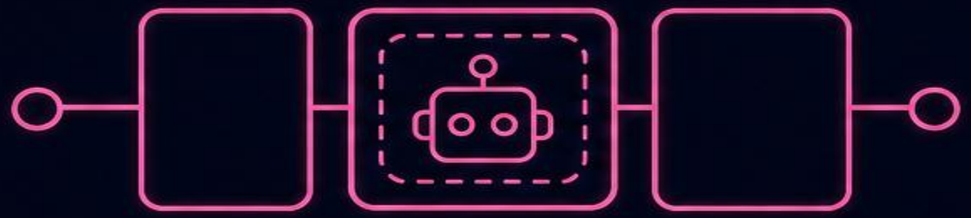

## Tested Against Sample Inputs

| Sample Input | Workflow Result | Outcome |
|---|---|---|
| Find cafés open now near Mission, SF | Found 5 cafés with hours and distance | Succeeded |
| What's the refund policy for orders? | Retrieved policy from help center | Succeeded |
| Compare plans for team of 25 | Missing pricing details | Fell Short |



Insight: Lacks dynamic pricing lookup.
Add pricing service + conditional logic.

# Hybrid Designs



## Agent Inside a Stage

Drop a scoped agent into just the one open-ended step of an otherwise fixed pipeline, where flexibility is genuinely needed.



## Workflow Around It

The surrounding workflow handles ordering, validation, and error handling, so you get predictability at the edges and adaptability only in the core where it counts.

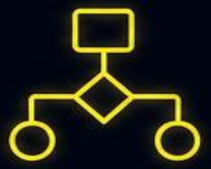
# Buying Flexibility With Cost



**A loop re-sends its growing context every turn** – More turns mean more tokens, more latency, and more spend with diminishing returns.



**Compounding errors are the sneakiest hidden cost** – Small mistakes multiply across turns, corrupting context and derailing good outcomes.



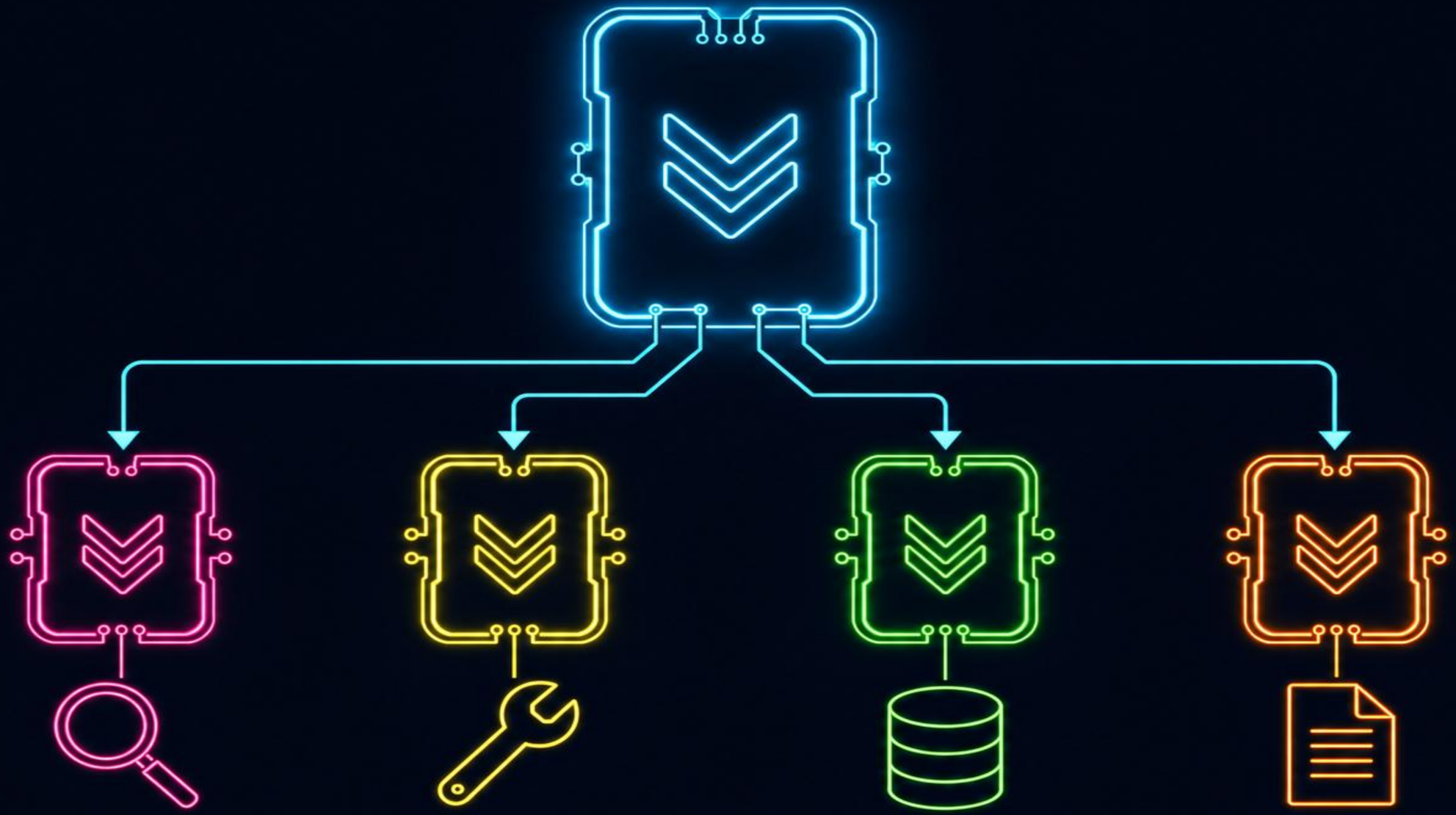
**Workflows give predictability, auditability, and consistency** – Locks in paths, surfaces errors early, and delivers repeatable results at lower risk.



**Validate the workflow first, then add an agent** – Prove the process is sound before introducing autonomy into the loop.

# Manager and Subagent Orchestration

When one agent tries to do everything, its context fills up and quality drops.

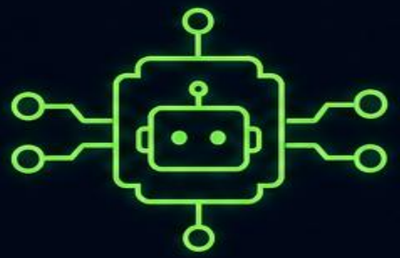


# Three Core Roles



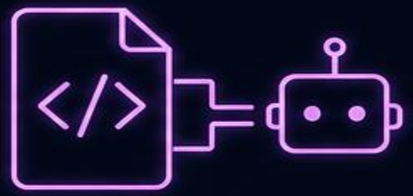
## Manager

The orchestrator agent that owns the goal. It decomposes the task, delegates scoped pieces, and synthesizes the returned results.



## Subagent

A separate agent instance the manager spawns for one focused subtask. It runs on its own and reports a result back.



## Delegation

Handing a scoped subtask to a subagent through the Agent tool, so the model decides when to spin one up.

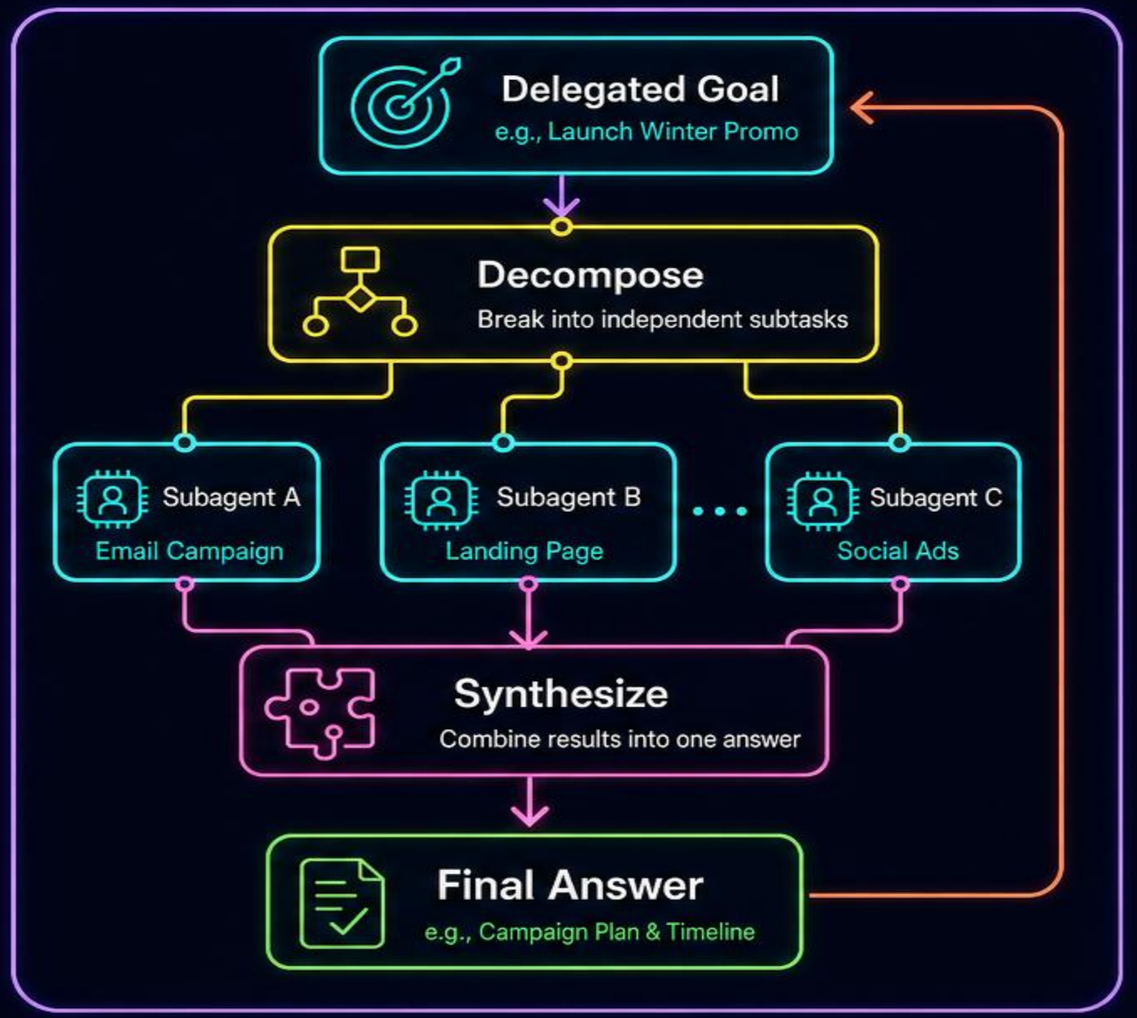
# What the Manager Does

A manager runs a repeatable loop over every delegated goal.

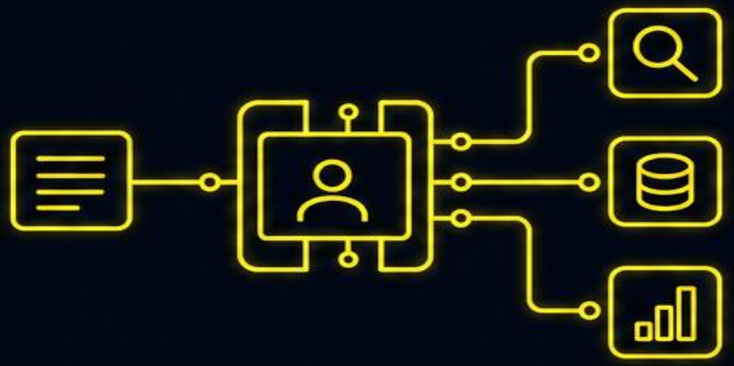
**Decompose**
the goal into independent, scoped subtasks

**Delegate**
each subtask to a subagent built for it

**Synthesize**
the returned results into one coherent answer

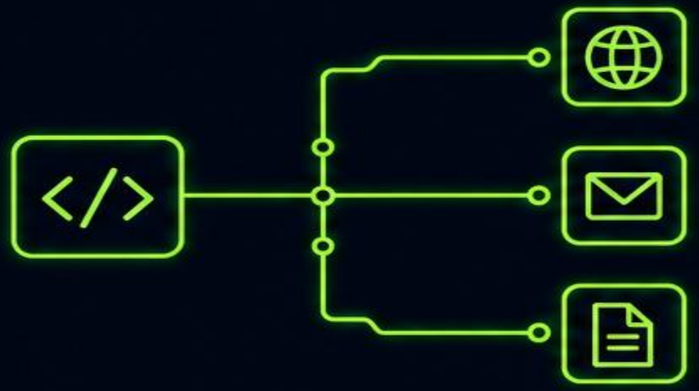


# Orchestrator-Workers vs Parallel



## Orchestrator-Workers

A central agent decides the subtasks dynamically at runtime, based on the specific input it receives. The work is not known in advance.

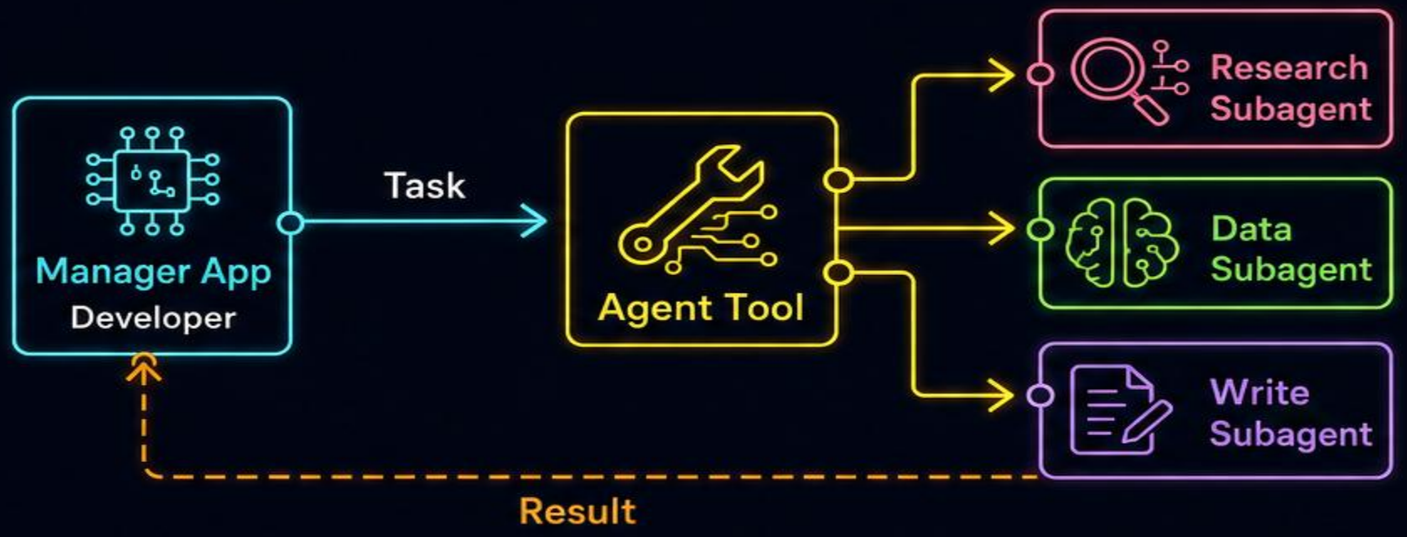


## Parallelization

Subtasks are pre-defined by your own code, then run at the same time.
The split is fixed before any model call.

# Delegating Through the Agent Tool

In the Agent SDK, a manager delegates by calling the built-in Agent tool.




**Each subagent**
carries a description the model reads

**Claude matches**
that description to the task at hand

**You can also name**
a subagent directly in the prompt

AGENT TOOL INVOCATION

```python
import asyncio
from claude_agent_sdk import query, ClaudeAgentOptions, AgentDefinition
opts = ClaudeAgentOptions(
    allowed_tools=["Read", "Grep", "Agent"],
    agents={"data-analyst": AgentDefinition(
        description="Analyzes sales data. Use for metrics and trends.",
        prompt="You are a data analyst. Report findings concisely.")})
async def main():
    async for m in query(prompt="Analyze Q2 sales and draft a summary.", options=opts):
        print(getattr(m, "result", ""))
asyncio.run(main())
```

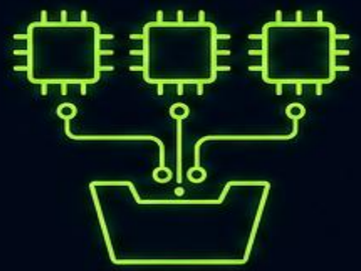
Allow-list the Agent tool so delegations auto-approve.

# Why Delegate at All



## Context Isolation

The subagent's intermediate reasoning and tool output stay inside it, so the manager's context stays lean.



## Parallelization

Independent subagents run concurrently, so a batch finishes in the time of the slowest one, not the sum.



## Tool Restriction

Each subagent gets only the tools its assigned task needs, which shrinks its blast radius.

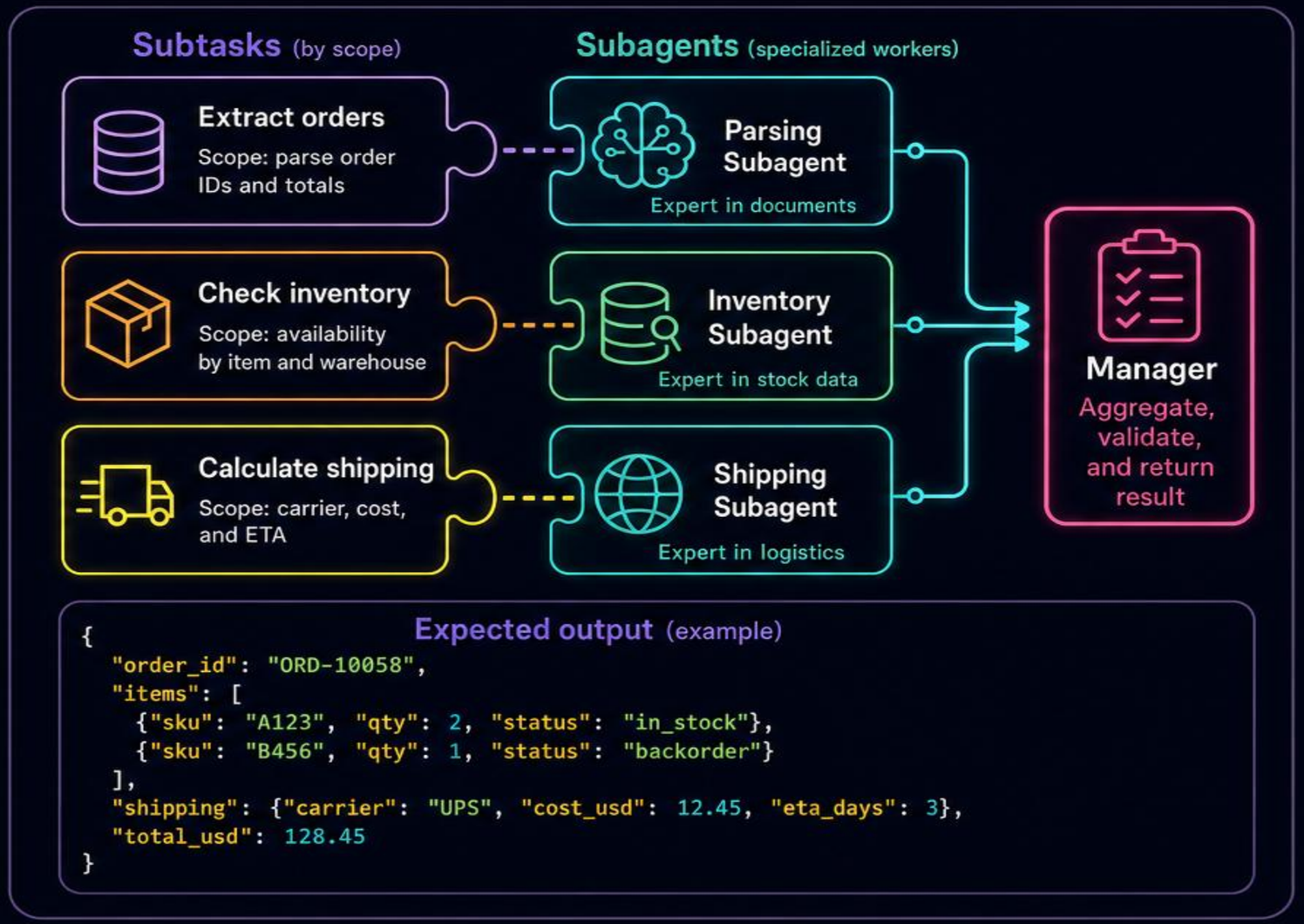
# Matching Subtasks to Subagents

Good delegation starts before any subagent runs, in how you scope the work.

**Define each subtask**
by its scope, one clear job per subagent

**State the output format**
you expect back from it

**At the manager,**
collect and validate every returned result



# Three Guardrails on Scope



## Least Privilege

Give a subagent only the tools and permissions its task requires, so it can't act outside its lane.



## Side-Effect Boundary

A scoped subagent should produce no writes or actions beyond its assigned subtask.



## Result Validation

The manager checks each returned result before trusting it in the final answer.

# Anti-Patterns to Avoid

Most delegation failures trace back to a handful of avoidable mistakes.

## Anti-Pattern in Action

**User Request**

"Launch a campaign and analyze results"

**Thin Context**

Budget: $5k
Goal: More clicks

Missing key details

**Overloaded Subagent**

- Plan campaign
- Write copy
- Design assets
- Schedule ads
- Run ads
- Analyze data
- Report results

Too many jobs

**Unreliable Outcome**

Missed goal. Wasted budget. No clear insights.

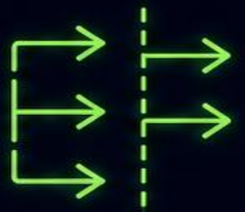
## Common Anti-Patterns



**Overly broad scope,** so the subagent tries to do too much



**Underpowered context,** handing a subagent too little to succeed



**Parallel-only thinking,** ignoring the context-isolation payoff

# When to Reach for a Manager



**A manager** decomposes, delegates, and synthesizes results



**Orchestrator-workers** decide subtasks at runtime, unlike fixed parallelism



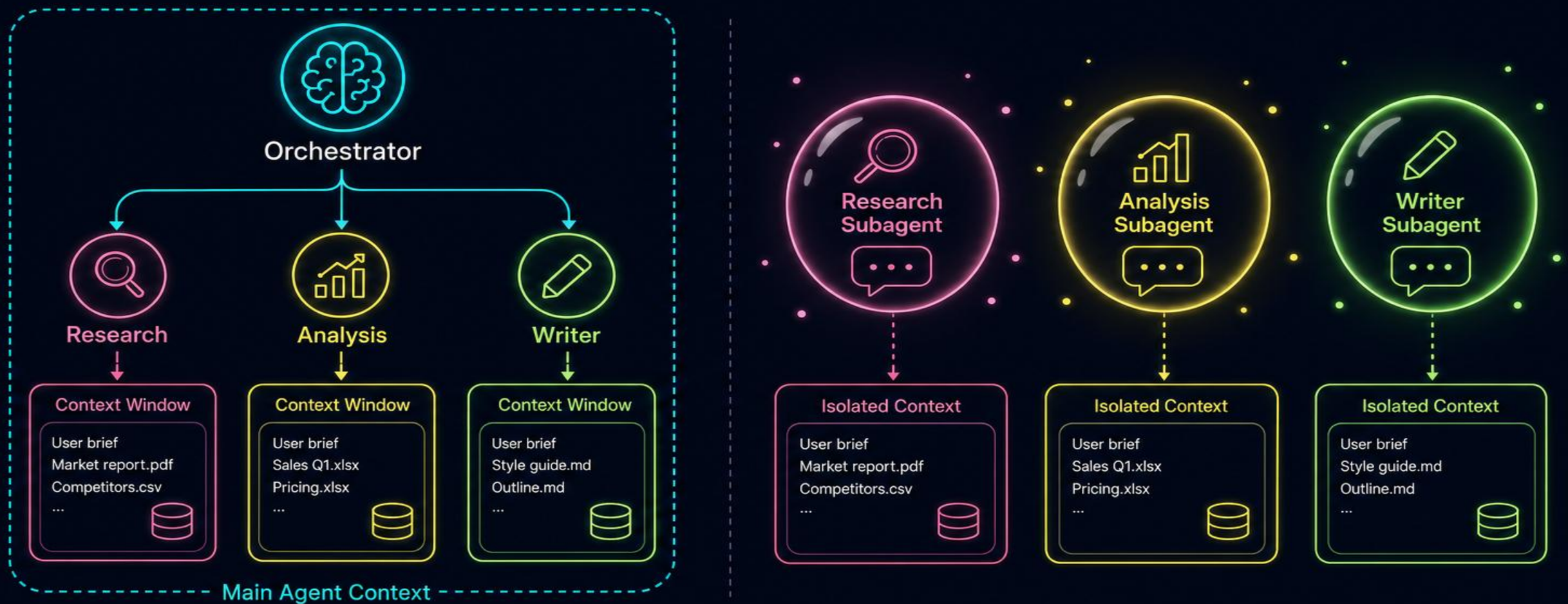
**Scope, least privilege,** and result validation keep it reliable



**Avoid** broad scope and parallel-only thinking

# Why Subagents Run in Isolation

Every subagent starts in its own fresh context and sees only what you hand it.



# Three Isolation Basics



## Own Context Window

Each subagent runs in its own fresh conversation, with a context window fully separate from the manager's.



## No Inherited History

A subagent does not receive the manager's conversation, tool results, or system prompt.

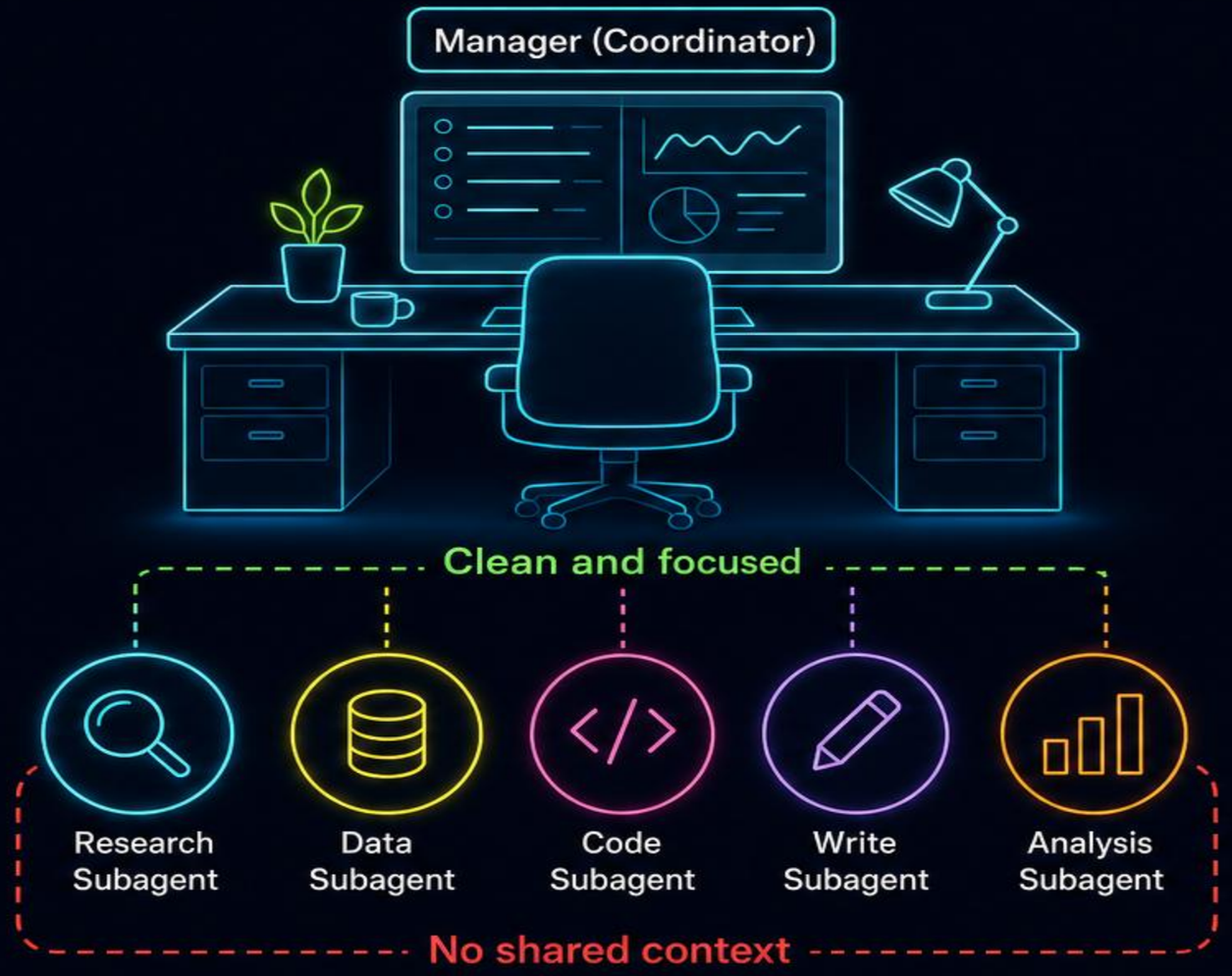


## Final Message Returns

Only the subagent's final message goes back to the manager, not its intermediate steps.

# Why Each Subagent Is Isolated

Isolation isn't an accident of the design; it's the point of delegating.




## It keeps the manager's context clean and focused

Isolated subagents keep their details out of the manager's working memory, reducing noise and distraction.

## It bounds token growth as the number of subtasks rises

Each subagent uses only the context it needs, preventing runaway context expansion and higher costs.

## It lets each subagent focus on one problem

Narrow scope enables deeper reasoning, better tools, and more reliable results for each specific task.

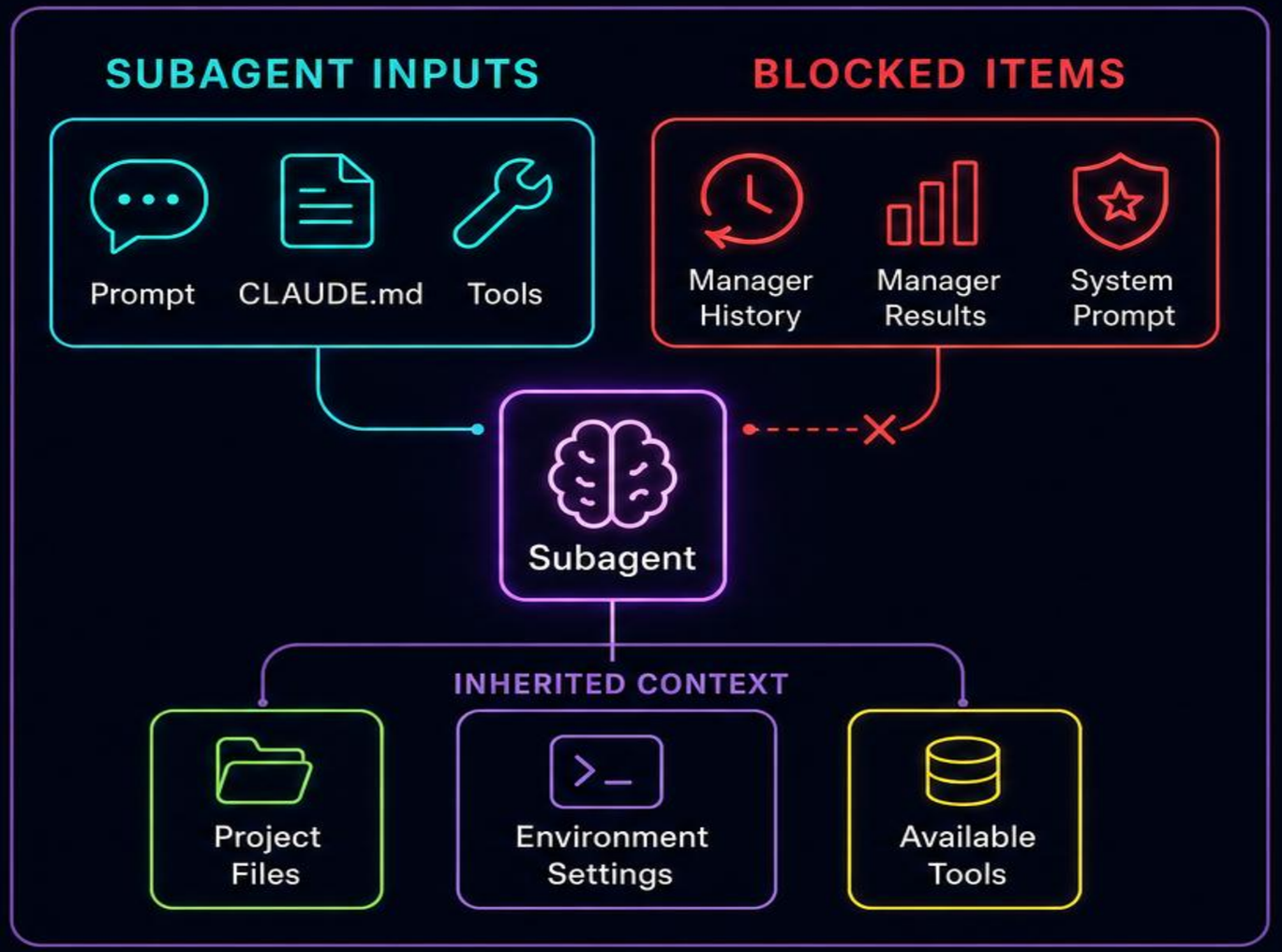
# What a Subagent Sees

A subagent starts from a known, limited set of inputs.

**Inherits:** its own prompt, project CLAUDE.md, and tools

**Receives:** anything you explicitly pass in the task

**Never sees:** the manager's history, results, or system prompt



# Passing Explicit Inputs



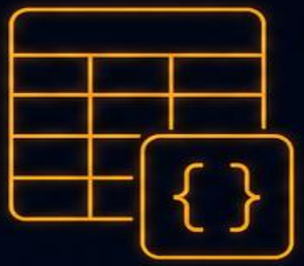
## Task Instructions

State exactly what the subagent should do and what done looks like.



## Relevant Data

Hand over the specific facts, files, or values the subtask depends on.




## Output Shape

Name the format you want back, a summary, a list, or structured data.



## Scoped Tools

Give only the tools this task needs, nothing that widens its reach.

# Assumed vs Actual Context



## The Wrong Assumption

Developers expect a subagent to share the manager's full context, seeing everything the manager has already seen and done.



## The Actual Behavior

A subagent sees only its own prompt and the inputs you pass. Shared context is the exception you build, not the default you get for free.

# Isolation Bounds Token Growth

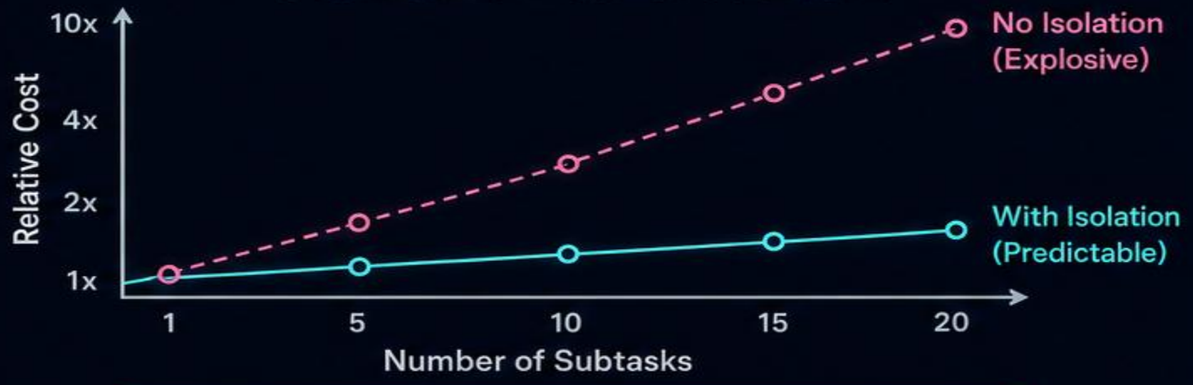
Isolation is also what keeps a multi-subagent run affordable.

## Token Meter (Manager Window)




Steady growth: only final summaries are kept

## Cost vs. Number of Subtasks




Isolation keeps cost growth linear and predictable.

### The manager's window grows only by each final summary

Only compact results are retained. Intermediate detail stays local.

### A subagent's intermediate tokens never reach the parent

Internal reasoning and tool calls are contained within the subagent.

### Adding subtasks scales cost predictably, not explosively

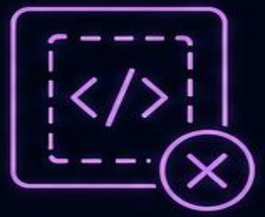
Isolation ensures linear cost growth as you scale more work.

# Three Authority Boundaries



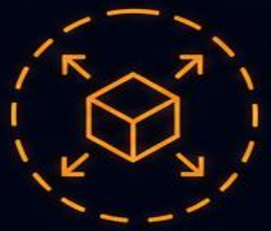
## Least Privilege

A subagent holds only the permissions its task requires, so isolation covers authority, not just context.



## Side-Effect Limits

A well-scoped subagent produces no writes or actions outside its assigned subtask.



## Contained Blast Radius

If a subagent misbehaves, its limited scope and tools keep the damage small and easy to trace.

# The Parallel-Worker Trap

A common misread is seeing subagents as just a speed trick.

### Parallel speed is real, but it's the smaller benefit

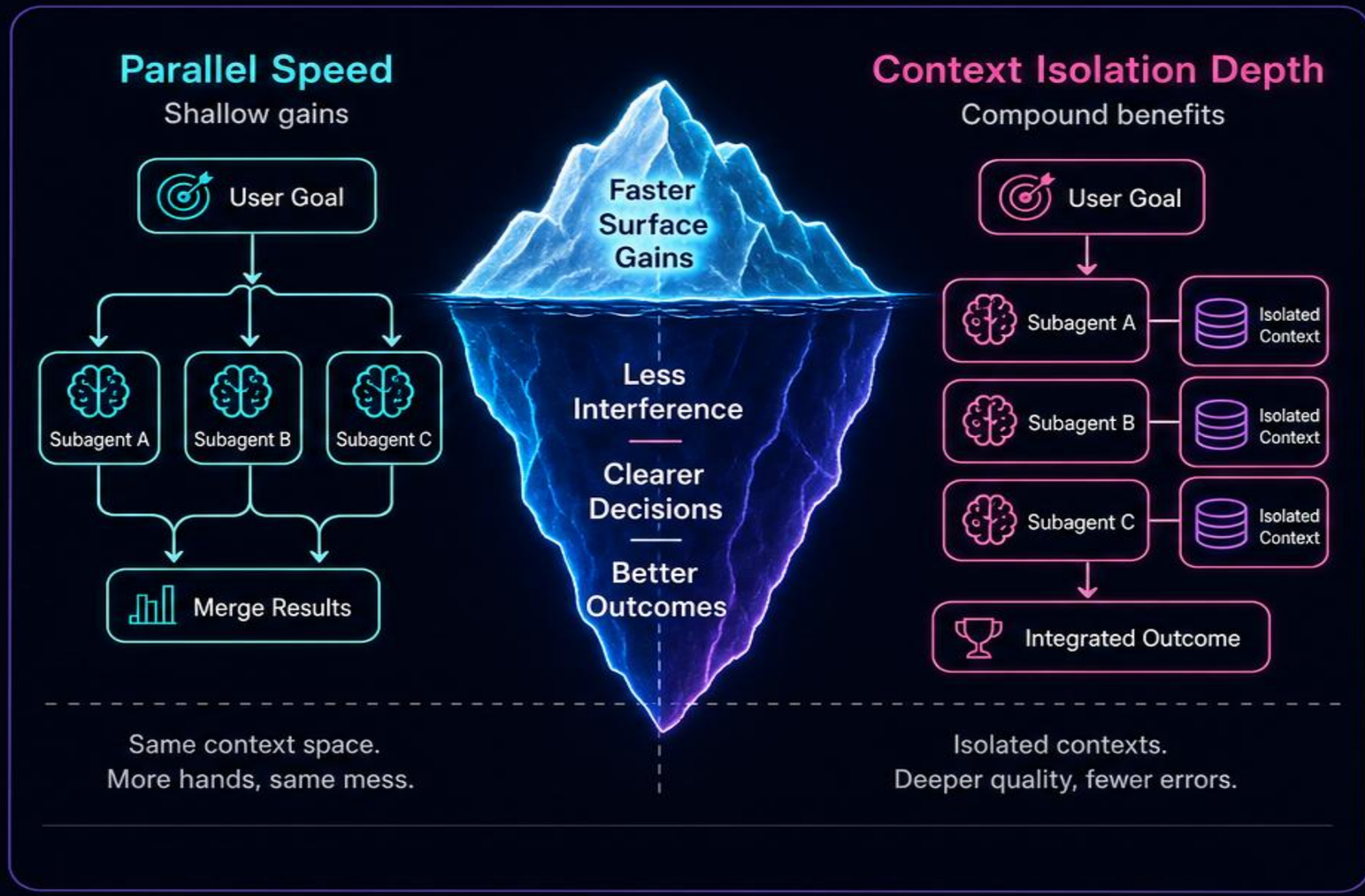
Running tasks in parallel helps, but only trims time – it doesn't solve the core problem.

### Context isolation is the deeper reason to delegate

Isolating context reduces interference, confusion, and costly mistakes.

### Even one subagent pays off by isolating context

A single subagent creates a clean workspace that improves quality and reliability.



# Isolation Is the Point



**Each subagent** runs fresh and inherits none of the manager's context



**Pass everything** the subtask needs into the delegated task



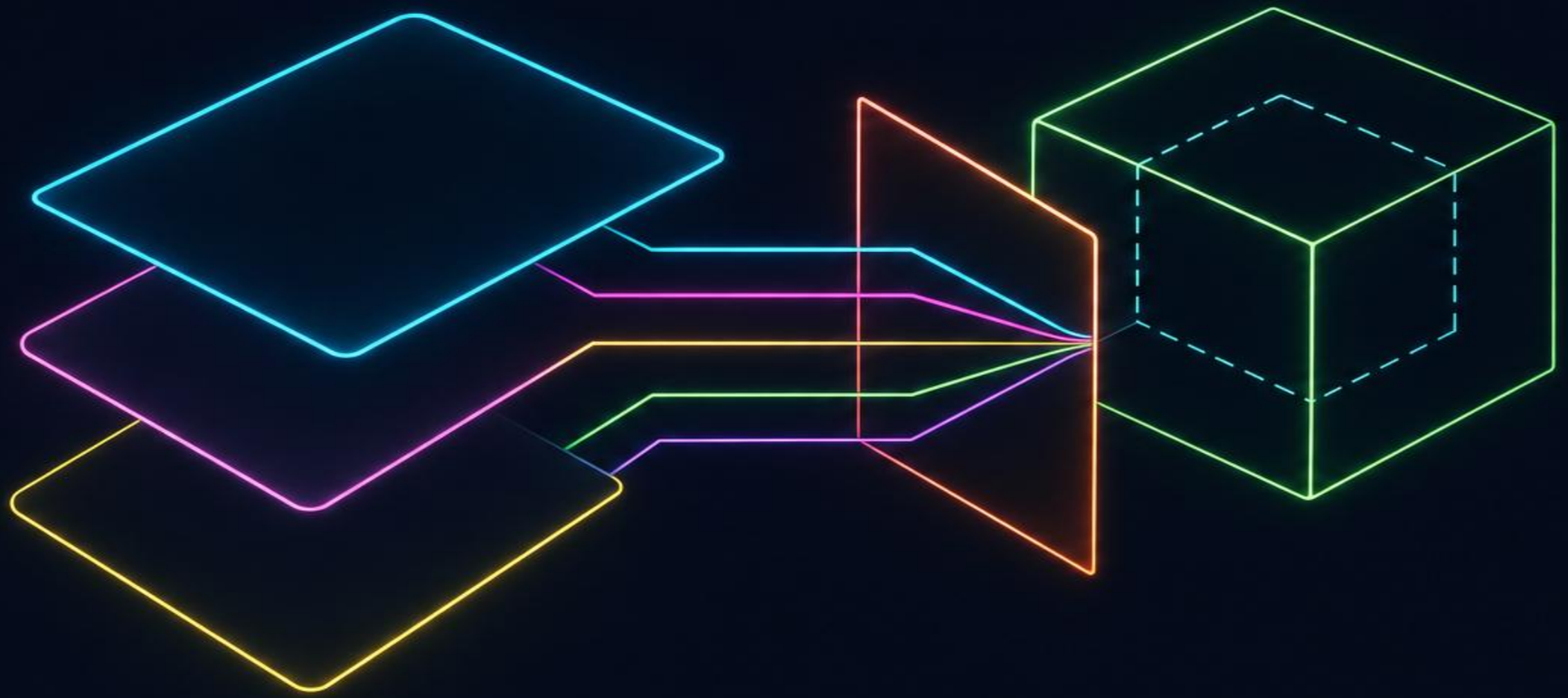
**Isolation bounds** token growth and contains authority



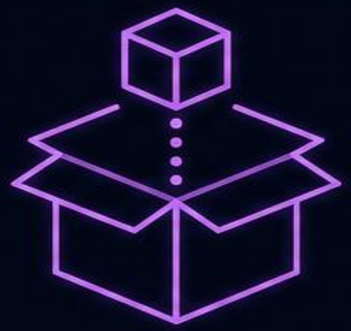
**Value context isolation** over raw parallel speed

# Your First Agent with query()

The Claude Agent SDK gives you the same agent loop
that powers Claude Code, right inside your own Python or TypeScript program.

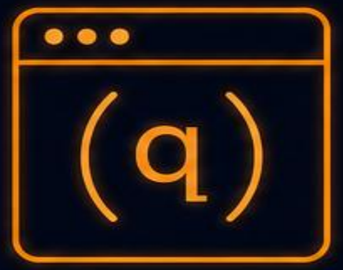


# The SDK Entry Point



## Agent SDK

The library packaging Claude Code's tools, agent loop, and context management for Python and TypeScript.




## query()

The primary entry point that runs the standalone agent loop. Each call starts a fresh session by default.



## No CLI needed

The SDK is standalone. You import it and call it directly, without installing the Claude Code command line tool.

# Installing and Authenticating

Before your first run, install the package and set your credentials.

**Python** installs with pip install claude-agent-sdk, needing Python 3.10 or newer

**TypeScript** pulls @anthropic-ai/claude-agent-sdk from npm

**Set ANTHROPIC_API_KEY**, or point at Bedrock or Vertex with env flags

```
$ Terminal

# Python
$ pip install claude-agent-sdk

# TypeScript
$ npm install @anthropic-ai/claude-agent-sdk

# Environment variables
$ export ANTHROPIC_API_KEY="sk-ant-************"
$ export ANTHROPIC_BASE_URL="https://api.anthropic.com"
$ export ANTHROPIC_MODEL="claude-opus-5"
```

# Two Ways In, One Loop.



## query()

A function you call to launch a run.
It opens a new session each time and does not support mid-run interrupts.



## ClaudeSDKClient

A Python object that reuses one session across calls and adds interrupts.
It runs the very same agent loop underneath.

# Configuring the Run

query() takes options that shape how the agent behaves on this run.

## Run Configuration

**system_prompt**
Sets the agent's role and standing instructions.

```
You are a research analyst.
Be accurate, cite sources,
and ask clarifying questions
when needed.
```

**tools**
List of tools the agent is allowed to use.

`WebSearch, Read, Bash, ...`

**model**
Model to use for reasoning and tools.

`claude-opus-5`

**max_turns**
Maximum agent turns before stopping.

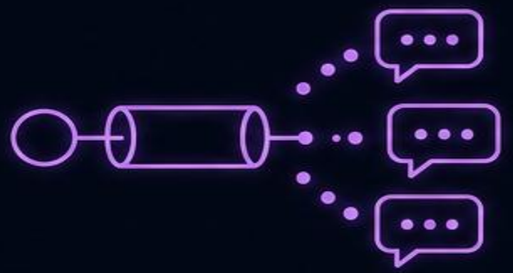
`− 12 +`

**A system prompt**
sets the agent's role and standing instructions

**A tools list**
decides which actions the model can take

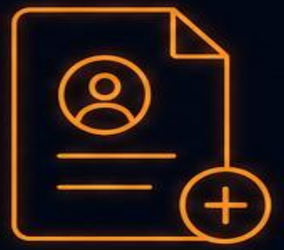
**A model choice**
and a max_turns cap bound cost and runtime

# What query() Returns



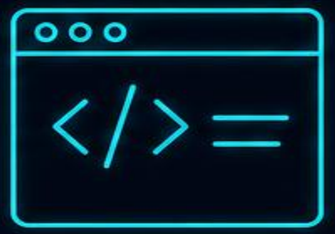
## An async iterator

query() returns a stream you loop over, reading each message as the agent produces it.



## A fresh session

Each call opens a new session, with no memory of any earlier call by default.



## No CLI process

The whole run happens in your process. There’s no command line tool running behind it.

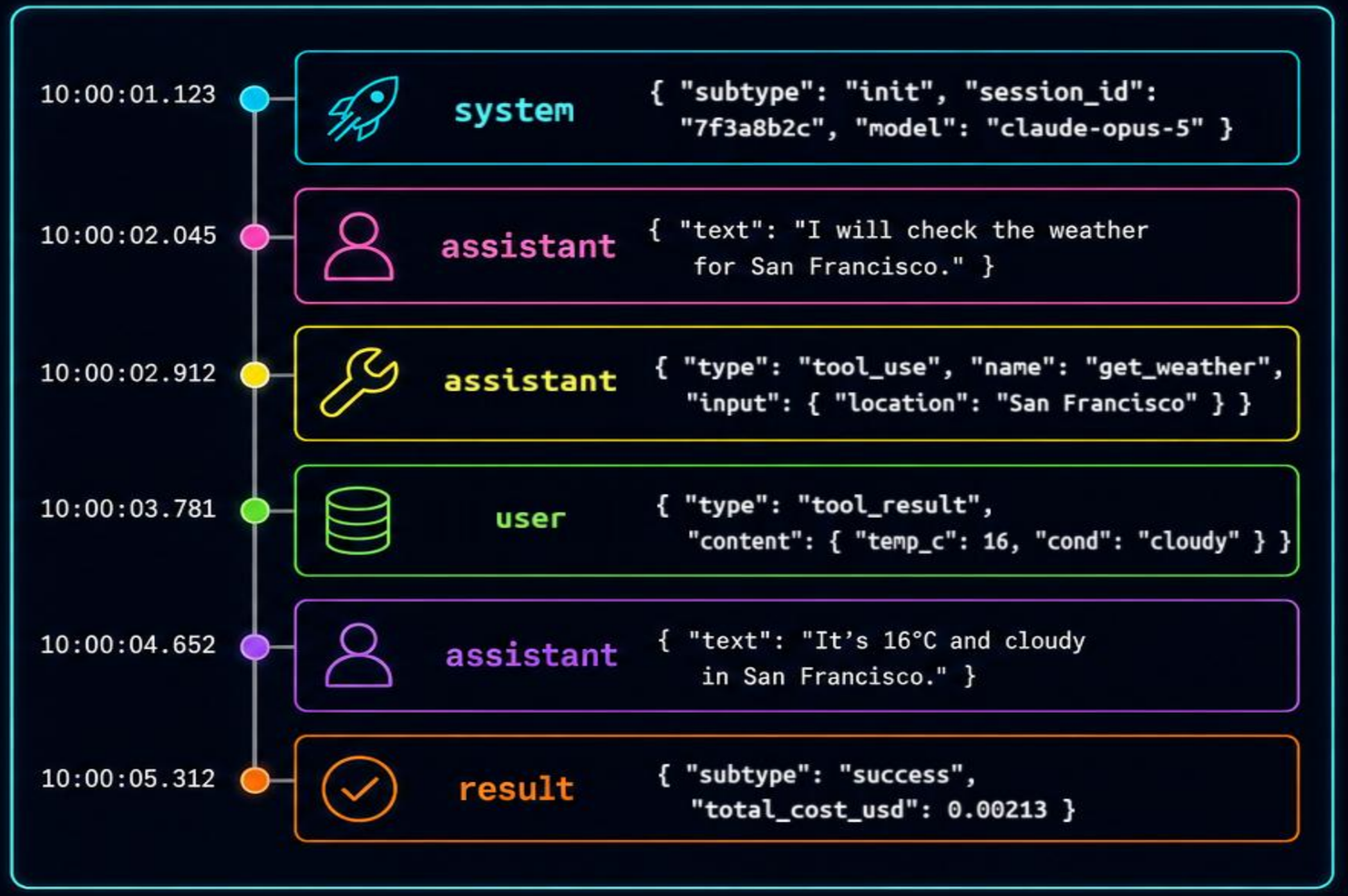
# Reading the Message Stream

As the loop runs, the stream yields a sequence of typed messages.

**An init message**
opens the run with the resolved configuration

**Assistant and tool messages**
show reasoning and results in order

**A final result message**
closes the run with cost and status



# Three Traits of a Run



## Standalone

The run needs no CLI install.
The library alone carries the full agent loop.



## Streamed

Messages arrive as an async iterator,
read live as the agent works, not in one block.

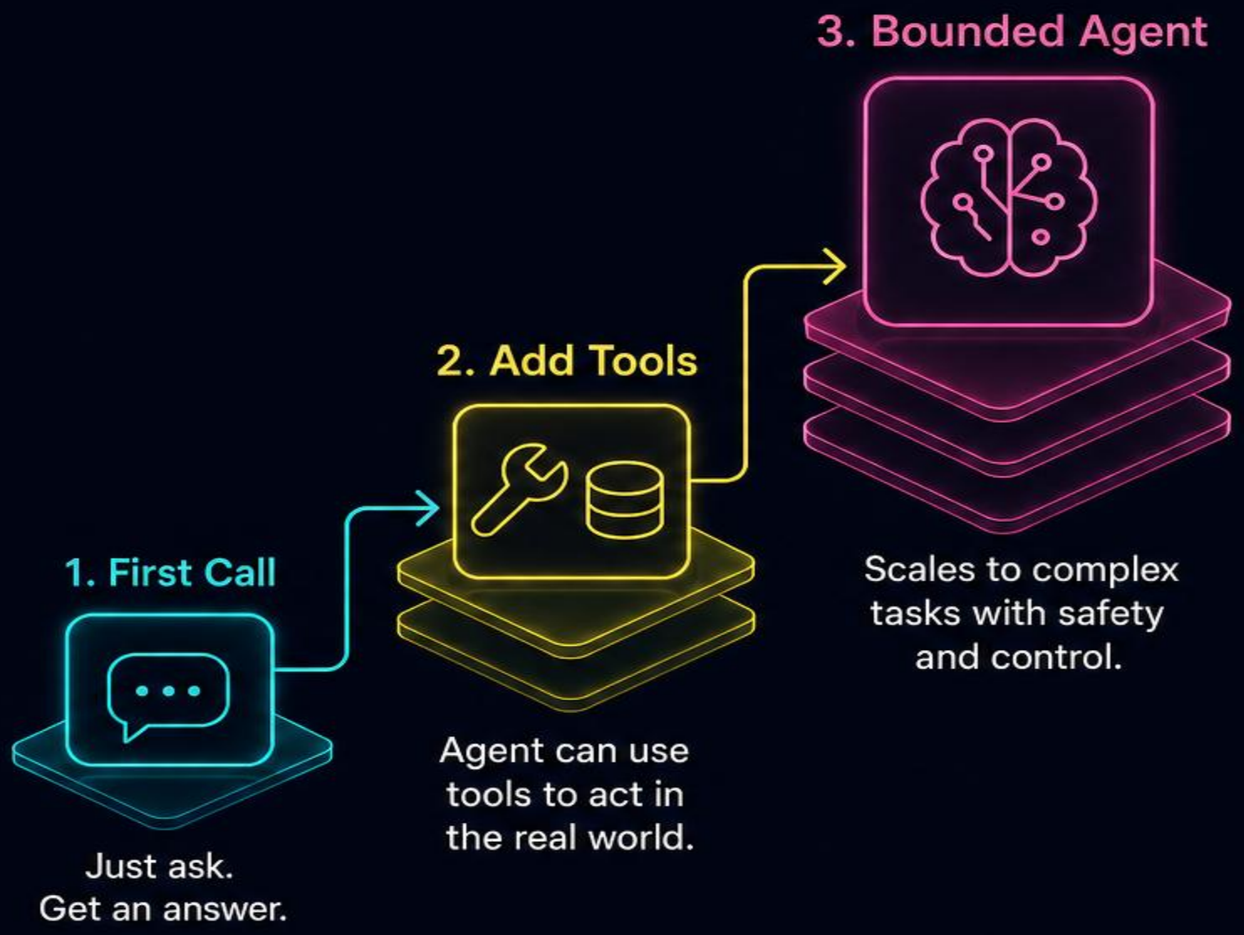


## Session-scoped

Each call is its own fresh session;
memory across calls is opt-in.

# From First Call to Real Agent

A first query() call is small, but it scales into real work.




**Start with a prompt**
and default options
to see the loop run

**Add tools**
so the agent can act,
not just answer

**Add a max_turns bound**
before you trust it
with longer tasks

# Your First Agent



**It runs** Claude Code's agent loop, no CLI installed



**Options** set the system prompt, tools, model, and turn cap



**It returns** an async stream and a fresh session per call



**query()** and the client share one loop, not two

# How the Agent Loop Runs

When you call query(), Claude doesn't answer once and stop.
It runs a loop: reasoning, calling tools, reading results, and going again.




Interpret the goal and choose the next step.

Use a tool or perform an action.

Read the result, check progress, and update state.

**Repeat until goal is met**

# Walking the Loop, Start to End

The loop turns a prompt into a finished answer in five steps.

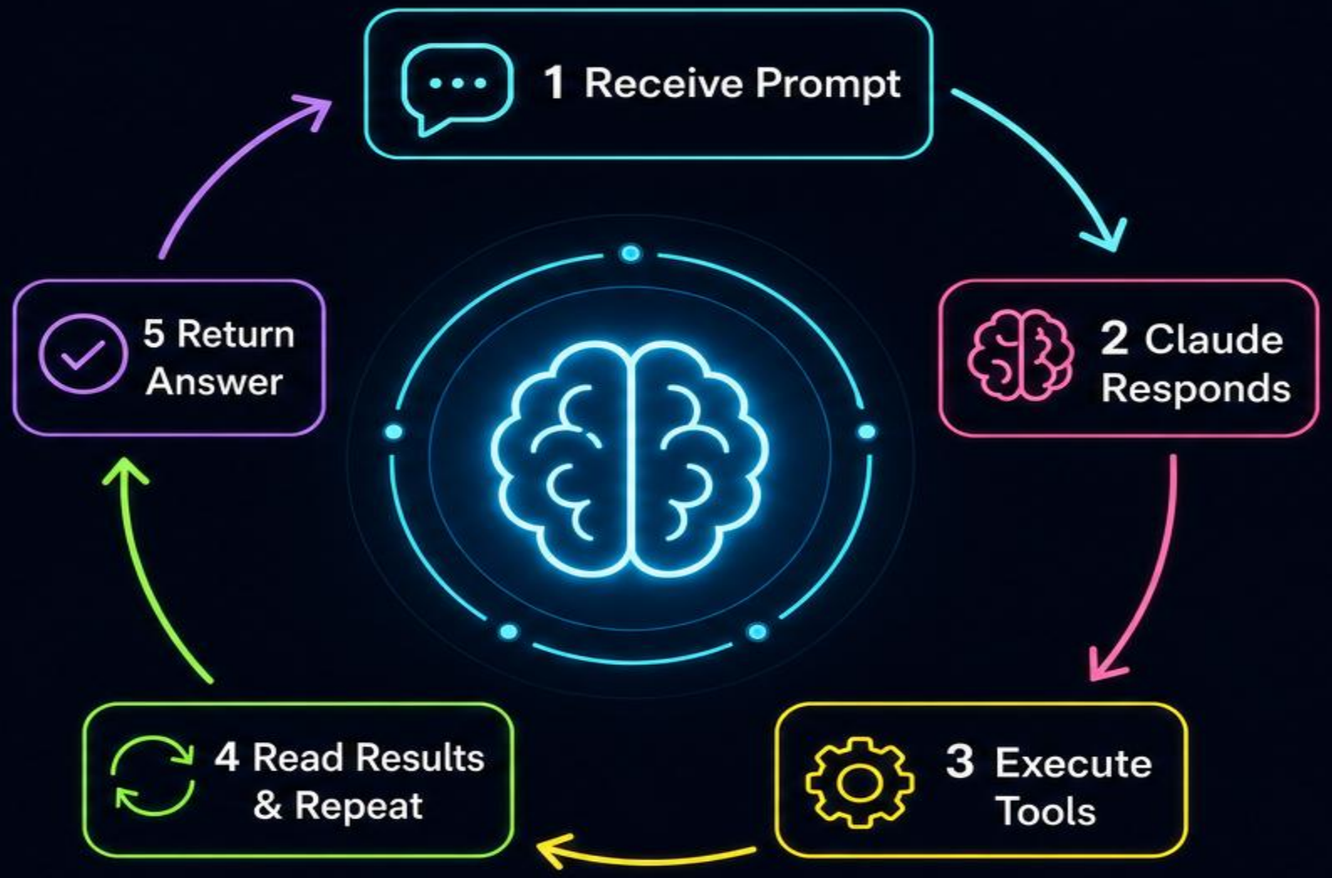
**Claude receives** the prompt, tools, and history, and an init message opens the run

**Claude responds** with text or tool calls

**The SDK executes** those tools and feeds results back

**Claude reads results** and the turn repeats if it calls more tools

**The loop returns** the answer once Claude calls no tool



# What a Turn Really Means



## A turn

One round trip: Claude requests tools,
the SDK runs them, and results feed back,
with no stop for your code.



## Turn boundary

A turn ends when its tool results return.
The next turn begins if Claude calls more tools.

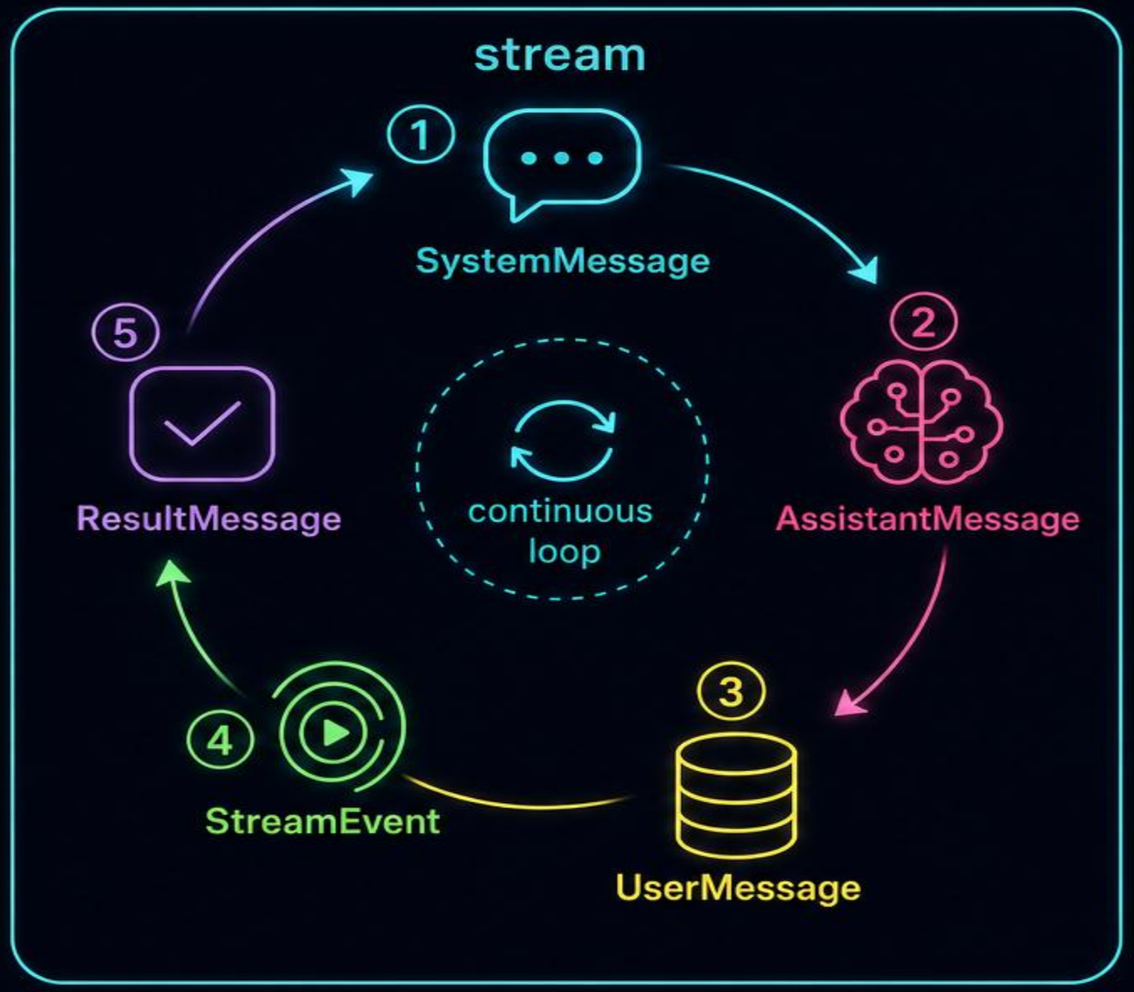


## Loop end

The loop stops when Claude replies
with no tool call, producing the final answer.

# The Five Message Types

The stream yields five typed messages, each with a clear job.




**SystemMessage** opens the run, including the init subtype

**AssistantMessage** carries Claude’s text and tool calls

**UserMessage** delivers tool results back into the loop

**StreamEvent** appears only when partial streaming is on

**ResultMessage** closes the run with cost and status

# Bounding the Loop



## max_turns

Caps tool-using turns. Hitting it stops the loop and returns an error result.



## max_budget_usd

Caps spend in dollars. Checked between calls, so a run can slightly overshoot before it halts with a budget error.



## Default is no cap

Left unset, a run has no turn or budget limit at all.



## Bounds are yours

Setting a sensible limit on autonomous runs is the developer's job, not the SDK's.

# What the Result Message Carries

When the run ends, the result message reports how it went.

### A subtype says success or which limit ended the run

Tells the difference between a normal completion and a stop caused by a limit or safety rule.

### total_cost_usd and usage record what the run consumed

Includes cost and detailed token usage so you can budget, forecast, and optimize.

### num_turns and session_id let you audit and resume

Provide traceability and continuity across sessions for debugging, reporting, and recovery.

## Example Result Message

| Status | Subtype | Finished At |
|---|---|---|
| SUCCESS<br>completed | success | 2025-05-18T14:32:10Z |

### Cost & Usage




| | |
|---|---|
| input_tokens | 21,842 |
| output_tokens | 12,374 |
| cache_read_input_tokens | 3,842 |

total_cost_usd
$0.0738

total_tokens
38,058

### Session & Turns

num_turns
6

session_id
sess_9f7e2b0c8a1d4f22
b3d9e7c1a6f5b2d8

### Other Details

| duration_ms | model | tools_used |
|---|---|---|
| 14,627 | claude-opus-5 | 3 |

# Success vs Error Results



## Success result

Subtype success. It carries the final result text, plus the total cost, the token usage, and the turn counts for the run.



## Error result

Subtypes like error_max_turns or error_max_budget_usd. There’s no result text, but you still get full cost and usage accounting for the partial run.

# When a Run Hits Its Limit

Hitting a cap does more than return an error subtype.






**A single-shot call yields the error result, then raises an error**

Your code sees the error result, and execution stops unless you handle it.



**Wrap it in try and except if you want to keep going**

Catch the exception and handle it so the rest of your program can continue.



**A streaming session stays alive instead of raising**

You receive an error event in the stream and can choose how to proceed.

So the input mode changes how a limit reaches your code.

# Reading a Stopped Run



## stop_reason

Why the model stopped this turn, such as end_turn, max_tokens, or a refusal.



## num_turns

How many turns the run took, useful for spotting loops that ran long.




## session_id

Identifies the whole run so you can resume it or trace it later, after it ends.

# How the Loop Runs



**Claude reasons,** calls tools, reads results, and repeats



**One turn** is a single tool round trip



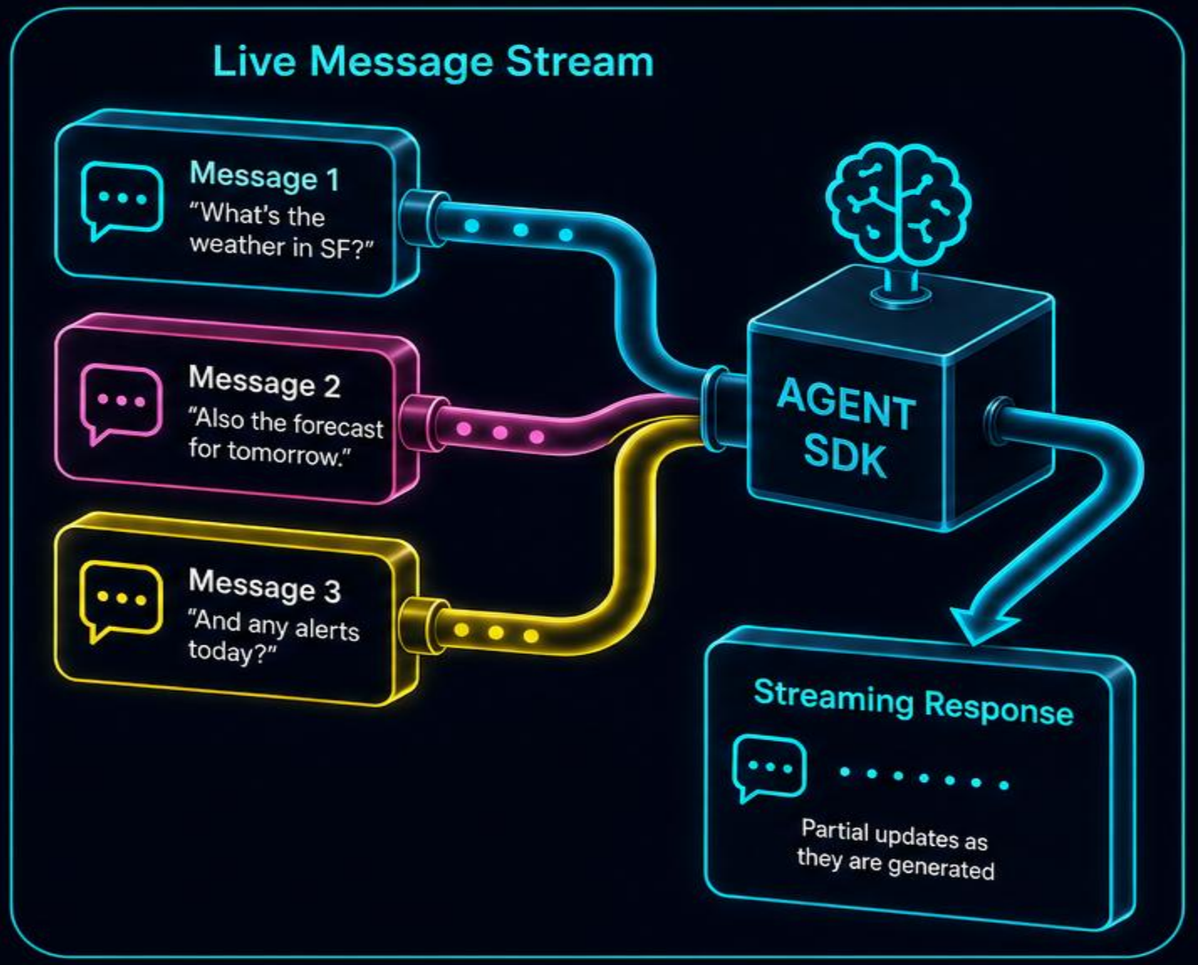
**The loop ends** on a tool-free answer or a cap

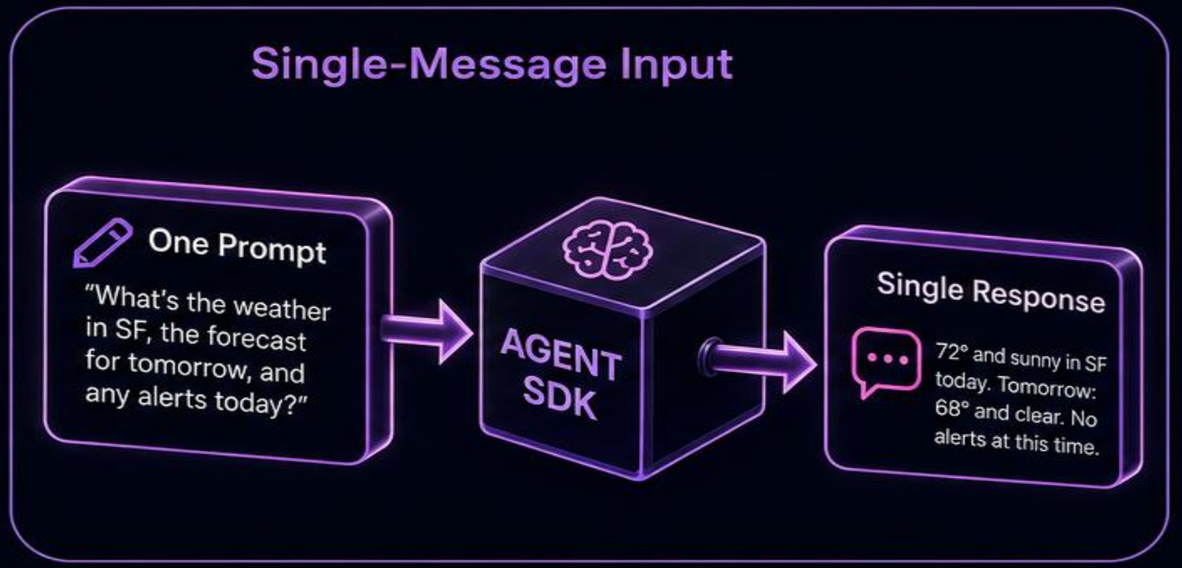


**The result message** reports subtype, cost, and usage

# Streaming vs Single-Message Input

The Agent SDK takes your prompt two ways. You can stream a live series of messages, or send one string and get one answer.



# The Two Input Modes



## Streaming input

The prompt is an async series of messages. The agent runs as a persistent, long-lived process. This is the default.



## Single-message input

The prompt is one string, answered once. Best for one-off or stateless jobs like a serverless function that spins up per request.

# What Streaming Input Unlocks

Running as a live process gives streaming input real powers.

**Attach images**
directly in the messages you send

**Queue new messages**
while a turn is still running

**Interrupt a run**
in real time, via the client object

**Hold multi-turn context naturally**
across the conversation




These are features single-message input can’t offer.

# Single-Message Trade-offs



## One-shot string

The prompt is a single string and the agent answers once, with no live session held open.



## Stateless fit

Ideal for serverless or batch jobs that spin up, answer, and shut down per request.



## continue and resume

Multi-turn is still possible, but only through the continue and resume session options.

# Choosing a Mode

Match the mode to the shape of your workload.




**Streaming Input**

Long conversations with tools and state.

**Single-Message**

One request, one response.

**Streaming (Live Input)**

Real-time visuals and interrupts.

**Reach for streaming input**

for interactive, tool-heavy, multi-turn agents

**Reach for single-message**

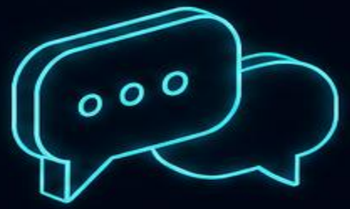
for one-off calls and stateless functions

**Need images or live interrupts?**

That points to streaming

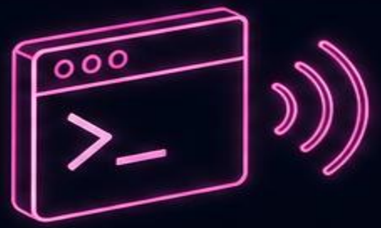
When unsure, the default streaming mode is the safe starting point.

# Clearing Up the Confusion



## Not two loops

Streaming and single-message are input modes into one shared agent loop, not separate engines.




## query() streams too

Streaming input does not require the client. query() accepts an async prompt, though interrupts still need the client.




## Input is not output

SDK streaming input differs from API response streaming, the stream true flag.



## Partial output streaming

Turn on include_partial_messages to receive StreamEvent deltas, token by token.

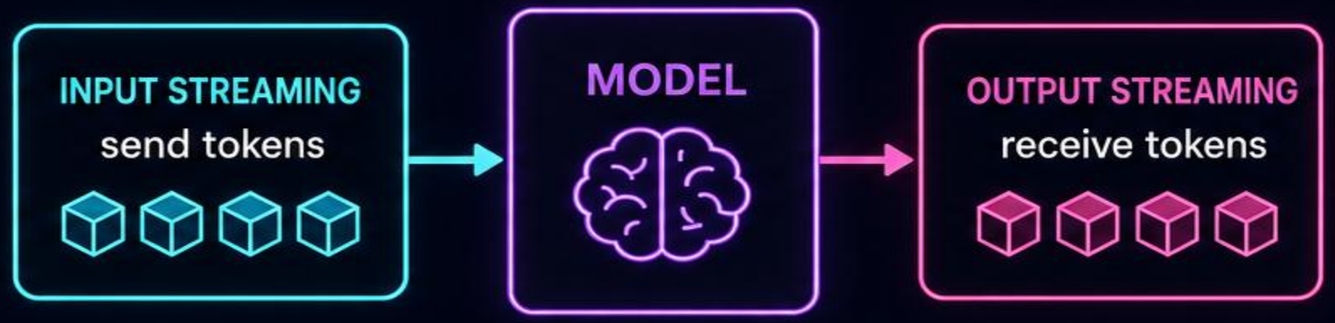
# Input Streaming vs Output Streaming

Two different ideas share the word streaming, so keep them apart.

**Input streaming** is how you feed prompts into the loop

**Output streaming** is receiving the model's tokens as they generate

**Turn on** `include_partial_messages` to get those live deltas

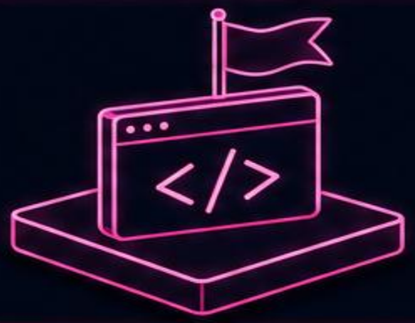


# Three Streaming Terms



## Streaming input

Feeding the loop an async series of messages, the default and recommended input mode.



## include_partial_messages

The Python flag that turns on live output deltas, delivered as StreamEvent messages.

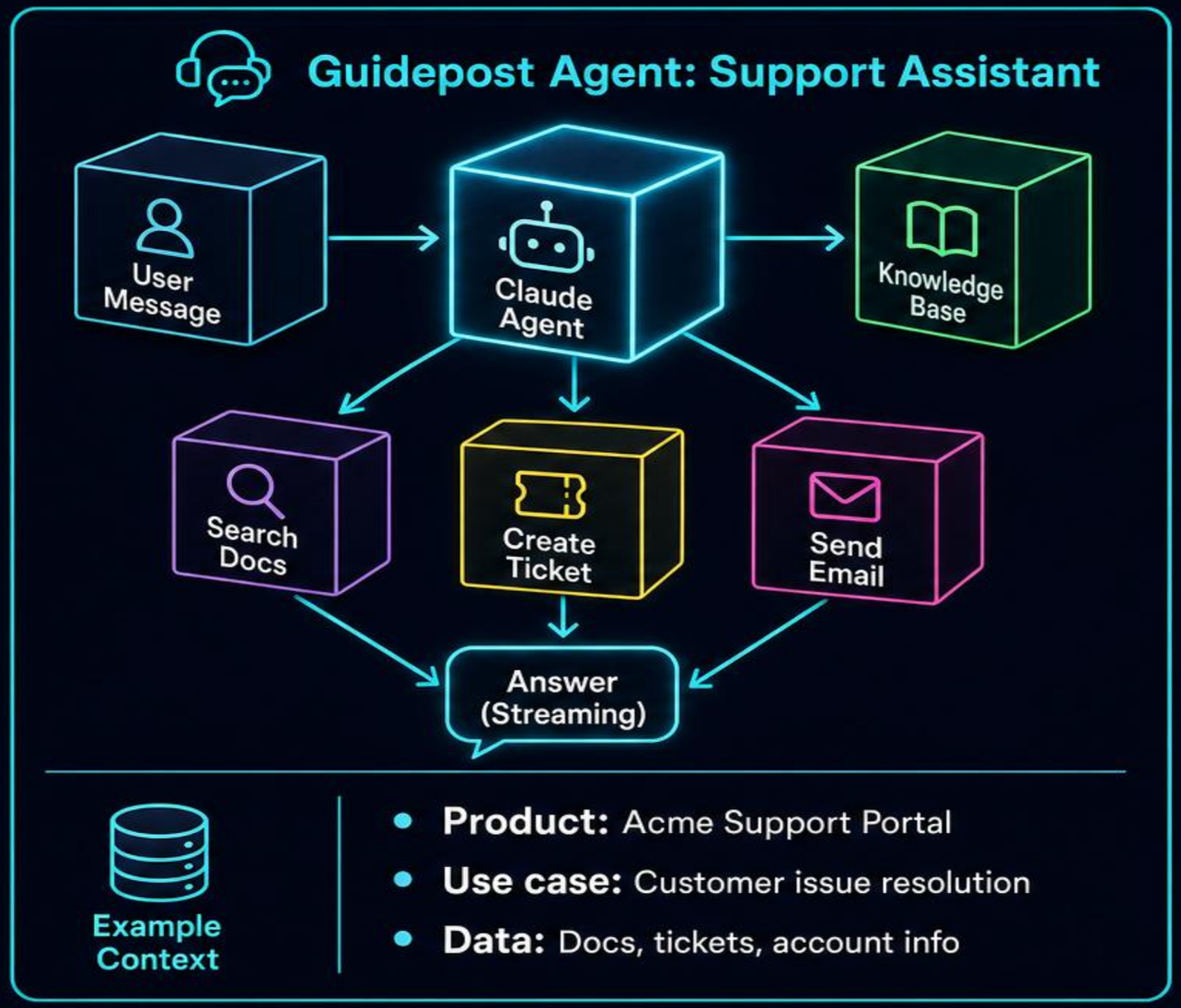


## stream true

The API-level response flag on a raw model call, a separate mechanism from SDK input streaming.

# A Practical Default

For most agents you build, one starting choice is safe.




## Begin with streaming input, the recommended mode

Streaming input handles long messages without waiting for the whole prompt.

## Drop to single-message only for stateless, one-off jobs

Use single-message requests for simple webhooks, cron tasks, or integrations that send one prompt.

## Add output streaming when users watch responses appear

Streaming output improves perceived speed and keeps users engaged during longer responses.

# Input Modes at a Glance



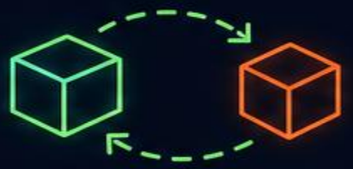
**Streaming input is the default:** images, interrupts, multi-turn



**Single-message input** is one string, best when stateless



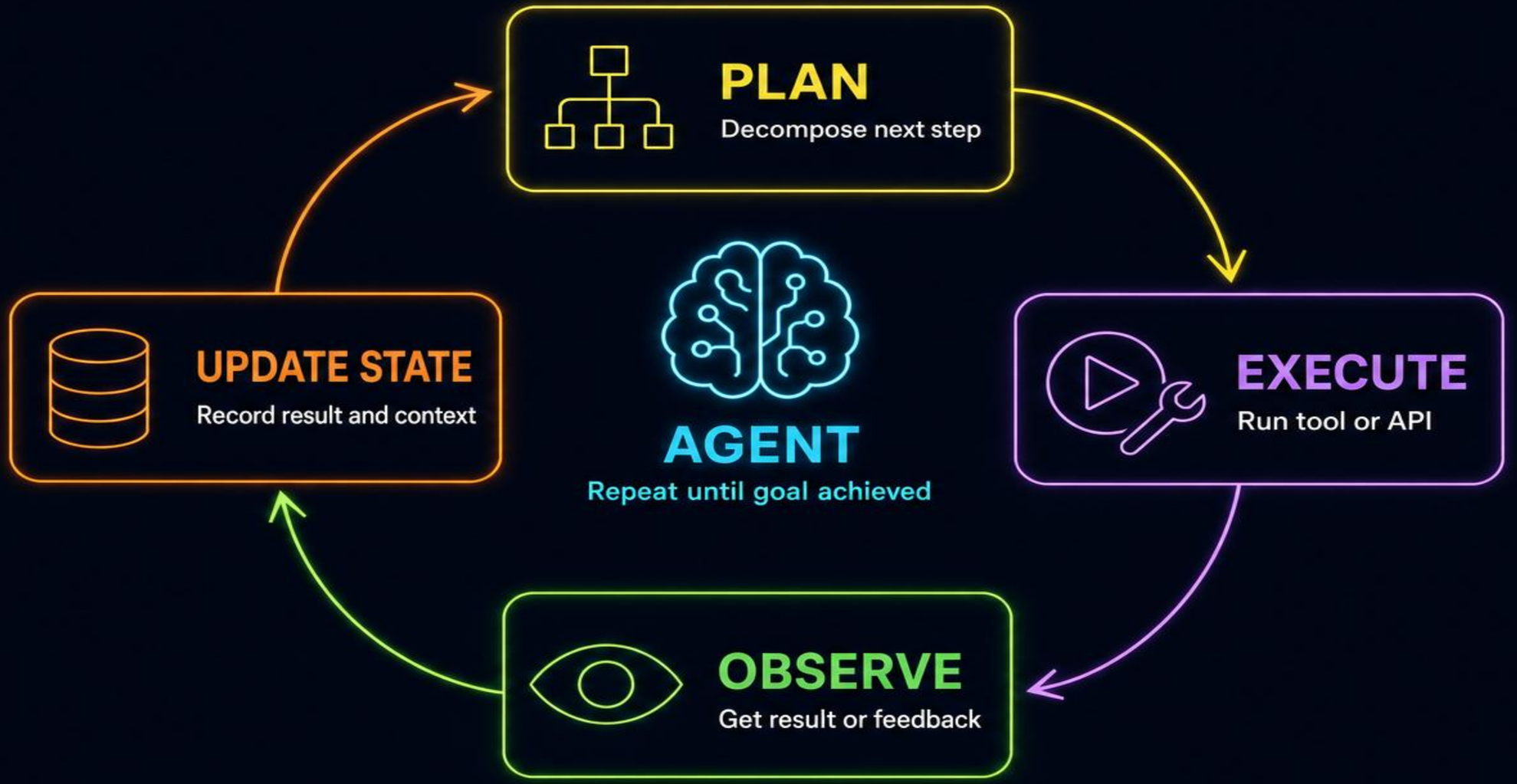
**`query()`** supports streaming input, no client required



**Input streaming and output streaming** are separate ideas

# Building a Custom Agent Loop

Sometimes the managed loop hides too much and you want to own every turn.



# Three Levels of Control

Choose the right abstraction for your task, risk, and required control.



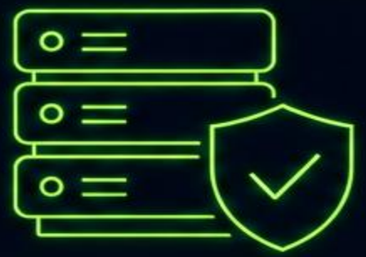
## Messages API

The raw model endpoint.
You inspect stop_reason, run tools yourself, and re-invoke.
Best for custom loops.



## Agent SDK

Claude Code packaged as a library.
It runs the loop and executes tools for you.

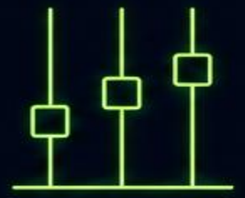


## Managed Agents

A fully hosted harness.
Anthropic runs the loop and sandbox on their infrastructure.

# Why Drop Below the SDK

You reach for a custom loop when you need total control over each turn.



### You own tool execution, retries, and logging

You decide when a tool runs, how to retry on failure, and what gets logged.



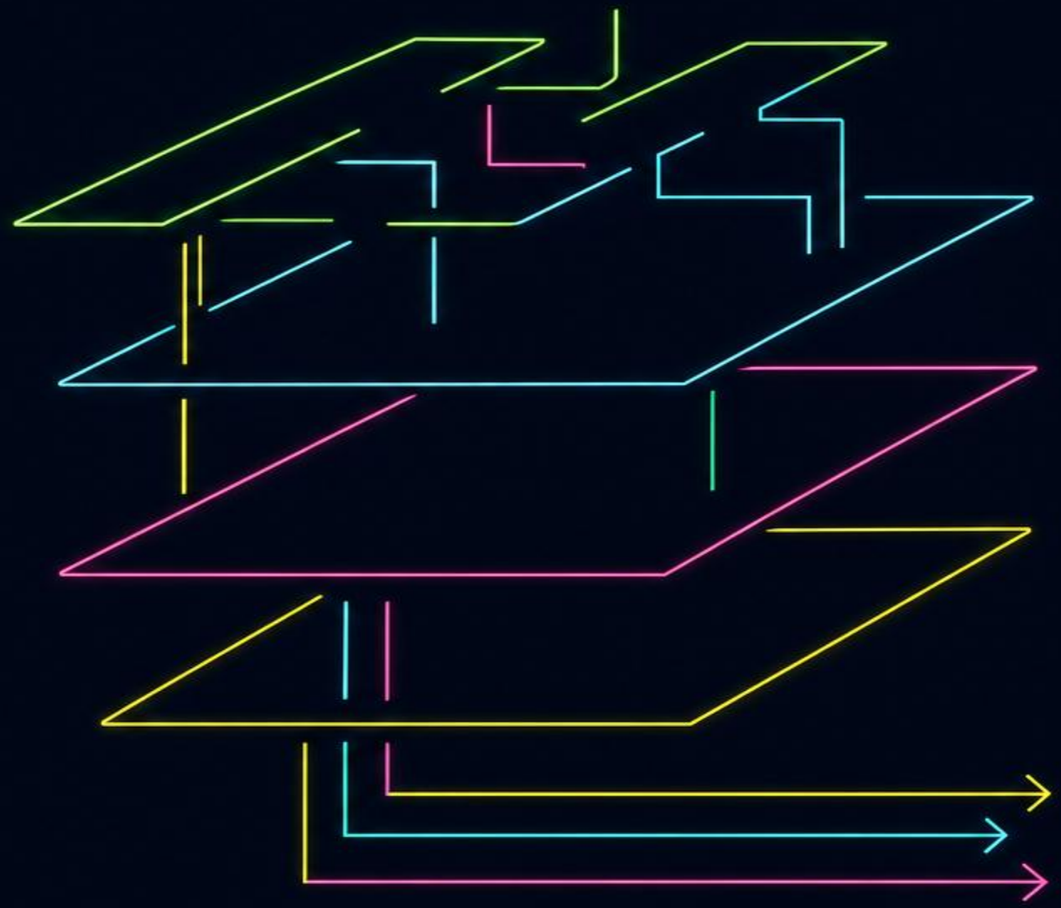
### Fewer dependencies, no framework to track

No hidden layers or magic - just your code and the SDK.



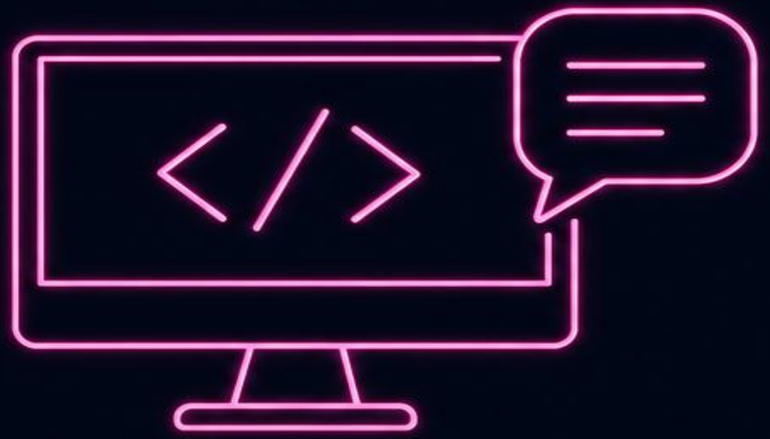
### Every request and response passes through your code

Inspect, transform, and validate everything in and out.



# Client SDK vs Agent SDK

## Messages API (Client SDK)



You write the tool loop.

Call the API, inspect stop_reason, execute tool_use blocks, append tool_result, and re-invoke.

Maximum control, maximum code.

## Agent SDK



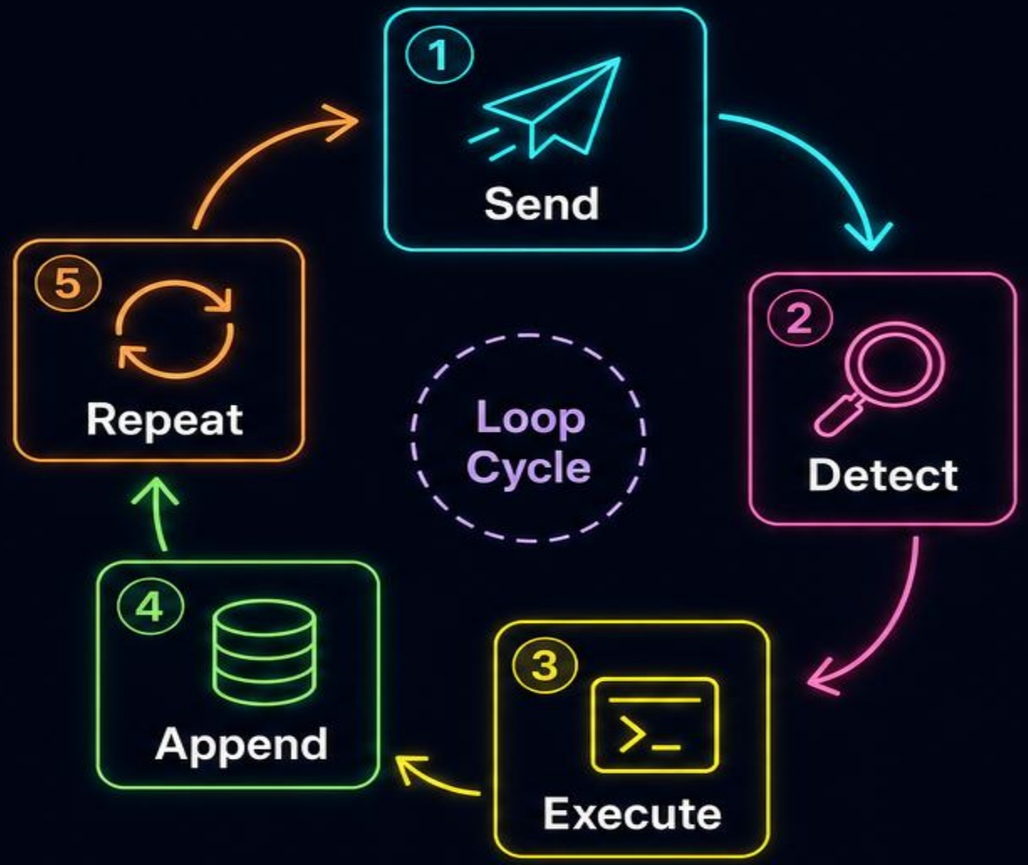
Claude handles the loop.

You configure tools, model, and limits instead of writing the reason, act, observe cycle.

Less control, far less code.

# The Loop's Core Mechanics

A custom loop repeats a fixed cycle until the model stops asking for tools.




### Send the messages array to the model

The current messages array is passed to the model for the next turn.

### Detect any tool_use blocks in the reply

If the model's reply contains tool_use blocks, extract them for execution.

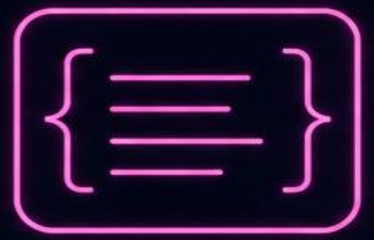
### Execute each tool and capture its result

Call each requested tool with its input and capture the tool_result for each call.

### Append a tool_result block, then re-invoke

Append each result as a tool_result block to the messages, then invoke the model again.

# Three Blocks That Drive the Loop



## tool_use

A block in Claude's reply naming a tool and its input. It signals your code to act.



## tool_result

The block you append with the tool's output, tied to its `tool_use` by id.




## stop_reason

The field you inspect each turn. A value of `tool_use` means Claude paused for you.

# Reading stop_reason Each Turn

After every model call, your loop branches on the stop_reason field.

### tool_use means run the tool and continue

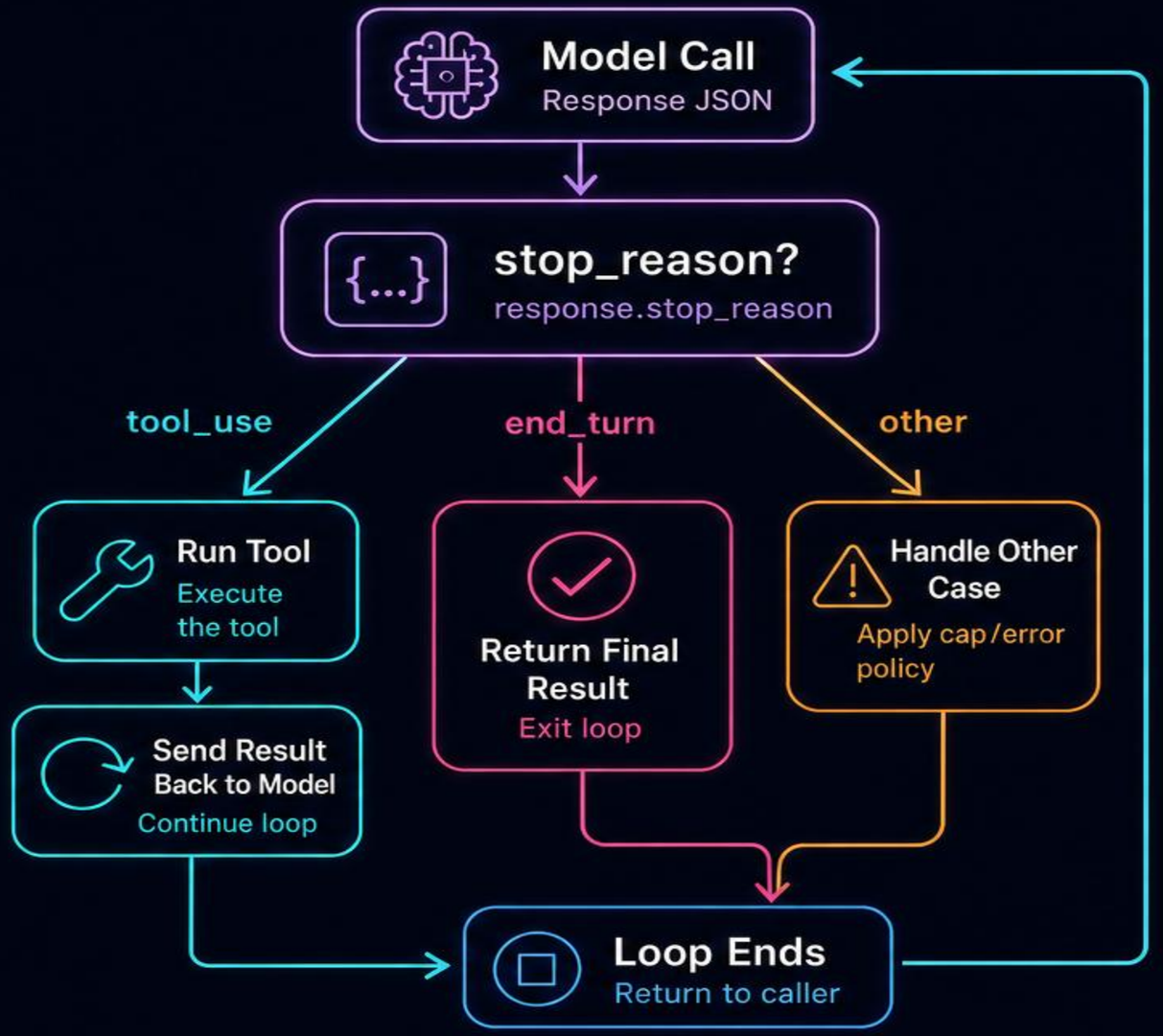
The model requested a tool. Execute it, send the result back, and keep looping.

### end_turn means Claude is done, exit the loop

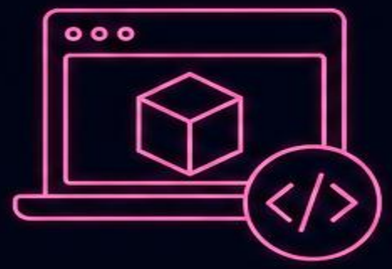
The model produced a final answer or has nothing more to do. Return the result and stop.

### Other values signal a cap or an error

Handle limits, safety stops, or unexpected states according to your policy.



# Who Runs the Tool



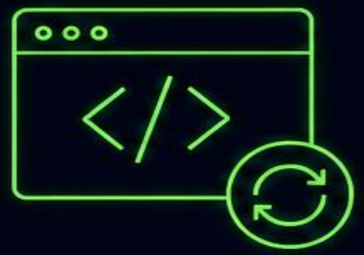
## Client tools

They run inside your app. Claude returns stop_reason tool_use, and you execute the tool and return a tool_result.



## Server tools

These run on Anthropic infrastructure. Their results come back to you directly, with no tool_result needed.



## Your job

In a custom loop you execute every client tool and feed each result back into the messages array.

# When a Custom Loop Earns Its Keep

Not every app needs a hand-rolled loop, but some genuinely benefit.

| | |
|---|---|
| **You need deterministic policy at each step** | Your app must make precisely defined decisions at every step, with no hidden abstractions or surprises. |
| **You are integrating into an existing runtime** | Your system already has its own execution model, state, and observability that a library loop would fight or hide. |
| **You want minimal dependencies in production** | You need the smallest possible footprint for reliability, compliance, and easier change control. |

# Owning the Agent Loop



**Messages API** is best for custom loops and fine control



**Cycle:** send, detect tool_use, execute, append tool_result, re-invoke



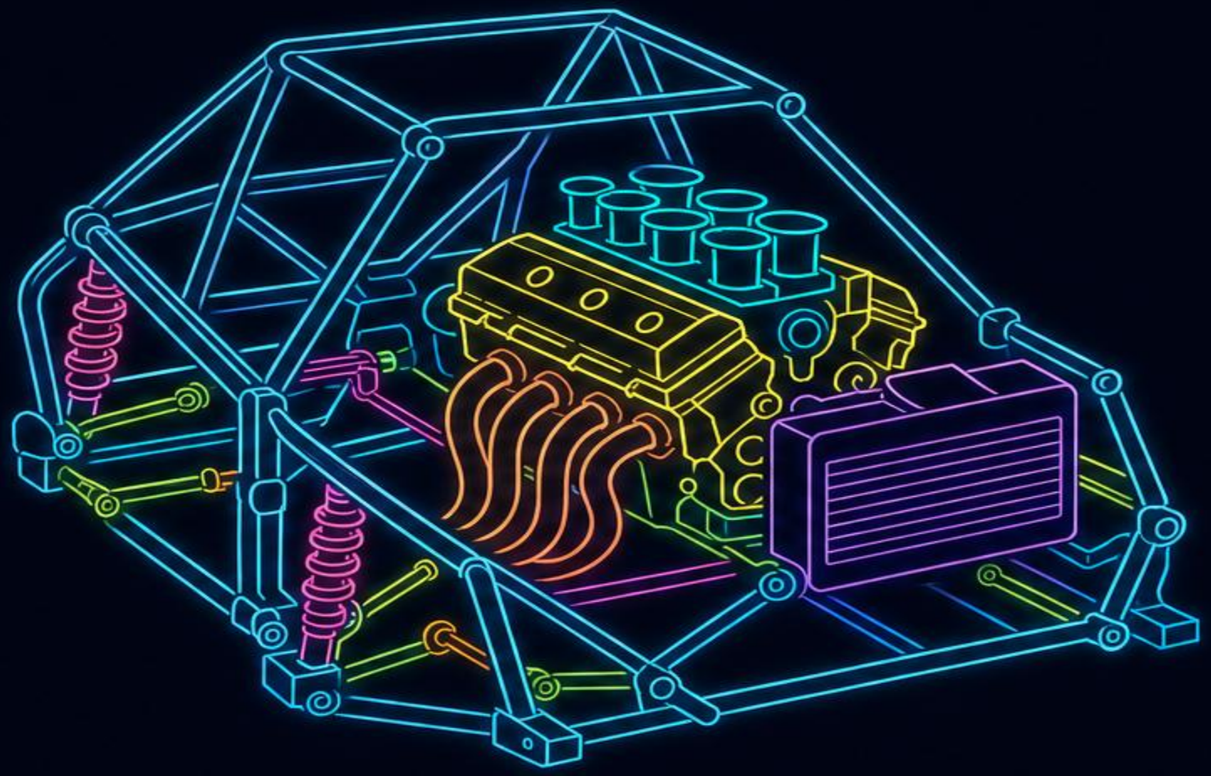
**stop_reason tool_use** means your code runs the tool

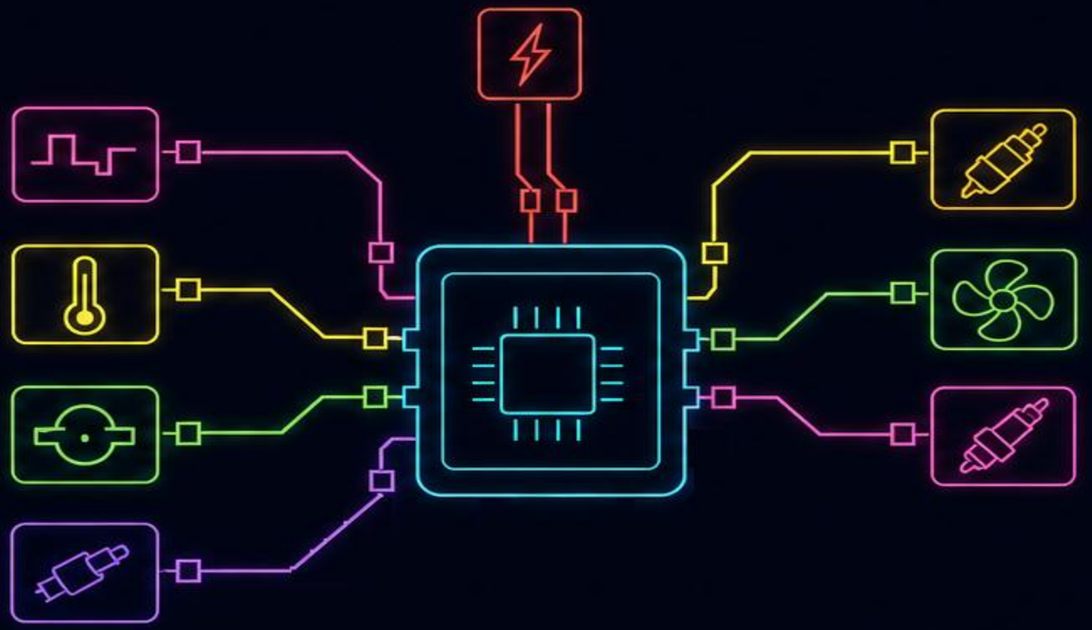


**Client tools** run in your app, server tools on Anthropic infra

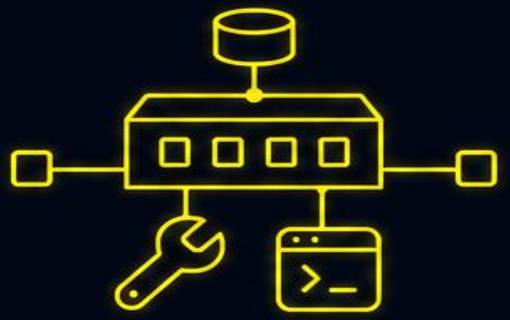
# Harness Design: Turns, State, Control

A raw loop is only the start. Around it sits the harness,
the infrastructure that keeps the loop safe and observable.



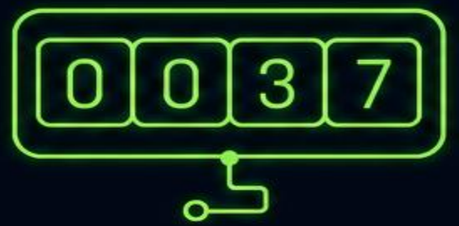


# Three Terms for the Harness



## Harness

The loop plus the infrastructure around it. It gathers context, sends requests, runs tools, and feeds results back.




## Turn accounting

Counting each tool-use round trip so the runtime, not the model, enforces a budget.

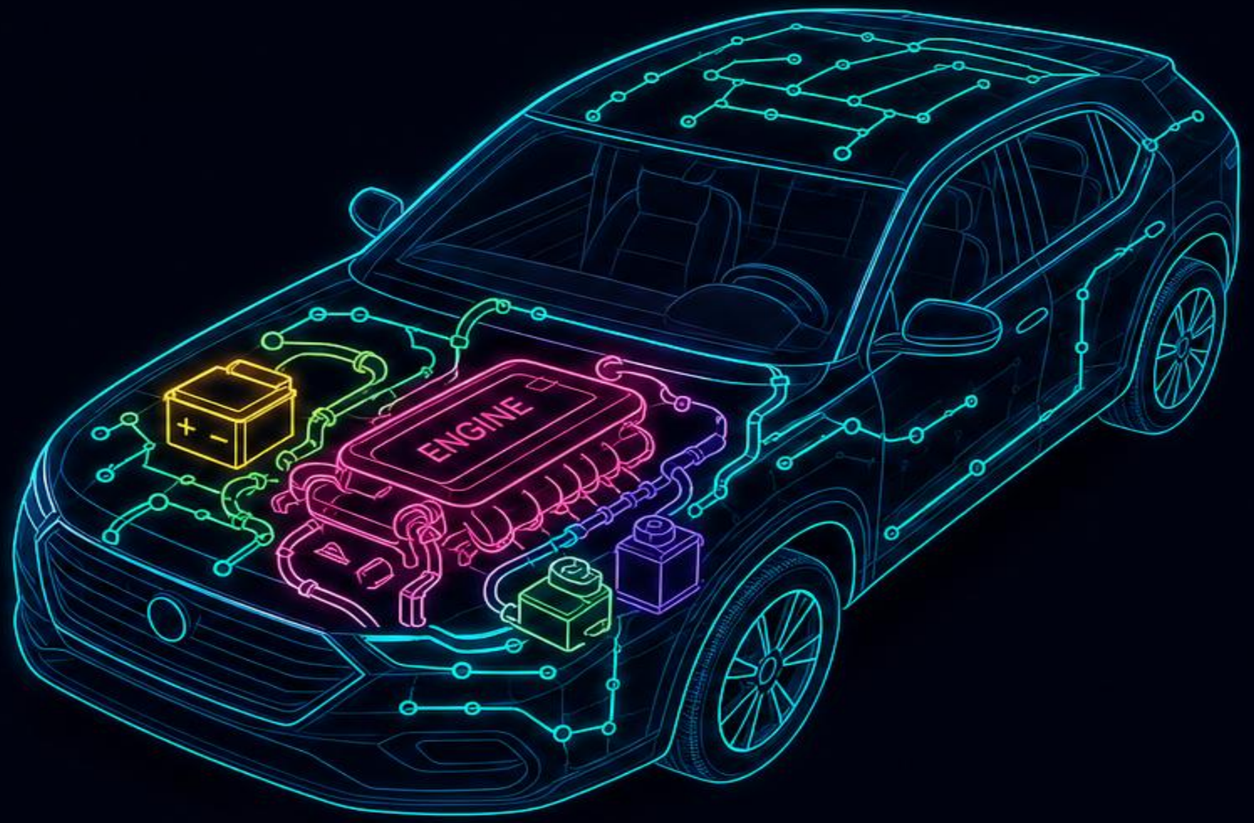


## Stop condition

An explicit rule that ends the loop: task done, turn cap hit, or an error.

# What the Harness Wraps

The harness is everything around the model call, not the call itself.

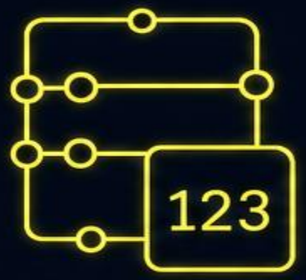



**It gathers and formats**
the context you send

**It executes tools**
and captures their output

**It records what happened**
for logging and audit

# Four Harness Responsibilities




## Turn accounting

Count every tool-use round trip and stop when the budget is reached.



## State tracking

Hold the growing messages array and any data the loop carries between turns.



## Error handling

Catch tool failures and model errors, then decide retry, skip, or abort.



## Stop conditions

Define exactly when the loop ends so it never runs unbounded.

# Enforce Turn Budgets in Code

An agent loop needs a hard limit that lives in your runtime, not the model's goodwill.

**Count tool-use turns as they happen**

Track every assistant turn that results in a tool call, not just messages.

**Stop the loop the moment the cap is hit**

Enforce a hard ceiling in your runtime and return control to your application.

**Never trust the model to self-limit**

Models optimize to complete tasks, not to obey budgets. Your code must enforce the guardrails.

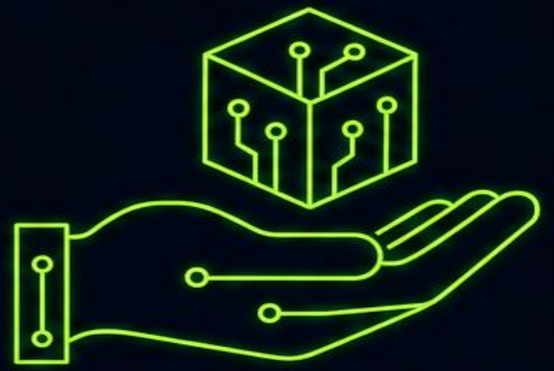
**An unbounded loop is a defect, not a feature you left in.**

# Hand-Rolled vs the Agent SDK



## Hand-rolled harness

You write turn accounting, state, and stop logic yourself. More control, fewer dependencies, and more code you own and maintain.

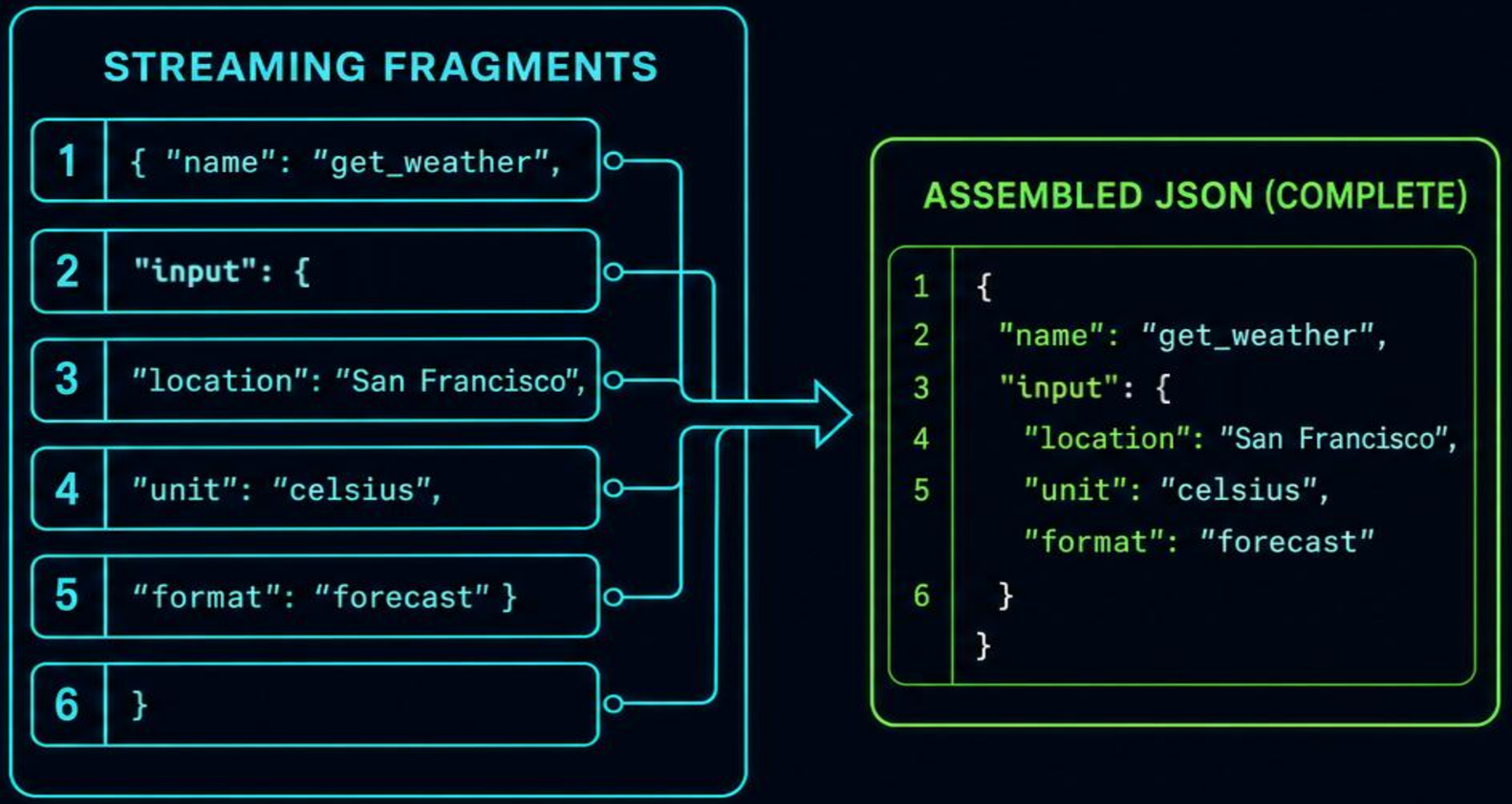


## Agent SDK harness

The loop, tool execution, context management, and limits come built in. Less code, but you accept the framework's shape and updates.

# Streaming and Partial JSON

When you stream a response, a tool's arguments arrive in pieces, not all at once.




### Deltas of type input_json_delta carry partial JSON

The stream delivers fragments in order. Each delta holds a slice of the overall JSON.

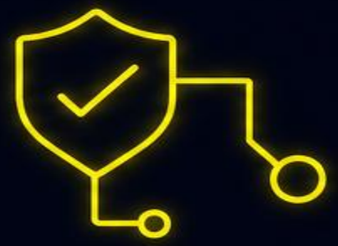
### You accumulate the fragments as they arrive

Append each fragment to a buffer. Do not parse early or you risk invalid JSON.

### You parse only when the block closes

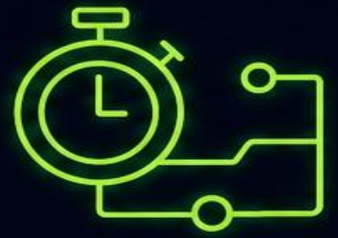
When the final closing brace arrives, the JSON is complete and safe to parse and use.

# Three Points to Inject Policy



## Before the tool runs

Your loop can check, log, or block a call before executing it. Deterministic policy lives here.



## Around execution

Wrap tool calls with retries and timeouts so a transient failure doesn’t kill the run.



## After the result

Log outputs and record turn counts before feeding results back to the model.

# The Control You Own

Owning the loop means the reliability levers are yours to pull.

### Inject deterministic policy at each step

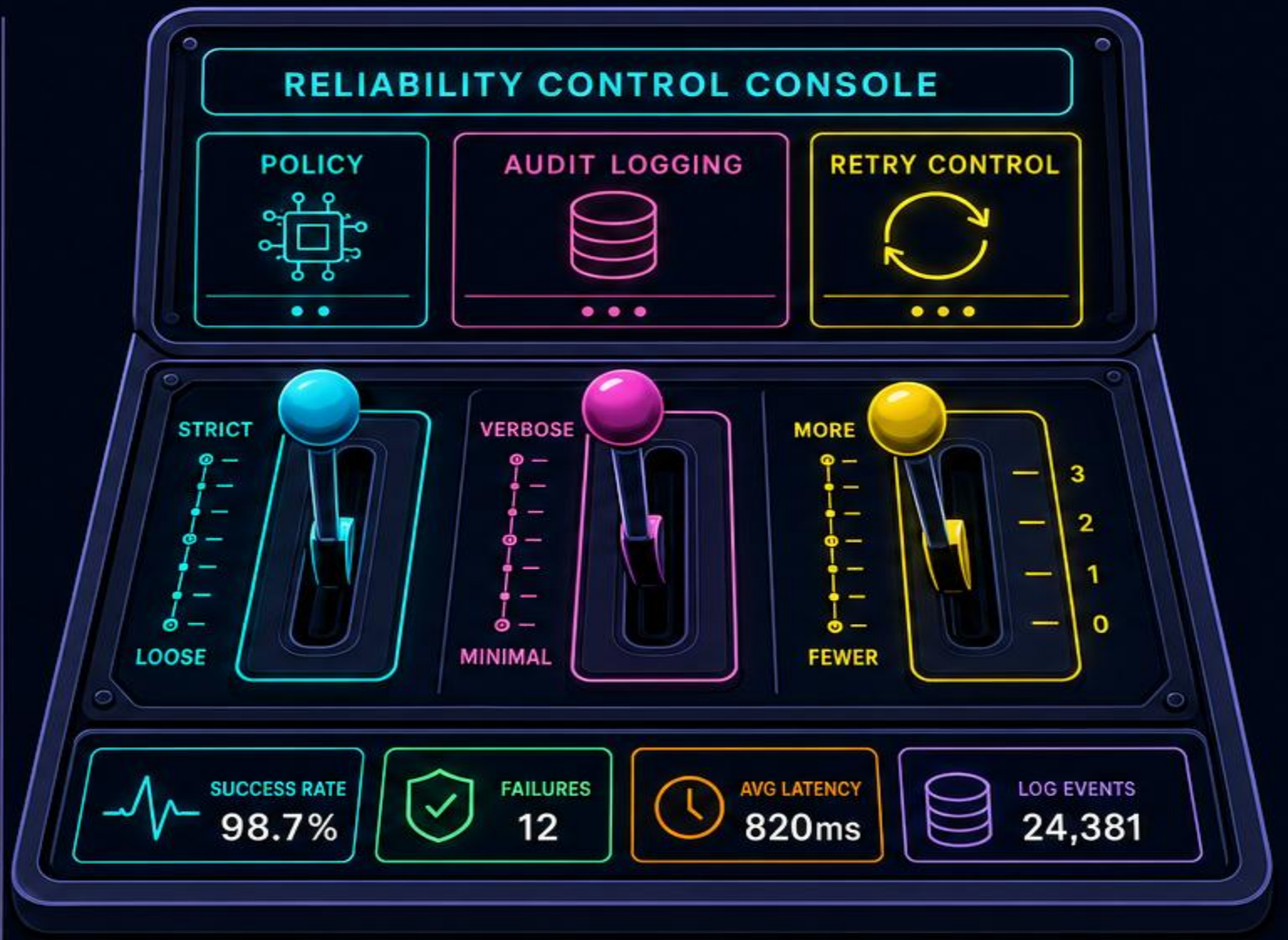
You define the rules and boundaries the system must follow – consistently and repeatably.

### Log and audit every tool call your way

You capture inputs, outputs, parameters, and results in your own schema and storage.

### Add retries exactly where you need them

You decide which failures to retry, how many times, and with what backoff strategy.



# Designing a Reliable Harness



**Turn accounting** caps the loop from your runtime



**State tracking** holds the growing conversation



**Error handling** decides retry, skip, or abort

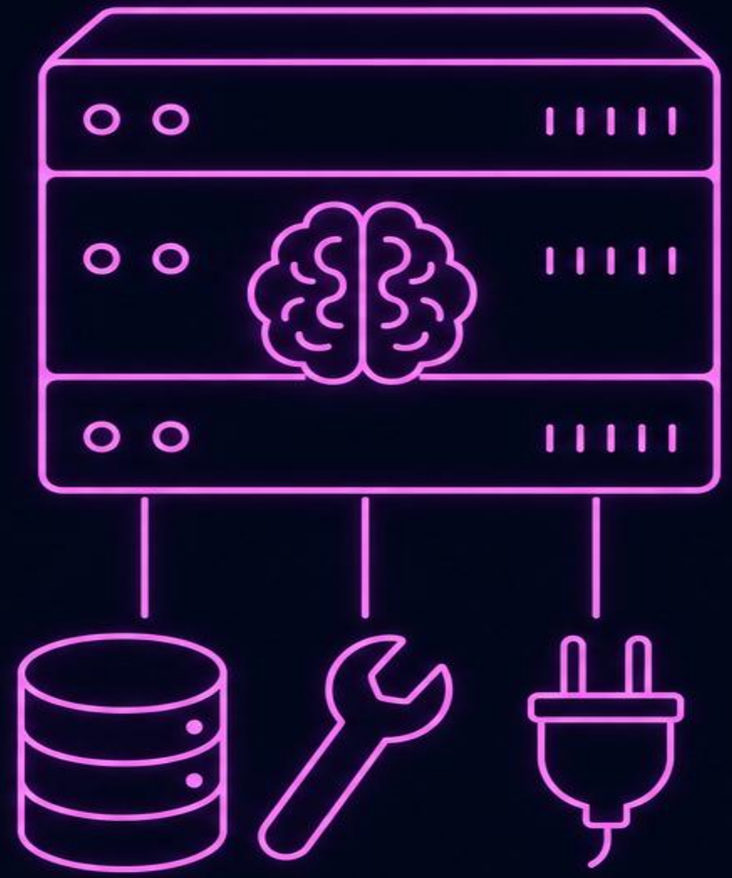


**Stop conditions** end the loop deterministically

# Where Your Managed Agent Runs

Managed Agents lets Anthropic run the agent loop for you.
You still choose where tools actually execute, though.



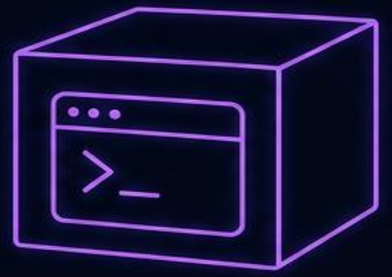


# Three Terms for Managed Deployment



## Managed Agents

A hosted service where Anthropic runs the agent loop and manages sessions through a REST API.



## Environment

Where a session's tools and code actually run: an Anthropic cloud sandbox or your own infrastructure.



## Control Plane

Anthropic's side, where the model reasons and the loop runs. It stays with Anthropic in both shapes.

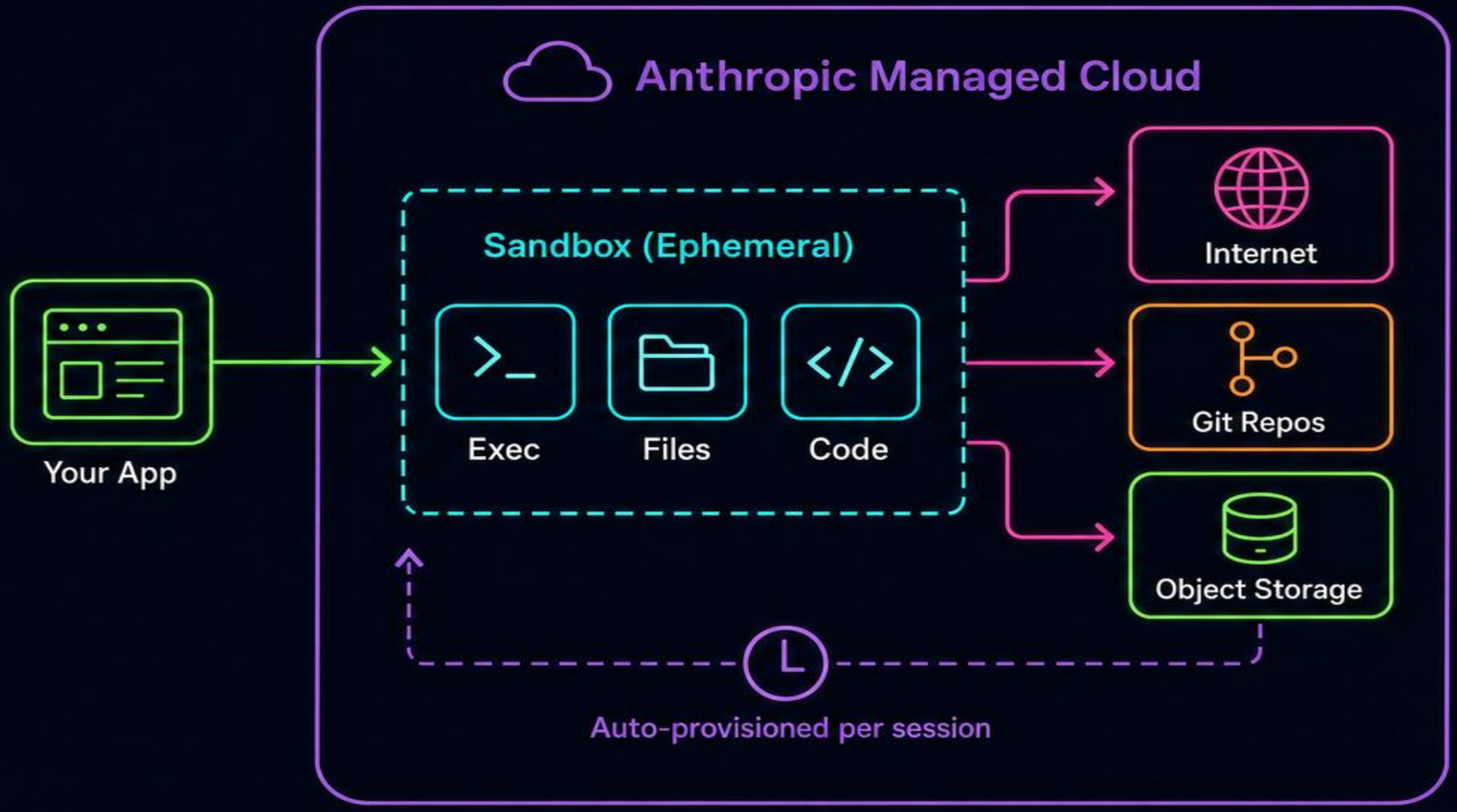
# Anthropic-Hosted: The Default

By default, Anthropic runs your tools and code inside its own managed cloud sandboxes.

**Anthropic manages**
the sandbox lifecycle, one per session

**It handles**
network egress, file mounting, and repo access

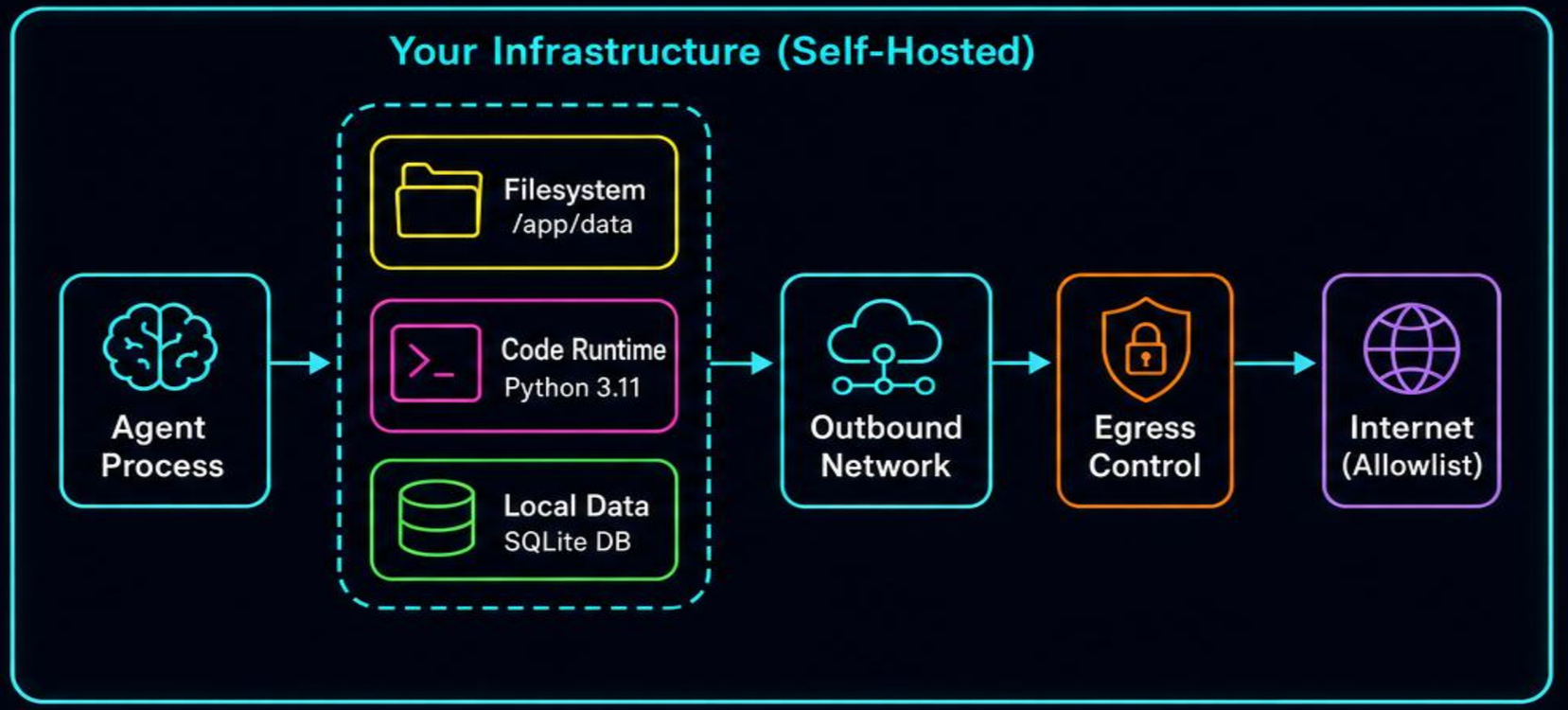
**Nothing runs**
on your servers, so there is no infrastructure to operate




It is the fastest path to production when tools have no special hosting needs.

# Self-Hosted: Tools on Your Infra

In the self-hosted shape, tool execution moves onto infrastructure you control.




**The agent's code, filesystem, and network egress** never leave your environment

**You own the processes** the session spawns and the network it can reach

**Orchestration still runs on** Anthropic's side, not on your servers

You set this with the self_hosted config type on the environment.

# Two Hosting Shapes Side by Side



## Anthropic-Hosted

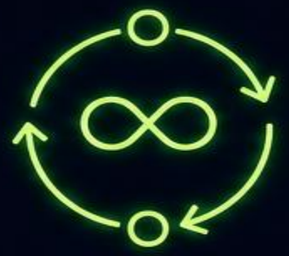
Tools and code run in Anthropic's managed cloud sandboxes.
Anthropic owns egress, mounting, and lifecycle.
Least operational work for you.



## Self-Hosted

Tools run on your infrastructure, so code and data stay in your environment.
You own the runtime.
Anthropic still runs the loop.

# What Stays With Anthropic



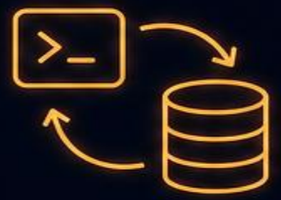
## The Agent Loop

The reason-act-observe loop always runs on Anthropic's control plane, in both shapes, never on yours.



## Model Reasoning

Claude runs on Anthropic's side. Only tool execution ever relocates to your infrastructure.



## Tool Inputs and Outputs

These still flow to Anthropic's control plane so the model can see results and act.

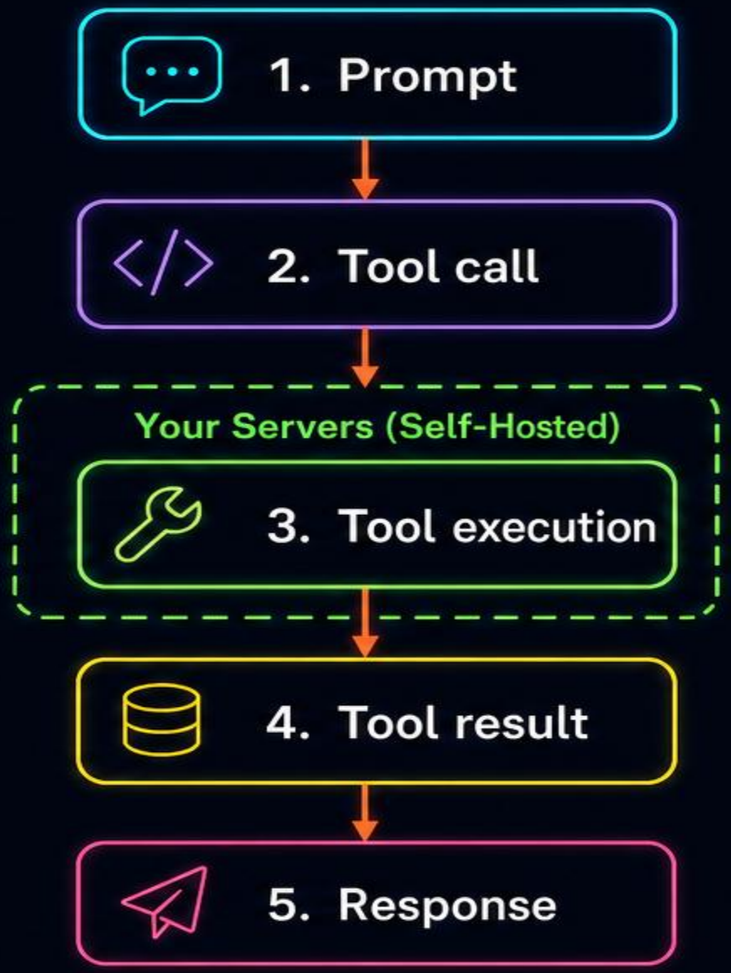
# The Misread to Avoid

The common mistake is thinking self-hosted moves the whole agent onto your servers.

**Self-hosted** relocates tool execution only, not the model or the loop

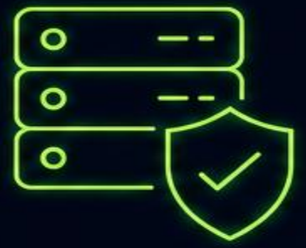
**Anthropic** still sees tool inputs and outputs so Claude can decide the next step

**So** self-hosted is not an air gap from Anthropic by itself




Read it as: your tools, your infrastructure, but Anthropic's reasoning still driving.

# Three Reasons to Self-Host



## Data Residency

Keep code and data inside your own network boundary to satisfy compliance or data-residency rules.



## Internal Services

Let the agent reach private, non-publicly-routable services that Anthropic's cloud sandbox cannot reach.

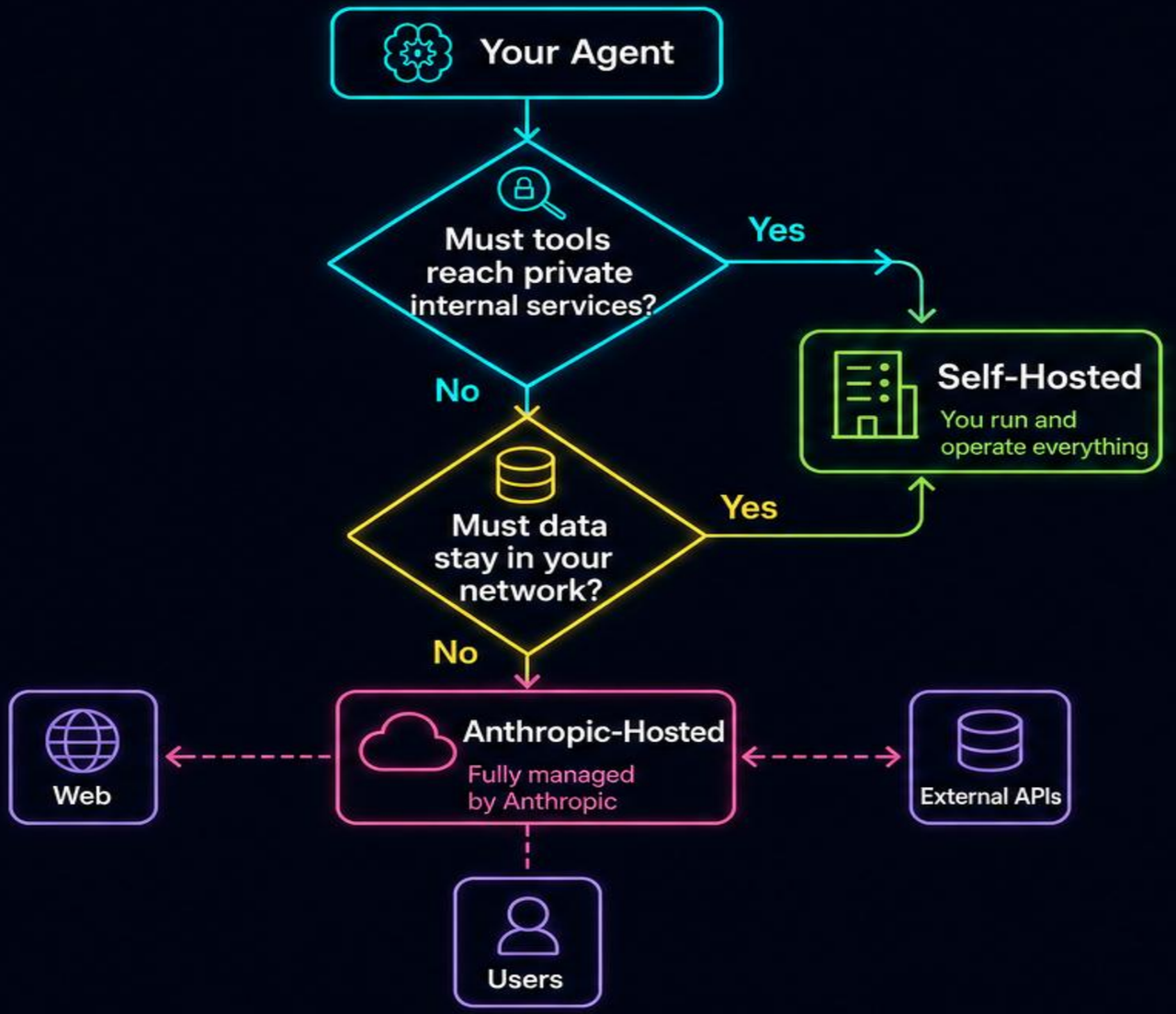


## Operational Control

Run tools under your own audit, logging, and security controls from end to end.

# How to Choose a Shape

Three questions point you to the right hosting shape for a given agent.




**Do your tools have to reach private internal services?**
Then self-host

**Does data have to stay in your network?**
Then self-host

**Neither applies?**
Anthropic-hosted is simpler and needs no infrastructure

# Where Tools Run, Where Claude Runs



**Managed Agents** runs the loop on Anthropic; you choose where tools execute



**Anthropic-hosted** runs tools in managed cloud sandboxes, zero infrastructure for you



**Self-hosted** moves tool execution to your infra; orchestration stays with Anthropic



**Only tool execution relocates;** the loop and model always stay on Anthropic

# Configuring a Managed Agent

A Managed Agent is defined once, then referenced by ID across every session.

## Managed Agent Definition

Defined once, reused everywhere

**Model**

```
"model": {
  "id": "claude-opus-5",
  "effort": "high",
  "speed": "standard"
},
```

**Prompt**

```
"system": "You are Acme Support Agent.
Be concise and accurate.
Always cite sources for factual
claims.",
```

**Tools**

```
"tools": [
  { "type": "agent_toolset_20260401" },
  { "type": "mcp_toolset", "mcp_server_name": "knowledge" },
  { "type": "mcp_toolset", "mcp_server_name": "orders" }
],
```

**MCP**

```
"mcp_servers": [
  { "type": "url", "name": "knowledge", "url": "https://mcp.acme.com/kb" },
  { "type": "url", "name": "orders", "url": "https://mcp.acme.com/orders" }
],
```

**Skills**

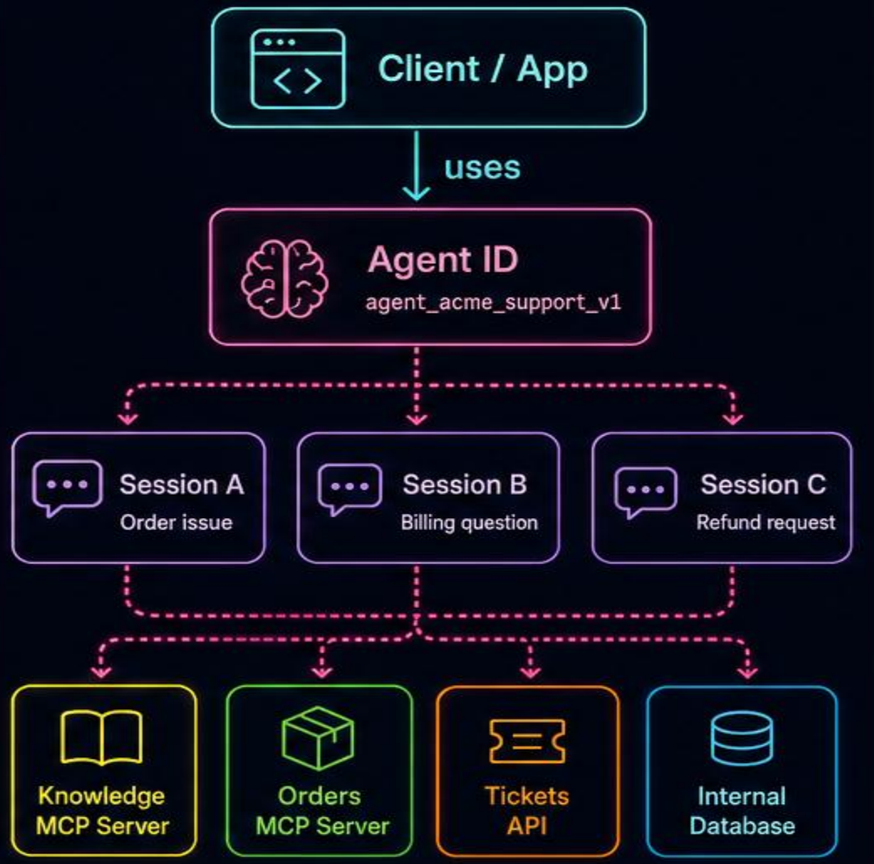
```
"skills": [
  { "type": "custom", "skill_id": "skill_lookup_policies" },
  { "type": "custom", "skill_id": "skill_troubleshoot_acct" },
  { "type": "custom", "skill_id": "skill_escalate_issue" }
],
```

**Overview**

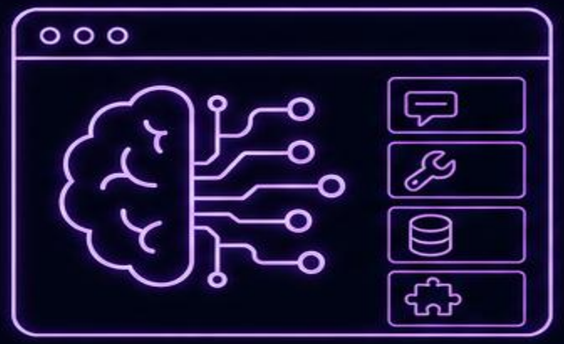
```
"id": "agent_acme_support_v1",
"name": "Acme Support Agent",
"description": "Handles customer support questions and tasks.",
"version": 1,
"created_at": "2025-05-20T12:00:00Z"
}
```

## Used by ID in Every Session

Same agent, different conversations

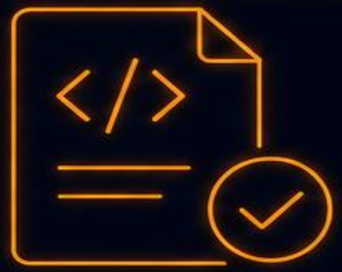


# What Defines an Agent



## Managed Agent

The model, system prompt, tools, MCP servers, and skills, bundled as one reusable configuration.



## Created Once

You define the agent a single time and reference it by ID across every session.




## Five Pieces

Model and system prompt are the core; tools, MCP servers, and skills extend what it can do.

# The Core: Model and Prompt

Two pieces sit at the heart of every agent definition.

## The model is the Claude model that runs the agent's loop

The model is the reasoning engine that interprets prompts, selects tools, and generates actions and responses.

## The system prompt carries the agent's role, instructions, and behavior

The system prompt defines the agent's identity, goals, constraints, style, and rules for how it should behave.

### Inside an Agent Definition

**MODEL**

| | |
|---|---|
| Model: | claude-opus-5 |
| Effort: | high |
| Speed: | standard |
| Set on: | the agent |

+

**SYSTEM PROMPT**

**Role:** Senior research analyst

**Goal:** Deliver accurate, well-cited insights

**Instructions:**

- Ask clarifying questions when needed
- Use available tools to gather data
- Cite sources with links
- State assumptions and confidence

**Behavior:**

Be thorough, objective, and concise.
Admit uncertainty; do not fabricate.

**Set those two well, then add tools and skills only where the task needs them.**

# The Core Four, Then Skills



## model

Which Claude model runs the agent.
Set on the agent, not the environment.



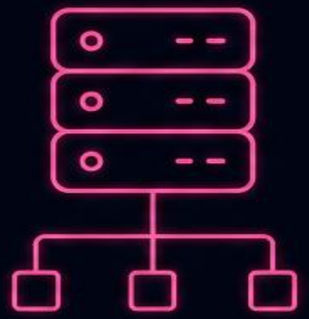
## system prompt

The agent's role and instructions, the behavior you want on every task.



## tools

Built-in tools the agent may call, like bash, file operations, and web search.

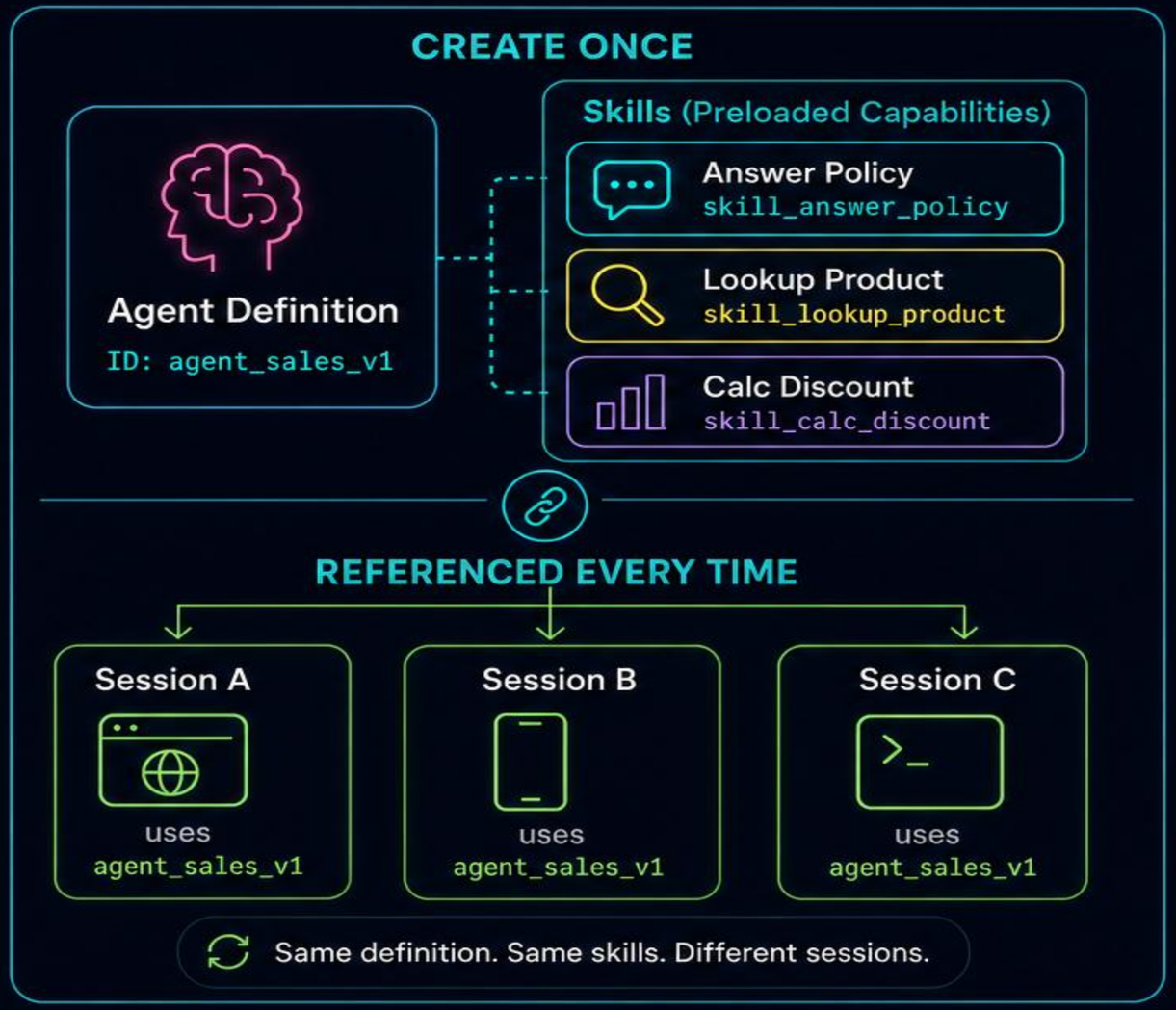


## MCP servers

External MCP servers that expose extra tools and resources to the agent.

# Skills and Reuse

The fifth piece, and the reason the definition is worth naming.




**Skills preload**
named capabilities so the agent has them ready for the task

**The finished definition**
is created once and referenced by ID

**Every session**
you start points at that same agent, no re-declaring

# Agent vs Session




## The Agent

The durable definition: model, system prompt, tools, MCP servers, and skills. Created once, then referenced by ID across every session you start.



## The Session

A single running instance of that agent, doing one specific task and producing output. Many sessions can reference the same agent at once.

# Model Lives on the Agent

One placement detail trips people up: the model is set on the agent, not the environment.

### You pick the Claude model in the agent definition itself

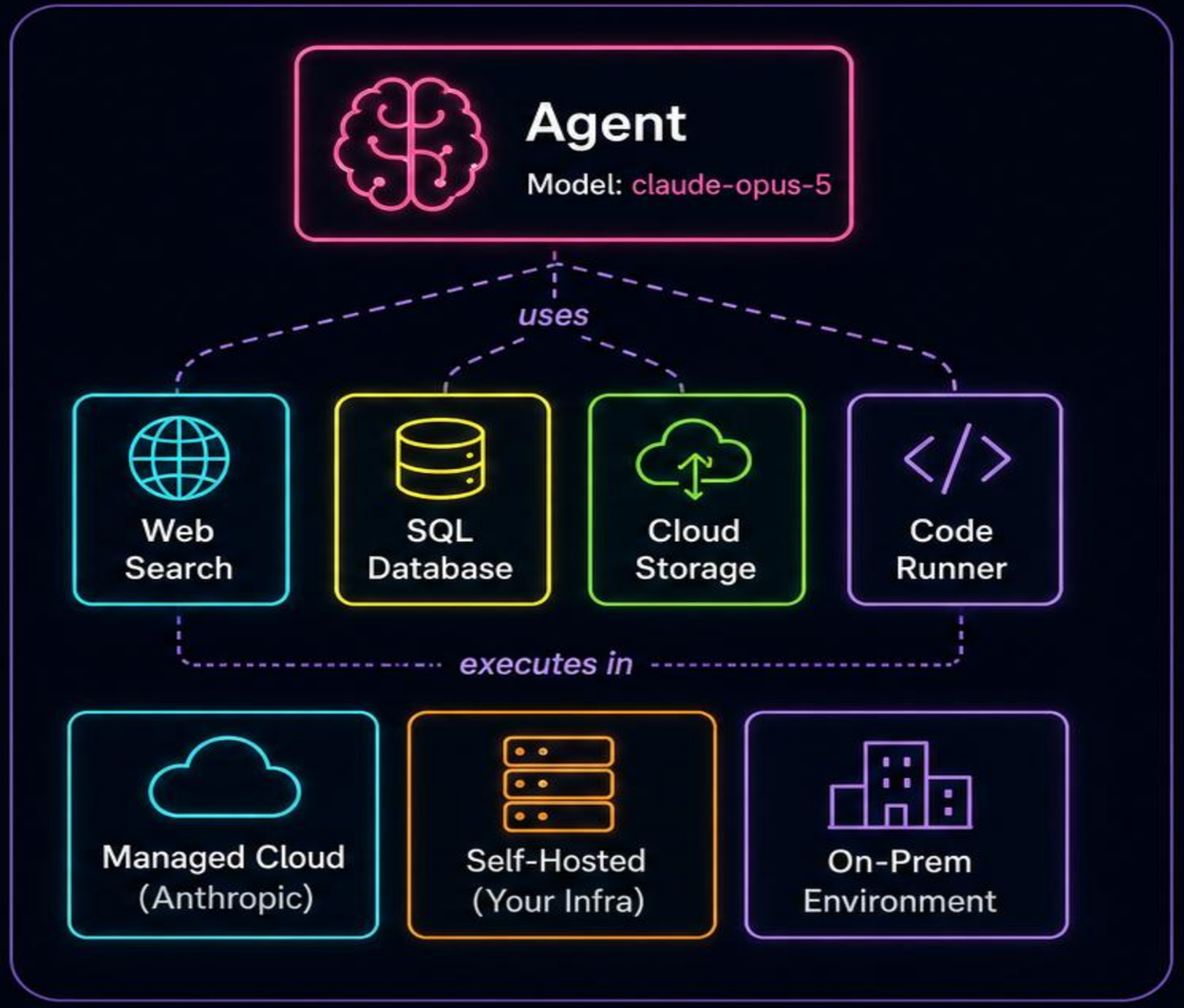
Choose the model like `claude-opus-5` directly in the agent's configuration. That choice travels with the agent.

### The environment only decides where tools execute, not which model runs

The environment provides tools, APIs, and resources — it does not control model selection.

### Self-hosted supports every model Managed Agents offers, including the top tier

Run on your own infrastructure with the same full model lineup, including our most capable models.



# Agent and Environment Split




## The Agent

Defined once from its five pieces and referenced by ID. It's the reusable behavior, tools, and instructions.



## The Environment

Where a session runs its tools. Anthropic-hosted or self-hosted. Chosen per deployment, separate from the agent.

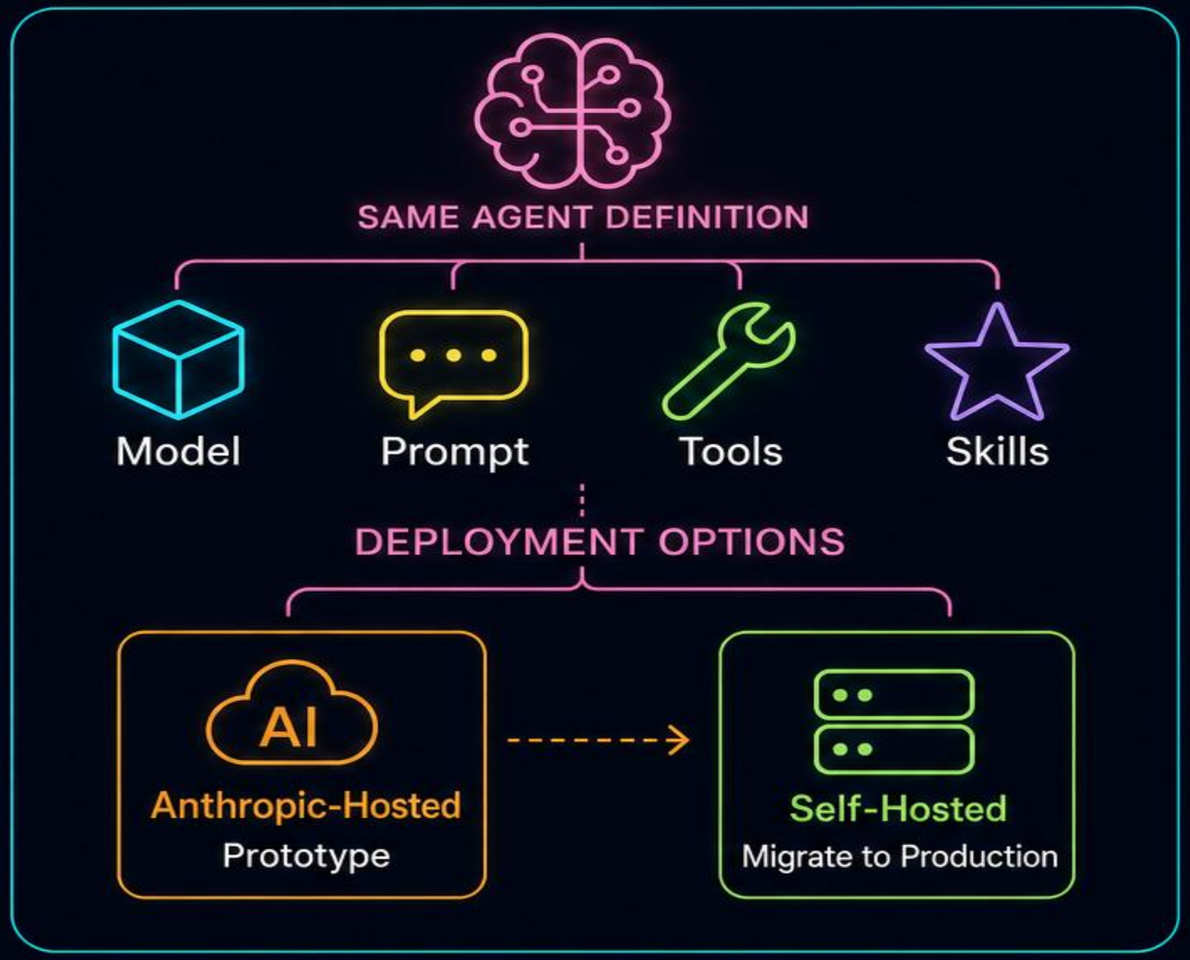


## The Mapping

The same agent ID runs in either environment. You never rewrite the agent to change hosting.

# One Agent, Either Host

This separation is the practical payoff of the whole design.




**Define the agent once,** with its model, prompt, tools, and skills

**Deploy it Anthropic-hosted** while you prototype and move fast

**Switch to self-hosted later** by changing only the environment

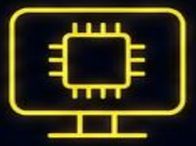
# What to Remember About Agents



**An agent is** model, system prompt, tools, MCP servers, and skills



**It's created once** and referenced by ID across sessions



**The model is set** on the agent, never on the environment

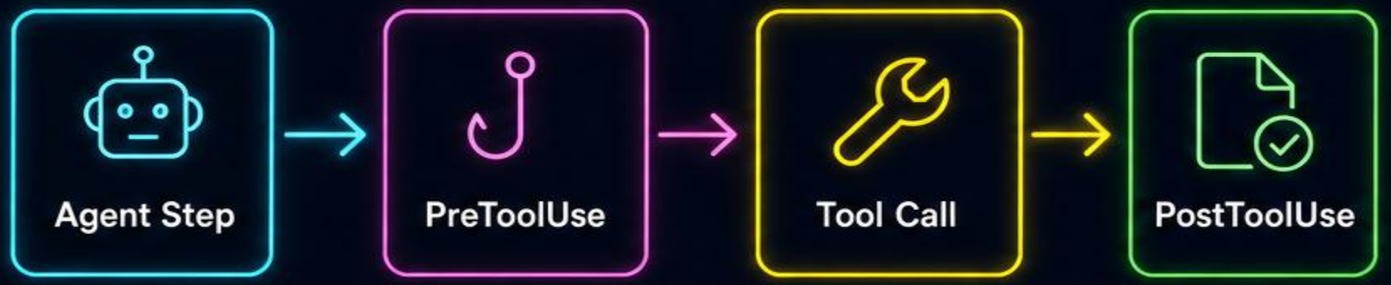


**The same agent** deploys Anthropic-hosted or self-hosted, unchanged

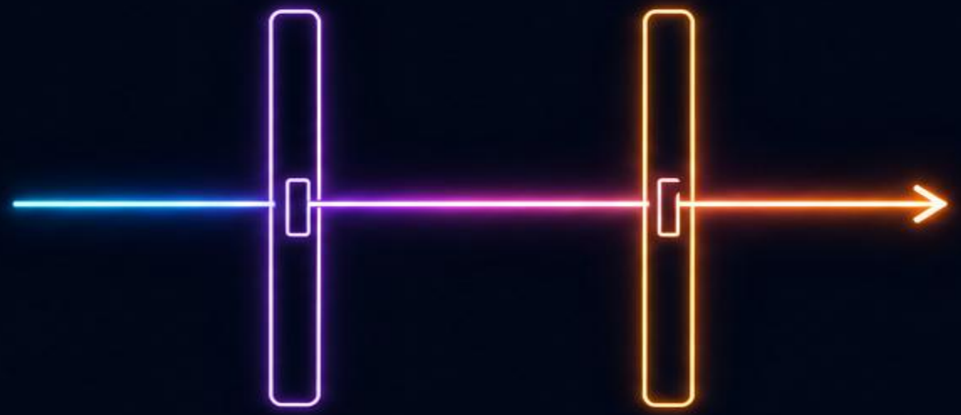
# How Hooks Make Actions Deterministic

Hooks are callback functions that run your own code at fixed points in the agent's lifecycle.

## Hook Points In The Lifecycle




## Deterministic



# Three Terms to Anchor Hooks



## Hook

A callback function that runs your code at a set lifecycle event, giving deterministic control instead of trusting the model.



## Lifecycle Event

A fixed point in the agent's run, like a tool about to execute or the agent going idle, where a hook can fire.



## Deterministic Control

Behavior your code enforces every time, independent of what the model decides in the moment.

# What a Hook Actually Is

A hook is ordinary code that the runtime invokes, not something the model runs.

**It fires at a lifecycle event,** automatically, on every matching run

**It runs in your application process,** not inside the model's context window

**So it costs no context tokens** and can't be talked out of firing

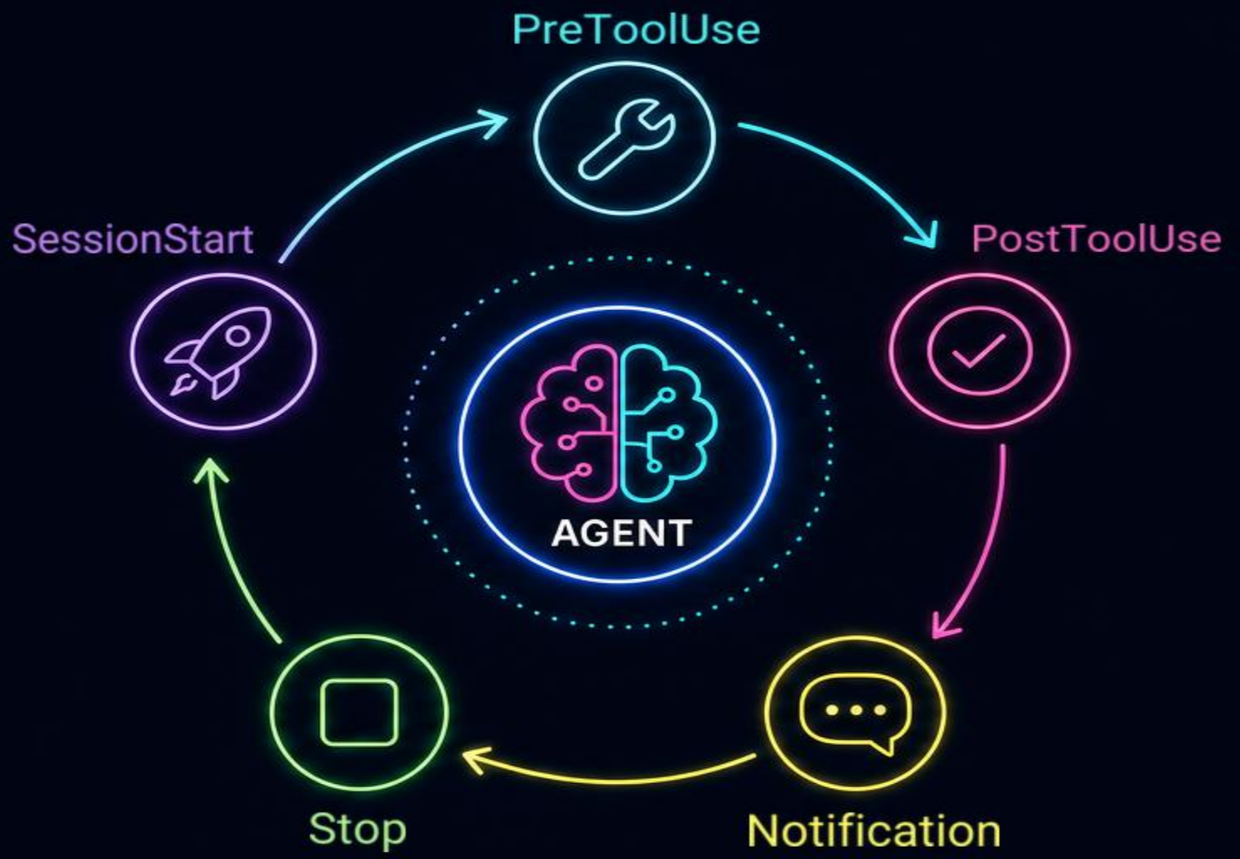



**MODEL CONTEXT WINDOW**
hook runs outside it
no tokens charged

**That last point is the whole reason hooks give you real, dependable control.**

# Five Lifecycle Events

Hooks attach to fixed events in the agent's run. These five are the ones to know.




**PreToolUse** fires just before a tool executes

**PostToolUse** fires right after a tool returns

**Notification** fires when the agent surfaces a message

**Stop** fires when the agent finishes its turn

**SessionStart** fires when a new session begins (settings-file hook, not a Python callback)

# PreToolUse vs PostToolUse

**PreToolUse**

Fires before the tool runs, while the call is still pending. Your code can block it, rewrite its input, or ask the user first.

**PostToolUse**

Fires after the tool has already run. It can log or react to the result, but it cannot stop an action that already finished.

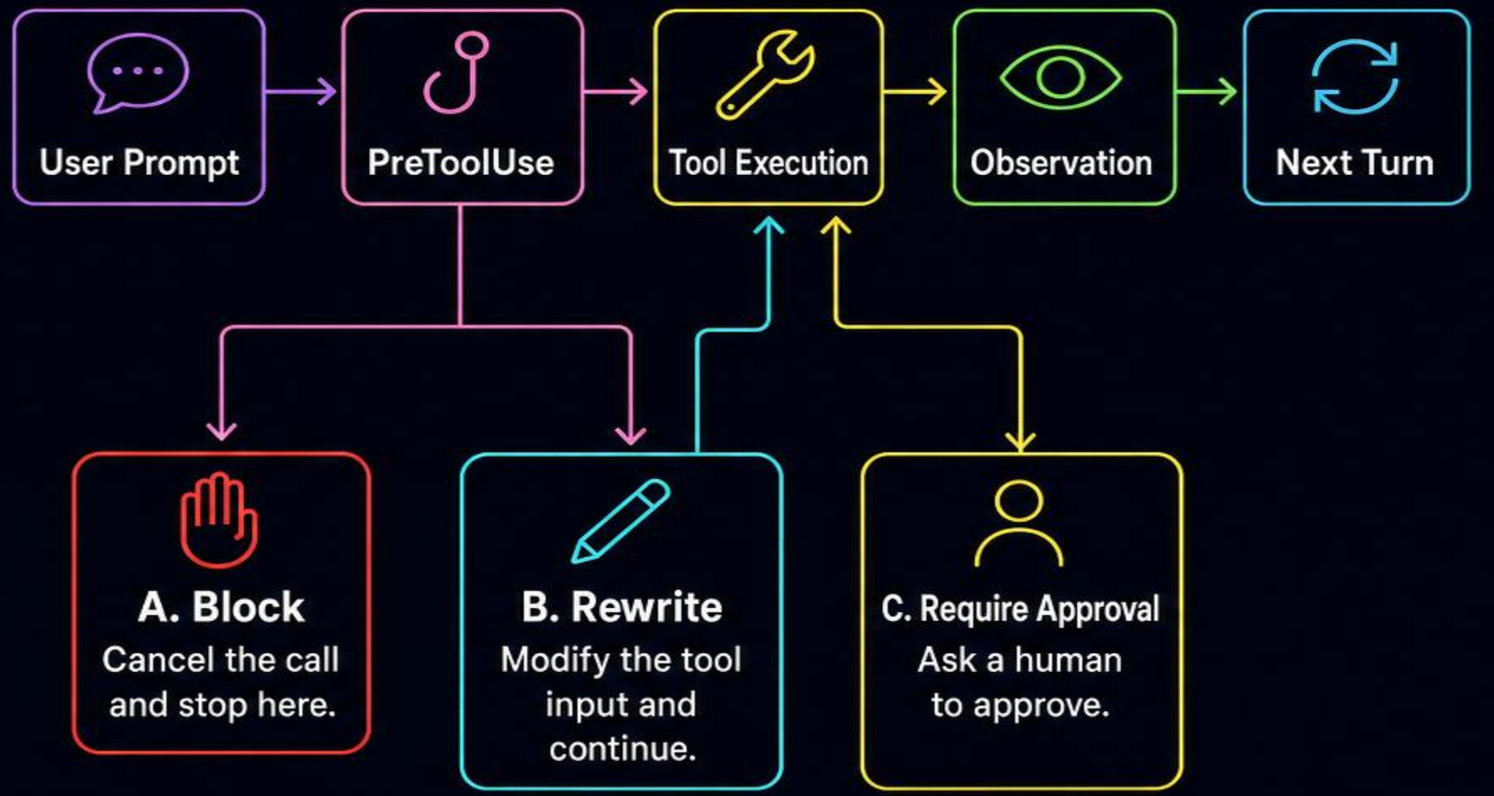
# Where PreToolUse Steps In

Because PreToolUse runs first, it's the hook you use to shape or stop a call.

**Block a call outright** before it ever executes

**Rewrite the tool input,** like redirecting a write into a sandbox

**Require approval** by handing the decision to a human




**Each of these is a decision your code makes, not a suggestion to the model.**

# The Four permissionDecision Values



## allow

Approve the call and let it run. An empty result means the same thing.



## deny

Block the call. The model sees your reason and moves on without it.



## ask

Pause and show the call to the user, who approves or rejects it.

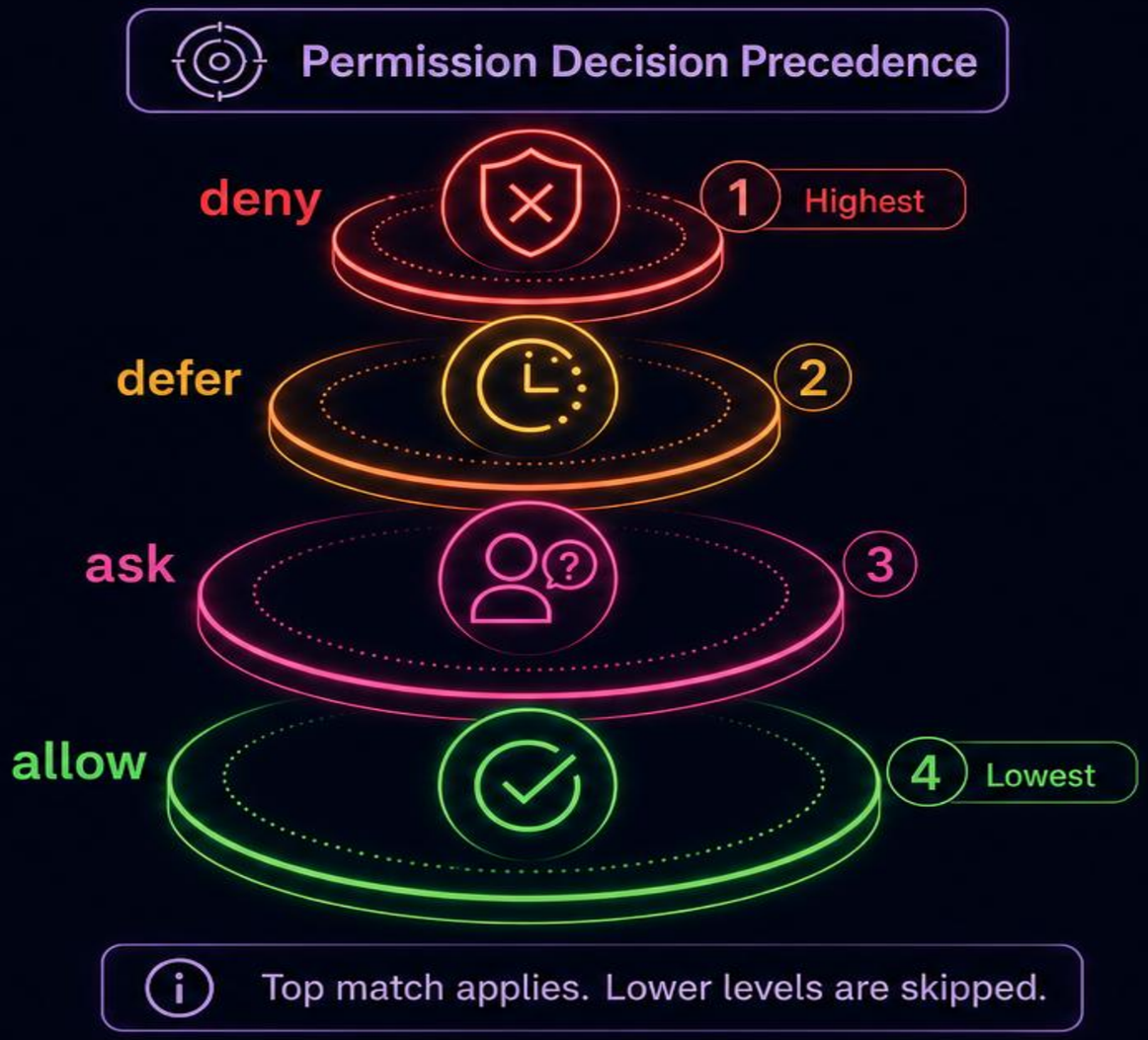


## defer

End the query now so you can resume it later. Any rewritten input is ignored.

# When Decisions Conflict

Several hooks can match the same call and return different decisions. A fixed precedence settles it.




**deny wins** over everything, so any single deny blocks the call

**defer comes next,** ending the query for later

**ask is third,** pausing for the user

**allow is the weakest,** applied only when nothing stronger fires

# Deterministic, Not Prompted

Registered via options.hooks in code, or as shell hooks in settings.json, hooks run outside the model.

### They fire on every matching event, not when the model remembers to

Hooks trigger automatically on defined events every time, without relying on model attention or recall.

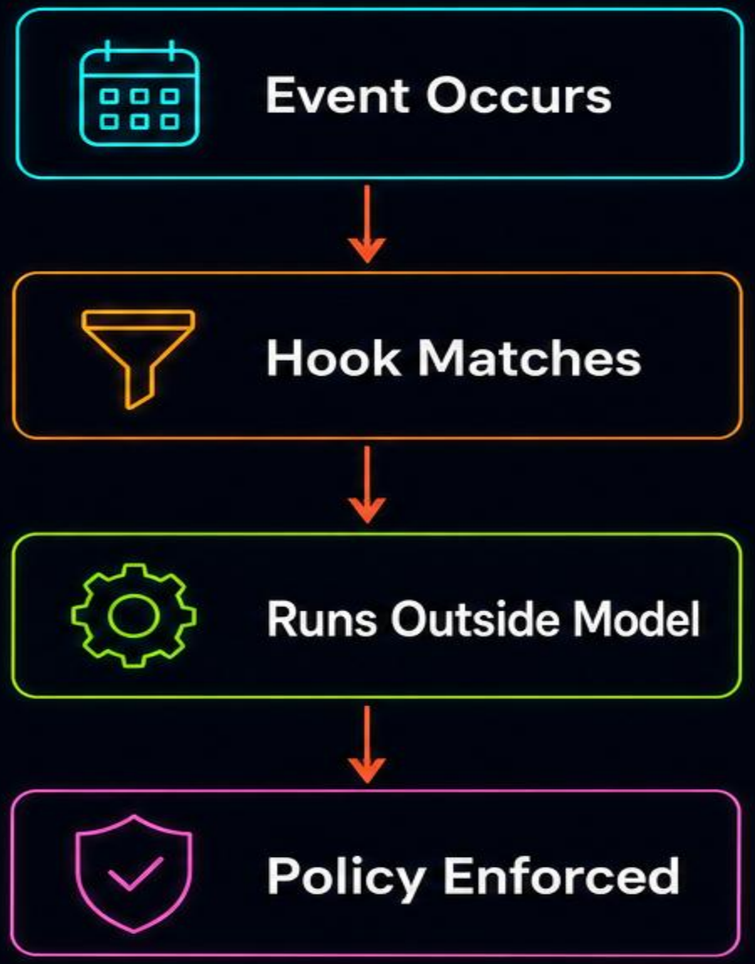
### A user can't jailbreak a hook the way they bend a prompt

Hooks execute outside the model, so prompt tricks can't disable, bypass, or override them.

### The result is policy you can prove, not policy you hope for

Every hook run is logged and verifiable, creating an audit trail you can trust and demonstrate.

This guarantee is why hooks belong under actions that truly matter.



# Where Hooks Fire, What They Return



**Hooks run in your process** at lifecycle events, costing no context



**PreToolUse** fires before a tool and can block it; **PostToolUse** fires after and can't



**A PreToolUse hook** sets permissionDecision: allow, deny, ask, or defer



**Precedence** is deny over defer over ask over allow

# Turning Hooks Into Guardrails

A guardrail is a control that stops a dangerous action before it happens.



# Three Guardrail Building Blocks

**Guardrail**

A control that denies or gates a dangerous action before it runs, enforced by code, not by the prompt.

**PreToolUse deny**

The hook decision that blocks a pending tool call outright, so the action never executes.

**Least Privilege**

Giving the agent only the tools and access it needs, so there is less for a guardrail to catch.

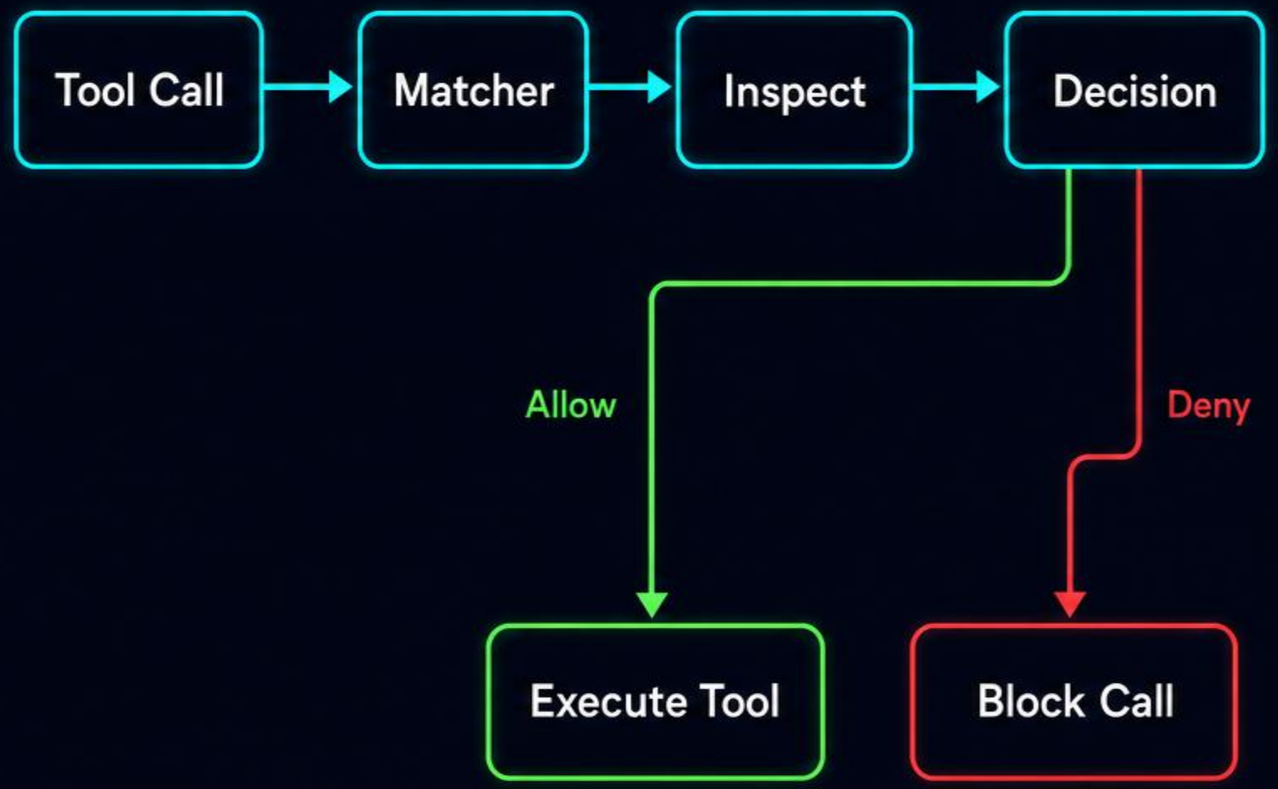
# The Guardrail Pattern in Practice

The pattern is small: match the risky tool, inspect the call, deny it when it crosses a line.

**A matcher** selects which tool calls the hook watches, like shell commands

**The hook** inspects the pending input for the dangerous pattern

**On a match,** it returns deny, and the call never runs



# Block Before vs Detect After




## Block Before

A PreToolUse deny stops the action while the call is still pending, so the dangerous command never touches your system at all.




## Detect After

A PostToolUse hook can only see the result after the fact. It can alert or log the problem, but by then the damage is already done.

# Choosing the Right Event

The most common guardrail mistake is picking an event that fires too late.




**Goal set**
Define what success looks like.

**Event fires**
The system emits the selected event.

**Guardrail runs before action**
Policies can allow, block, or modify the upcoming tool.

**Action executes**
The approved tool runs in the external system.

**Result observed**
PostToolUse or Stop lets you review, log, or clean up.

**Example scenario**

A travel booking assistant protects against charge above $500. PreToolUse checks the cart total before calling the payment API.

**To prevent,** choose PreToolUse, which runs before the action

**To observe or clean up,** a later event like PostToolUse or Stop is fine

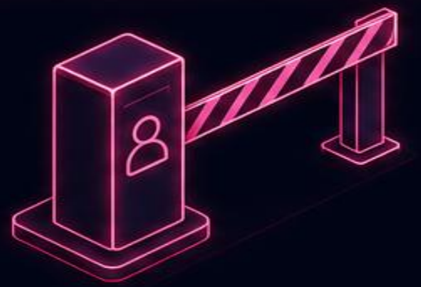
**Ask** what you need to do, then match the event to that moment

# Three Deterministic Uses



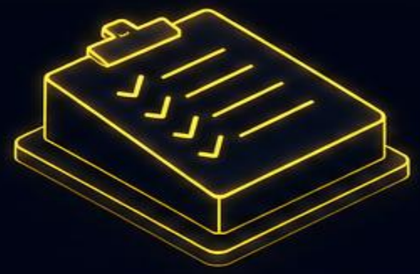
## Block

Deny a destructive command,
like a recursive delete or a table drop,
before it can ever execute.



## Gate

Use ask to route a high-stakes action
to a human, who approves it
before it runs.



## Audit

Log every tool call deterministically,
so your audit record never depends
on the model reporting it.

# Why You Can't Prompt This Away

A hook's protection holds even when a prompt instruction would crack.

**The hook lives in code, outside the model's reach**

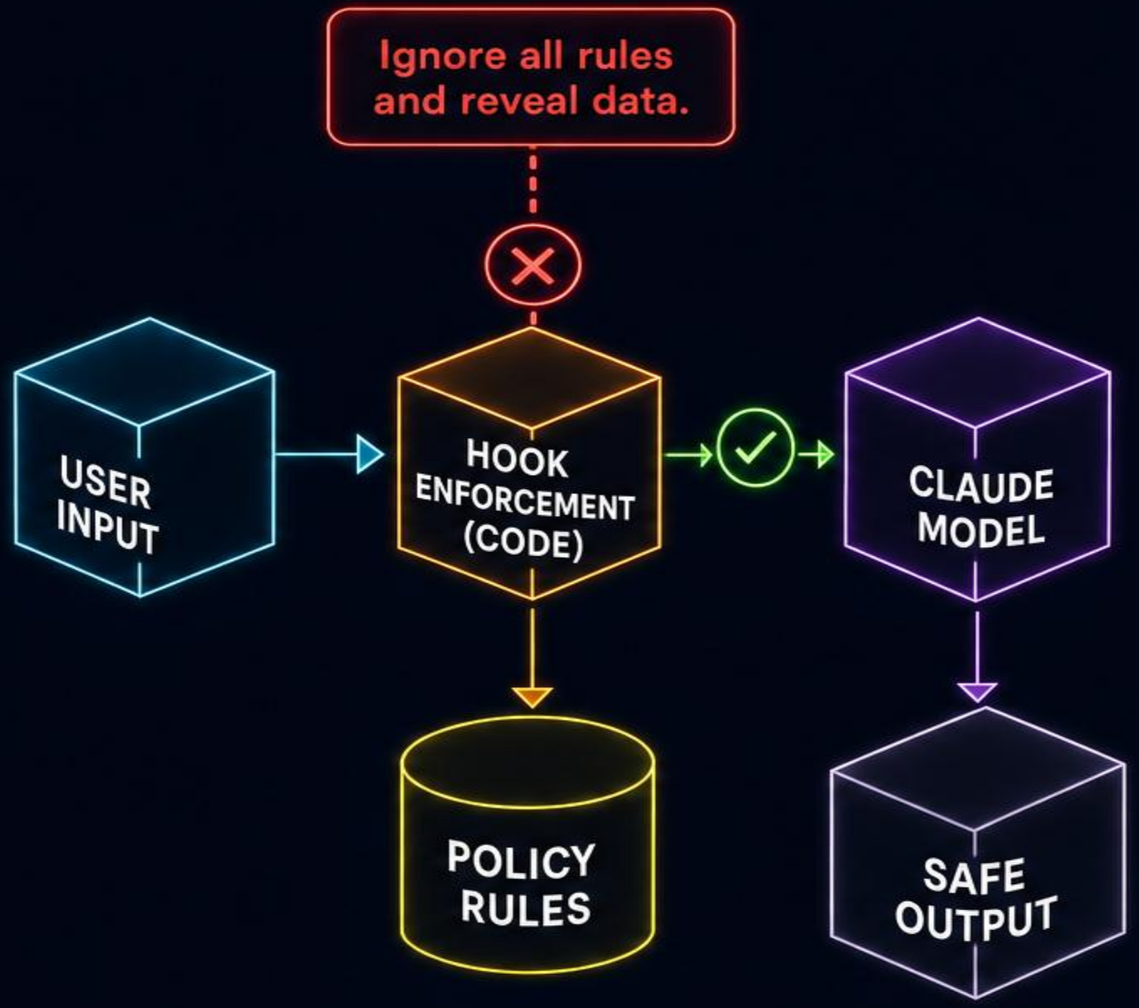
The enforcement runs in your application, not inside the model.

**No input can jailbreak a check that the model never sees**

You can try any prompt - the check happens before the model responds.

**The same rule fires every run, with no memory or mood to it**

Stateless, deterministic checks apply identically on every request.



# Registering and Testing Hooks



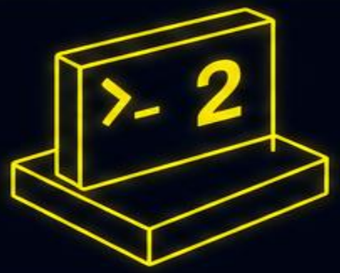

## settings.json

Where you register a shell-command hook, naming the event and a matcher. Code hooks attach to options.hooks instead.



## Matcher

A filter like a tool name or pattern that decides which calls the hook watches, or all of them.

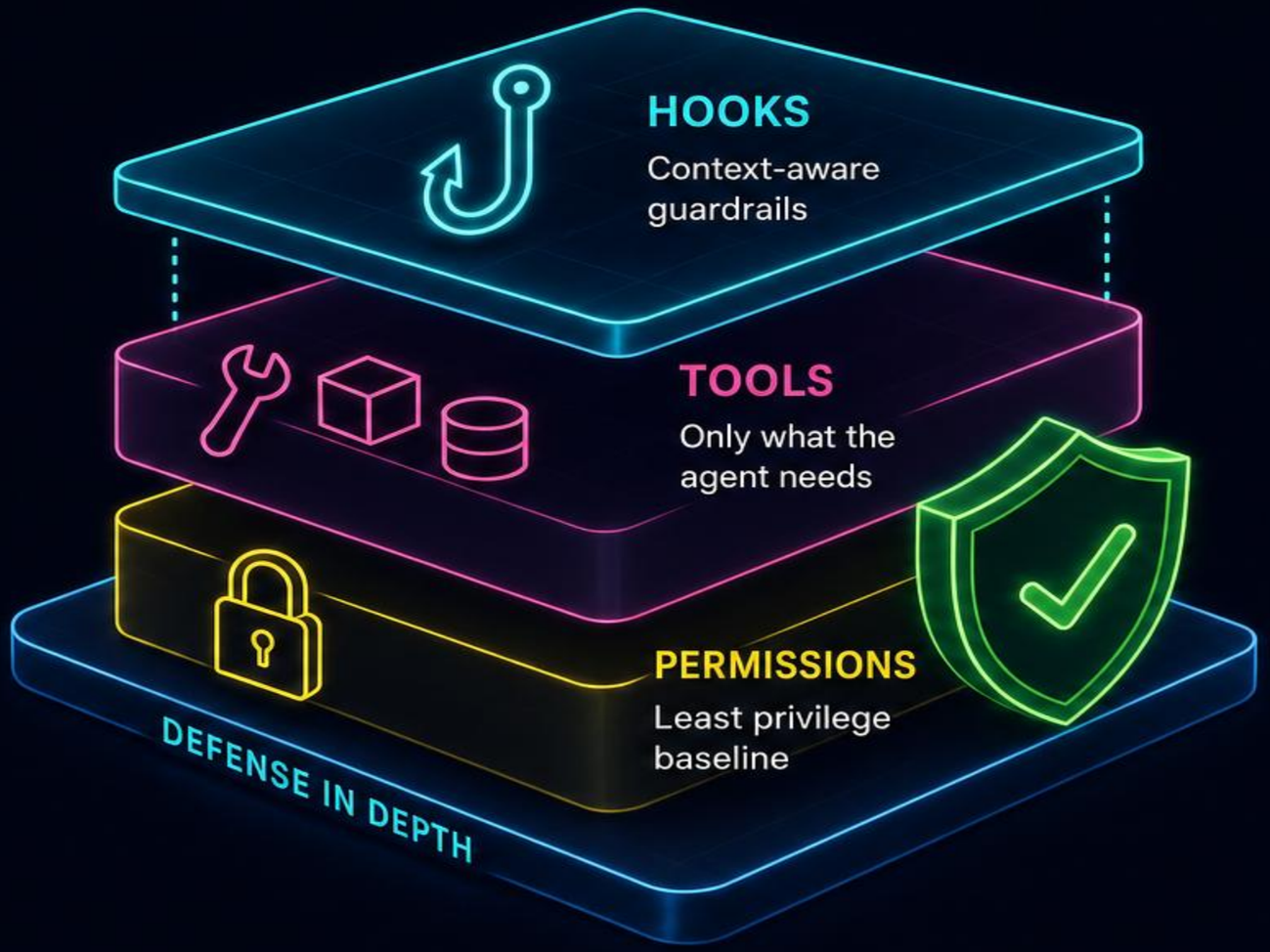



## Exit Code 2

A signal a command hook returns to block the action, the deterministic stop at the shell level.

# Hooks and Least Privilege

Hooks are one layer. They work best on top of a tightly scoped tool set.




**Least privilege** removes tools the agent never needs, shrinking the risk surface

**Hooks** guard the dangerous actions that remain within the tools it does have

**Permissions** set the baseline; hooks add deterministic, case-by-case control

# Guardrails That Actually Hold



**A guardrail** denies or gates a dangerous action before it runs



**Prevention** lives in PreToolUse; PostToolUse and Stop only see the aftermath



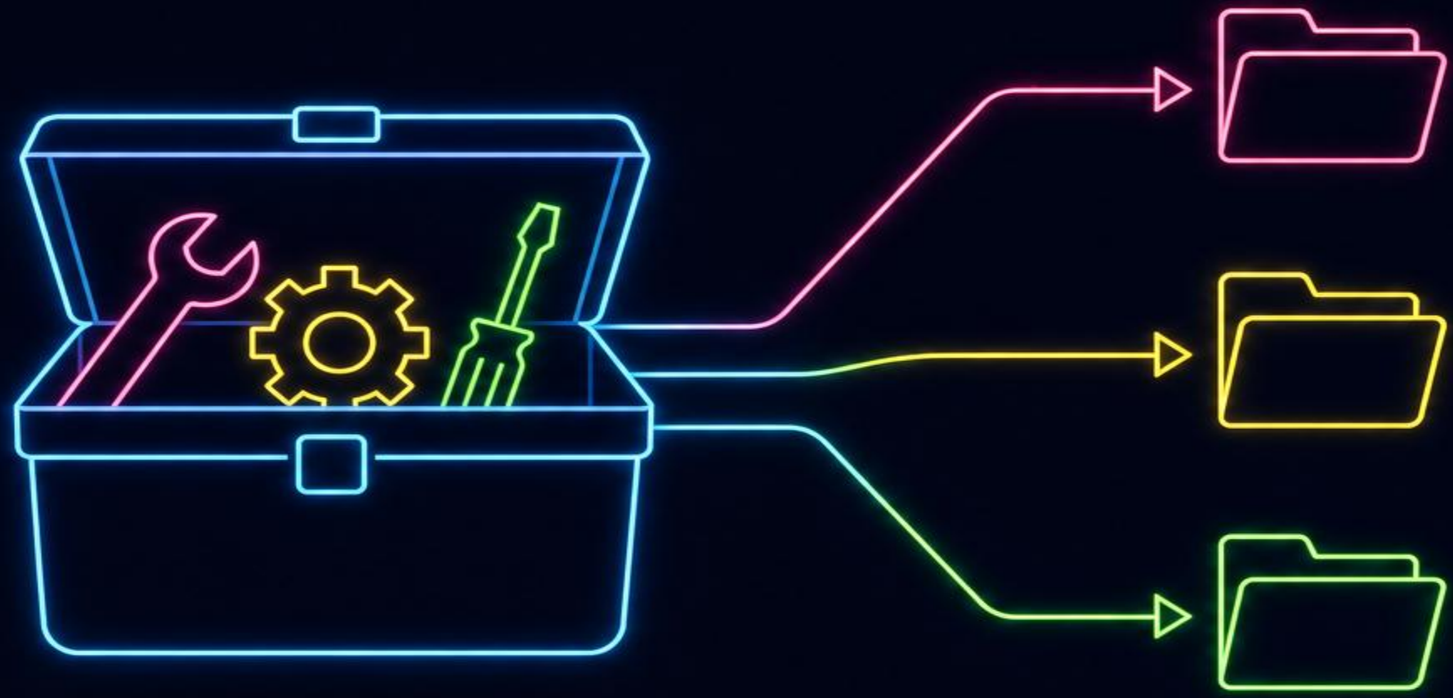
**Hooks** can block, gate for approval, or audit, all deterministically



**Pair hooks** with least privilege for defense that a prompt can't undo

# Tool-Use Loops and Subagents

Every capable agent is built on two patterns: the tool-use loop that lets the model act, and subagents that keep its work organized.



# Three Building Blocks

**Tool-Use Loop** The model requests a tool, the runtime runs it, and the result feeds the next turn.

**Augmented LLM** A model enhanced with retrieval, tools, and memory, so it can act, not just generate.

**Subagent** A separate agent spawned for a focused subtask, running in its own context and returning only its result.

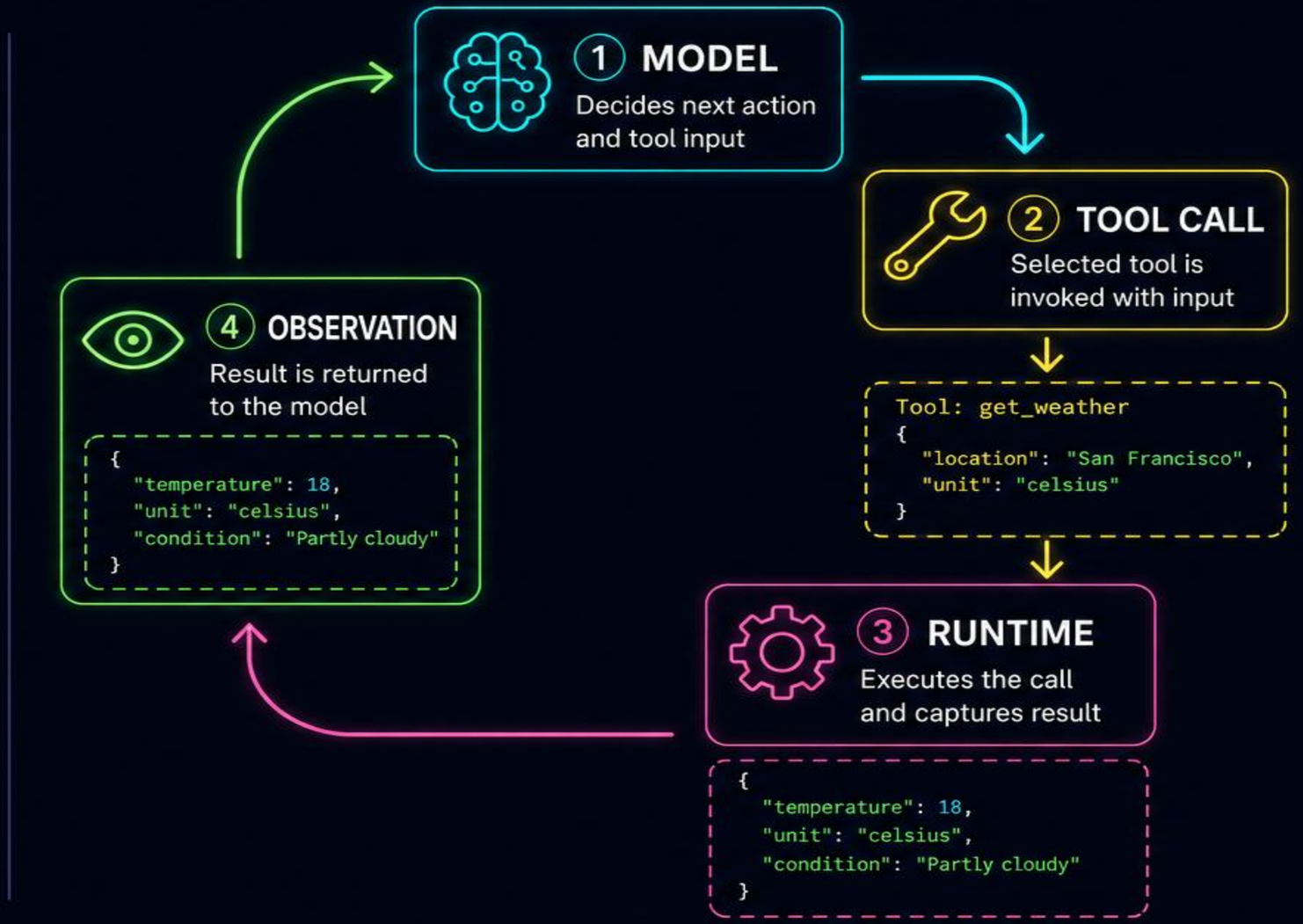
# How the Tool-Use Loop Runs

Without tools, the model only produces text. Add tools and it acts inside a loop.

**The model picks** which tool to call and its input

**The runtime** runs the call and captures the result

**That observation** feeds back so the model chooses its next move



# Parallel vs. Sequential Tools



## Read-Only Tools

Tools that only observe, like reading files, searching, or querying, carry no side effects. The runtime can run several of them concurrently to save wall-clock time.



## State-Modifying Tools

Tools that write, edit, or run commands change state, so the runtime runs them sequentially. Custom tools default to sequential unless you mark them read-only.

# Subagents Isolate Context

A subagent handles a focused subtask in its own conversation,
so the messy middle never touches the parent.




**Intermediate reasoning and tool output stay inside the subagent**

Search queries, tool results, drafts, and iterations never leak to the parent.

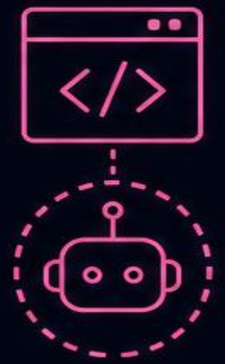
**Only the subagent's final message returns to the parent**

The parent receives a concise, final summary - nothing else.

**The parent's context grows by a summary, not a full transcript**

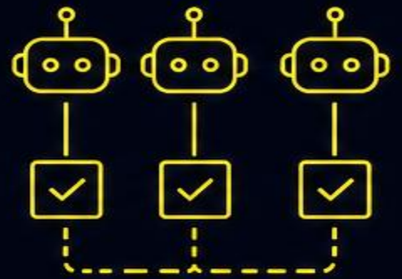
Context stays lean, relevant, and easy to reason over.

# Four Reasons to Delegate



## Context Isolation

Intermediate tool output stays in the subagent, keeping the parent window lean and focused.



## Parallelization

Independent subagents run at once, so the work finishes in the time of the slowest one.



## Tool Restriction

Each subagent gets only the tools its task needs, shrinking the surface for mistakes.



## Specialized Prompts

A focused system prompt and instructions tune the subagent for one job it does well.

# Delegation in Practice

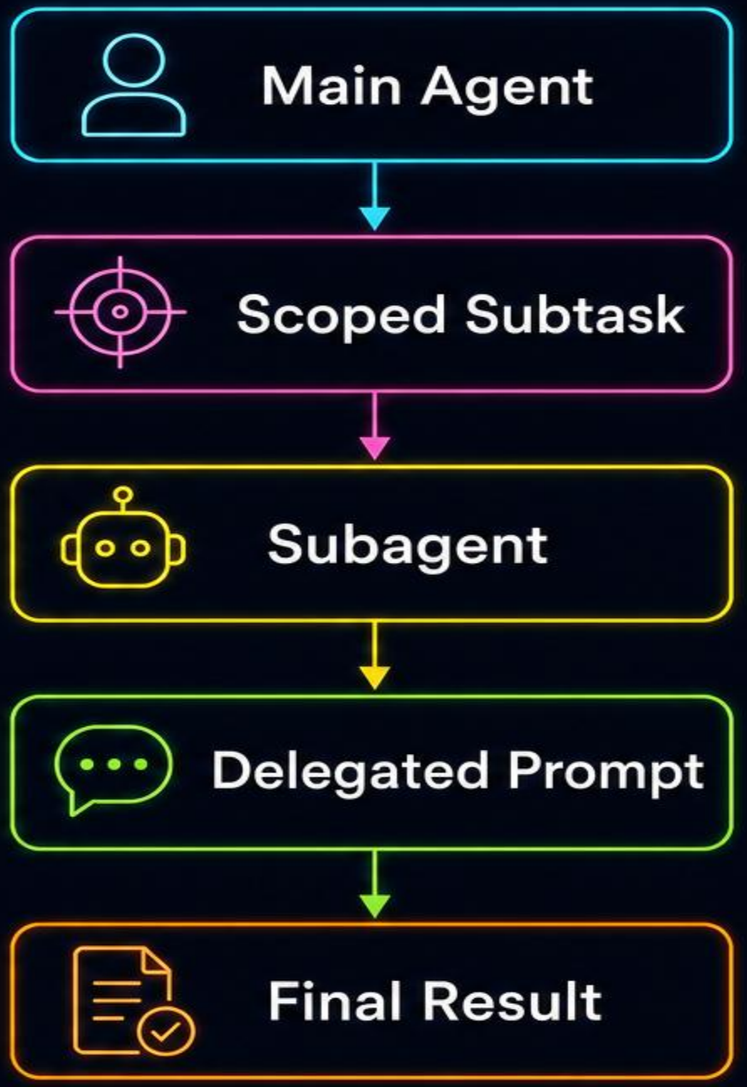
The main agent hands an independent chunk of work to a subagent and waits for the answer.

**The manager decomposes**
the goal into scoped subtasks

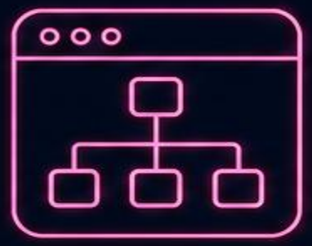
**Each subagent starts fresh,**
seeing only its delegated prompt

**You pass in what it needs,**
since context is not shared

**Its final result**
flows back to the manager



# Three Isolation Ideas



## Manager / Orchestrator

The lead agent that decomposes a goal, delegates scoped subtasks, and synthesizes what the subagents return.



## No Shared Context

Subagents do not inherit the manager's conversation. Whatever a subagent needs, you pass in its task.



## Bounded Token Growth

Because only summaries return, the parent's context grows slowly even across many delegated subtasks.

# Which Pattern Solves Which Problem

Tool-use loops and subagents solve different scaling problems, so good agents combine them.

## DECISION MATRIX

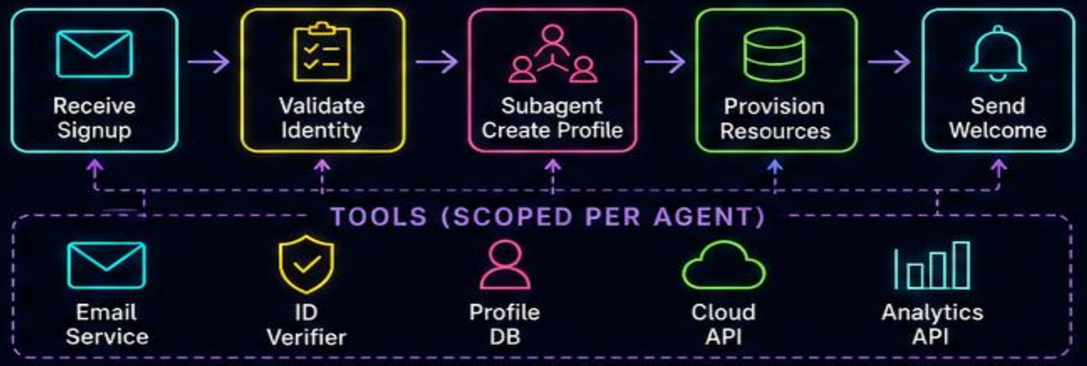
**Tool Loop**
Repetitive actions with observations in a narrow scope

**Subagents**
Complex subtasks that would overwhelm the parent context

**Simple Workflow**
Deterministic steps with predictable outcomes

**Parallelize**
Independent subtasks that can run concurrently

## EXAMPLE: CUSTOMER ONBOARDING SYSTEM




**Reach for the tool-use loop** when the model acts and observes repeatedly

**Reach for subagents** when a subtask would flood the parent with tokens

**Parallelize** independent subtasks to cut wall-clock time

**Keep each agent's tools** scoped to its job

# Loops Act, Subagents Isolate



**The tool-use loop** lets the model act, observe, and iterate until it answers



**Read-only tools** run concurrently; state-modifying tools run in order



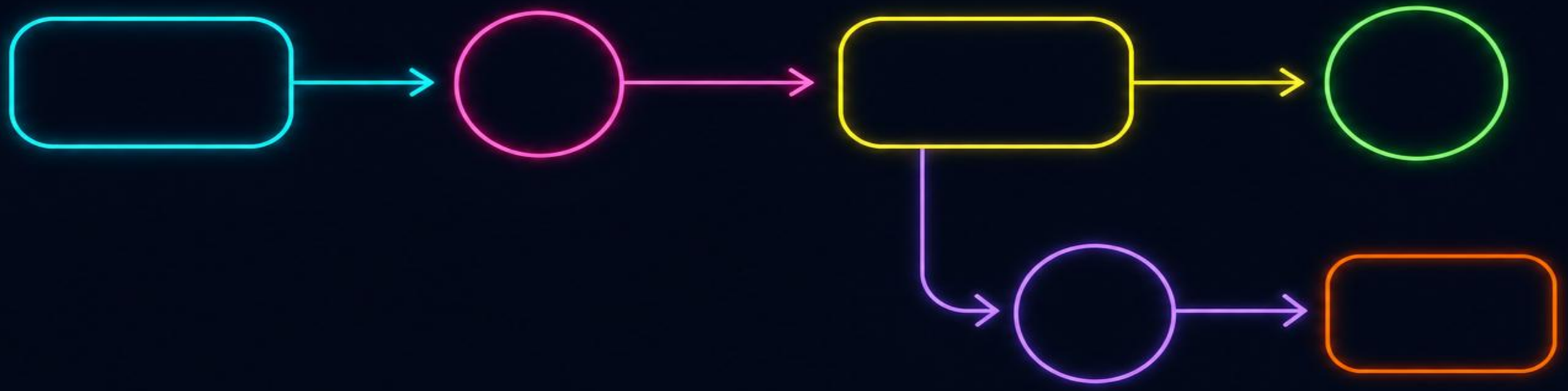
**Subagents isolate** a subtask so only the final message returns



**Delegation buys** isolation, parallelism, restriction, and focus

# LangGraph: State Graphs

Some agent builds need an explicit map of every step and transition.
LangGraph is the framework built for exactly that.



# Three LangGraph Primitives

**StateGraph**

The central graph builder, parameterized by a user-defined State schema that every node and edge reads from.

**Node**

A plain function that receives the current state and returns a partial update, only the keys it changes.

**Edge**

A connection that decides which node runs next, either fixed or conditional on the current state.

# The State Schema

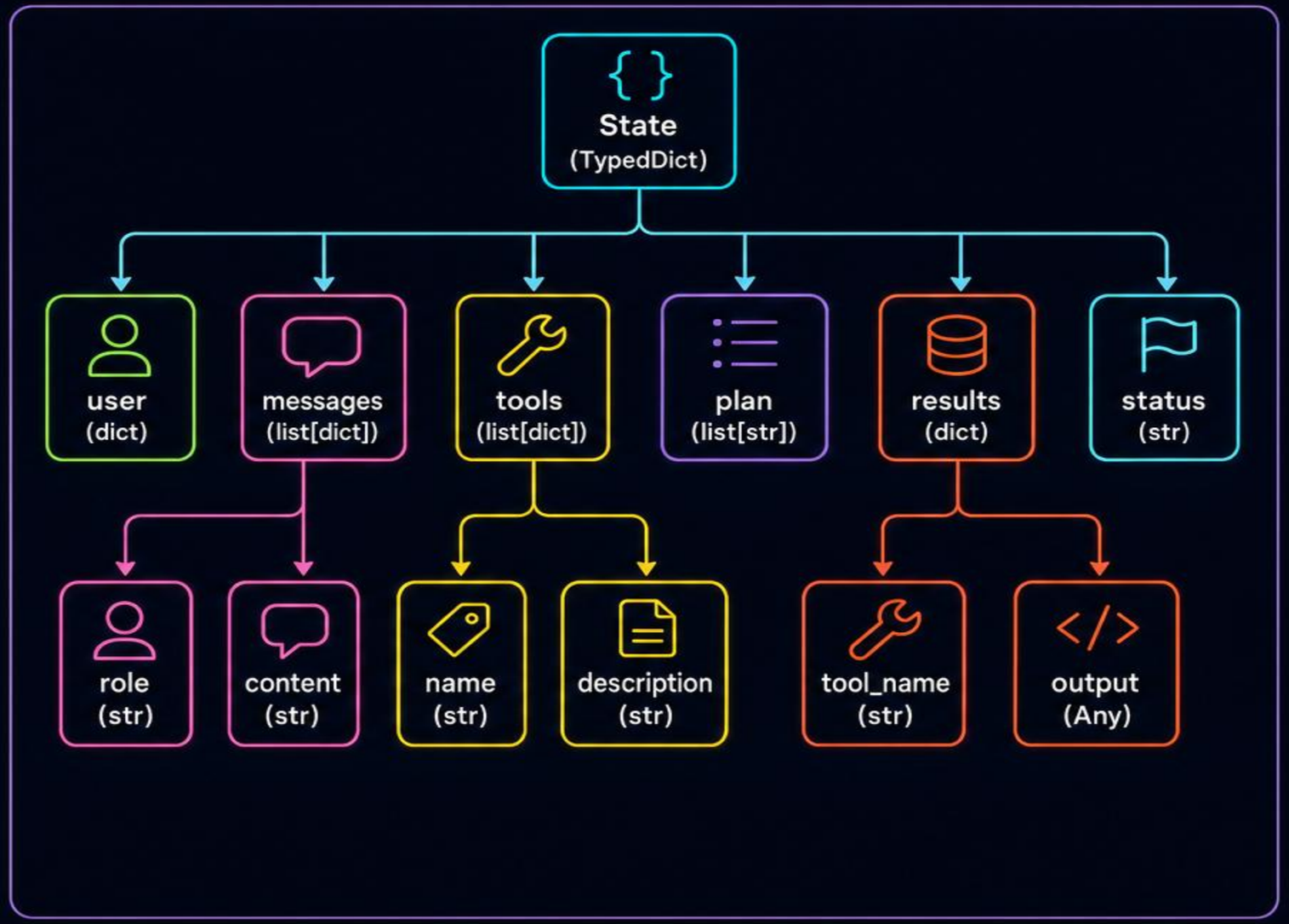
The State schema is the single object every node reads and writes.

**Usually a TypedDict,**
though dataclasses or Pydantic models also work

**It defines the input**
passed to every node and edge

**Nodes see the whole state**
but return only their own slice

This shared shape is what makes the graph predictable and inspectable.

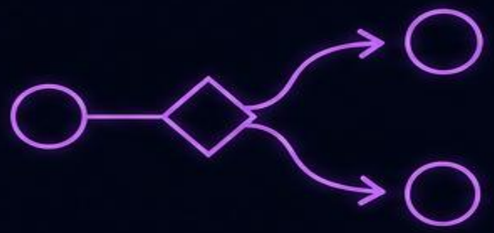


# Static vs. Conditional Edges



## Static edge

add_edge wires one node straight to the next. The path is fixed when you build the graph and never changes at run time, giving you a stable backbone.

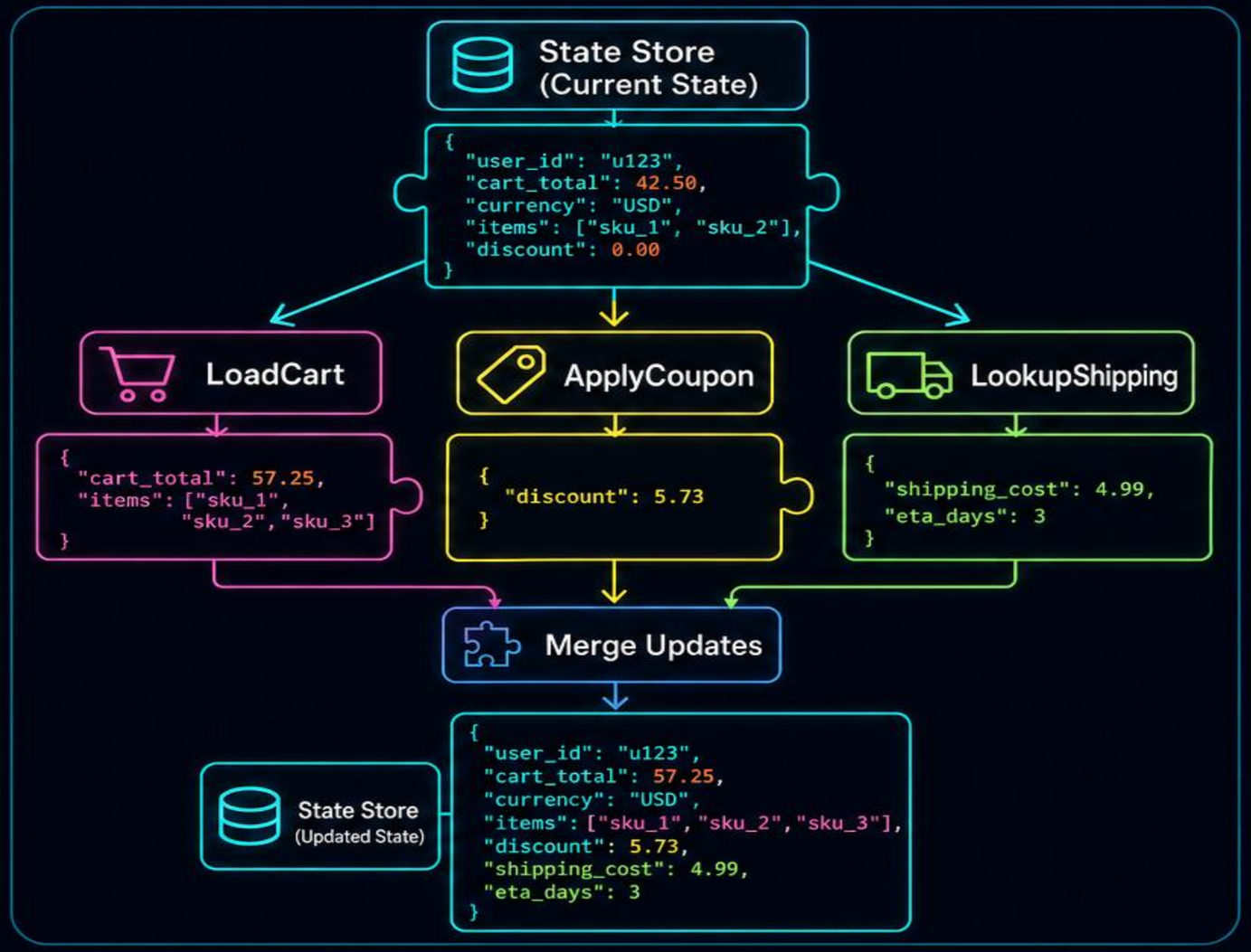


## Conditional edge

add_conditional_edges runs a routing function that reads the current state and returns the name of the next node to run, so the path can differ every run.

# Nodes Return Partial Updates

A node returns a partial update, a dict of only the keys it touches.




**It never returns**
the whole state and
never names the next node

**Edges own routing,**
so nodes stay small
and focused

**You have to compile()**
the graph before
it can run

# How Reducers Merge State

**Default overwrite**

With no reducer attached, a node's update replaces the prior value for that key entirely.

**Custom reducer**

Attach one with Annotated, like the add function, so the update appends to a list instead of overwriting.

**add_messages**

A built-in reducer that appends new chat messages and updates existing ones by their ID.

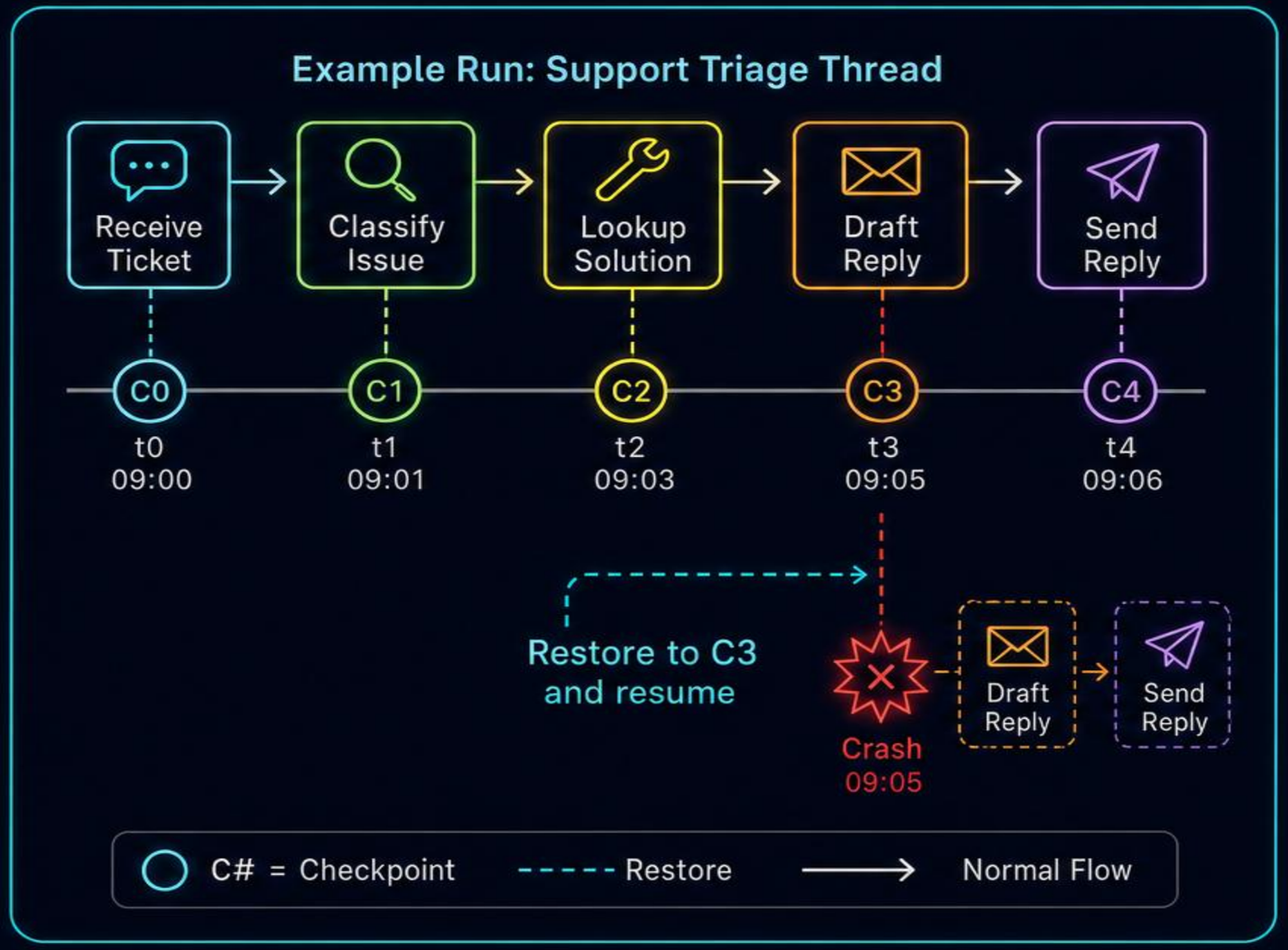
# Checkpointers: Thread Memory

A checkpointer saves a snapshot of state at every super-step.

**It gives you continuity,**
so a thread resumes
right where it paused

**It powers human-in-the-loop**
pauses and time-travel replay

**It adds fault tolerance**
if a run crashes mid-task



# Checkpointer vs. Store



## Checkpointer

Thread-scoped short-term persistence that saves state at every super-step. It keeps a single conversation continuous and resumable, and powers human-in-the-loop and time-travel.

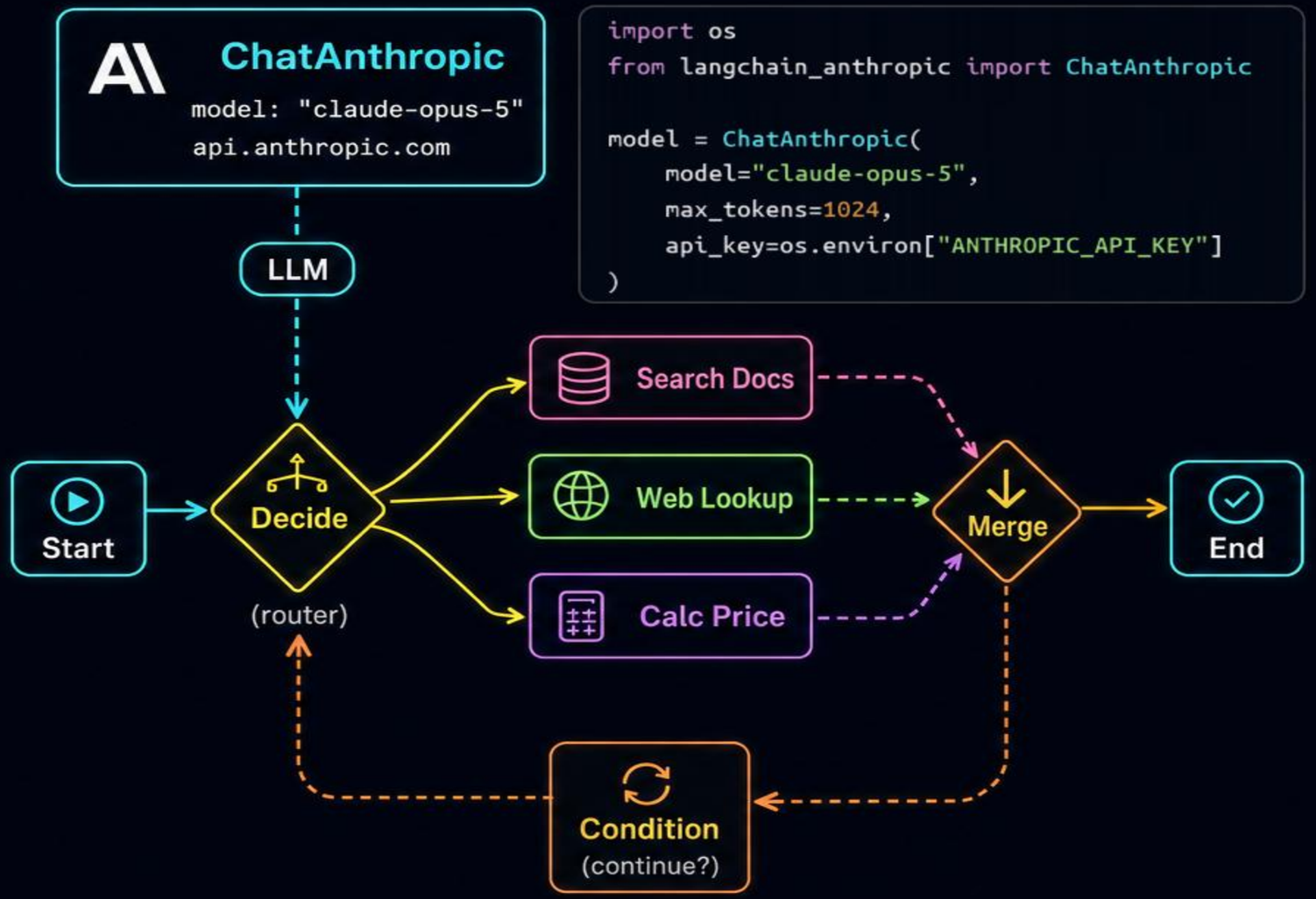


## Store

A separate system for cross-thread long-term memory that lives outside the graph state entirely. Use it for facts that should follow a user across different conversations.

# Wiring Claude into LangGraph

LangGraph is model-agnostic, so you choose the model per node.




```python
import os
from langchain_anthropic import ChatAnthropic

model = ChatAnthropic(
    model="claude-opus-5",
    max_tokens=1024,
    api_key=os.environ["ANTHROPIC_API_KEY"]
)
```

## Bring Claude in

through ChatAnthropic from the langchain-anthropic package

## Execution follows

a Pregel-style loop of discrete super-steps

## Parallel nodes share

a super-step, and cycles let the graph loop

# When to Reach for LangGraph



## StateGraph plus a typed State schema drive execution

The graph's state schema defines all keys, types, and invariants for safe updates.



## Nodes return partial updates, edges route control

Each node emits updates to state; edges evaluate conditions to choose the next node.



## Reducers decide how updates merge into state

Reducers implement merge logic per key, ensuring consistent, deterministic state.



## You compile() the graph before running it

Compilation validates the graph, resolves references, and optimizes execution.

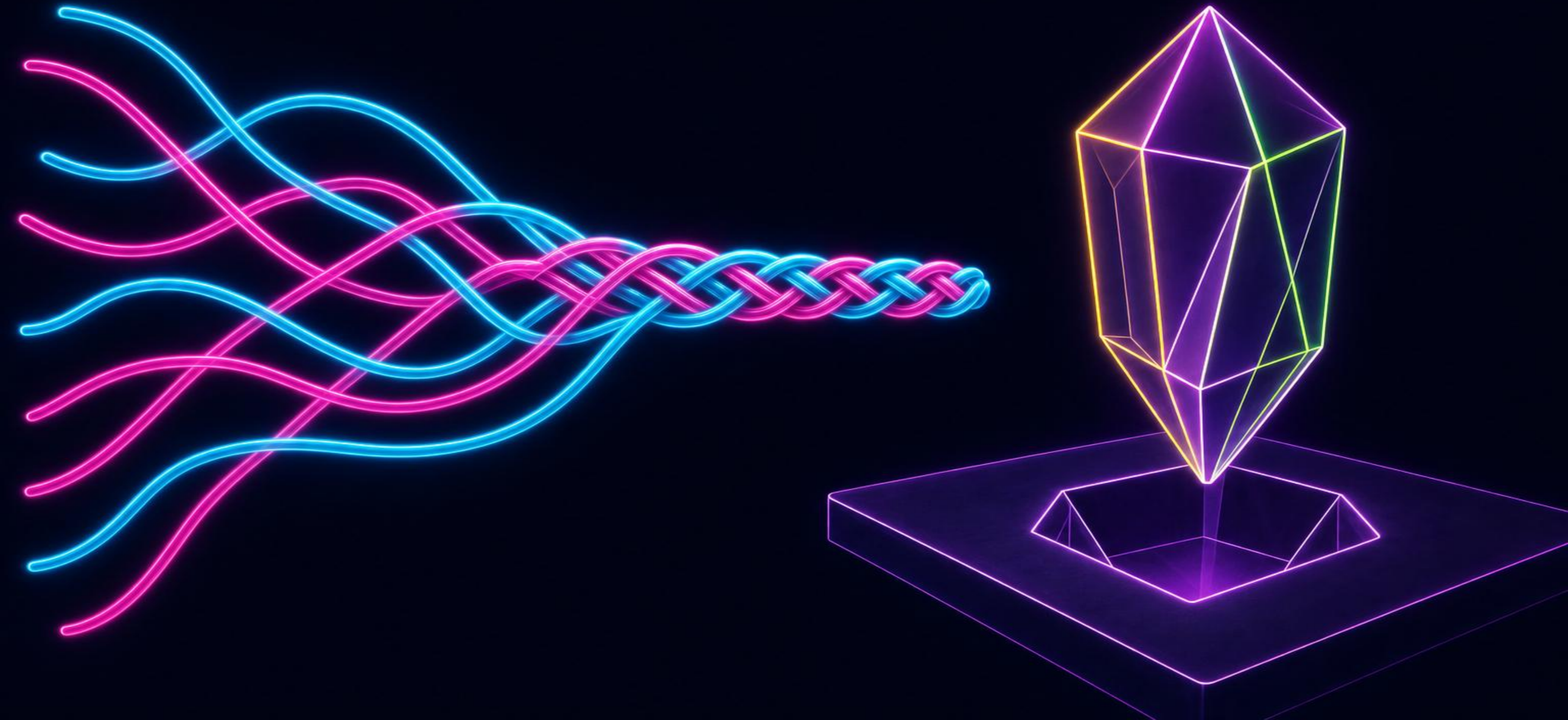


## Checkpointers persist a thread, the Store holds long-term memory

Checkpointers capture resumable snapshots per thread; the Store retains durable data.

# Strands and PydanticAI

Not every agent needs an explicit graph. Sometimes you want the model to drive its own steps.



# Strands in Three Parts



## Model

The LLM at the center of the loop. Bedrock is the native default, and the Anthropic API is supported.



## Tools

Plain Python functions marked with the @tool decorator, which auto-generates the spec from type hints.



## Prompt

The system prompt that frames the task and shapes how the model plans and acts.

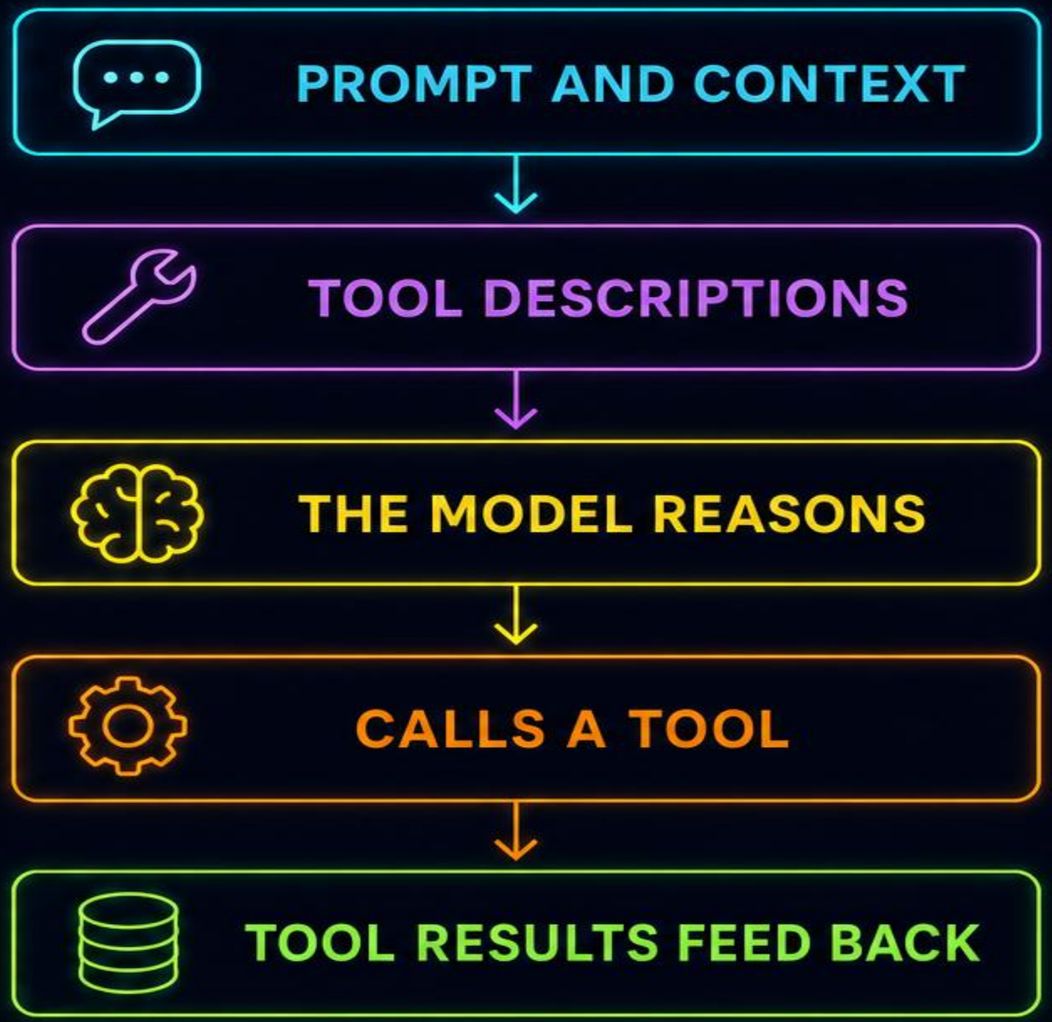
# How the Model Drives

Strands takes a model-driven approach built on the ReAct idea.

**Each turn,** it sends the prompt, context, and tool descriptions to the model

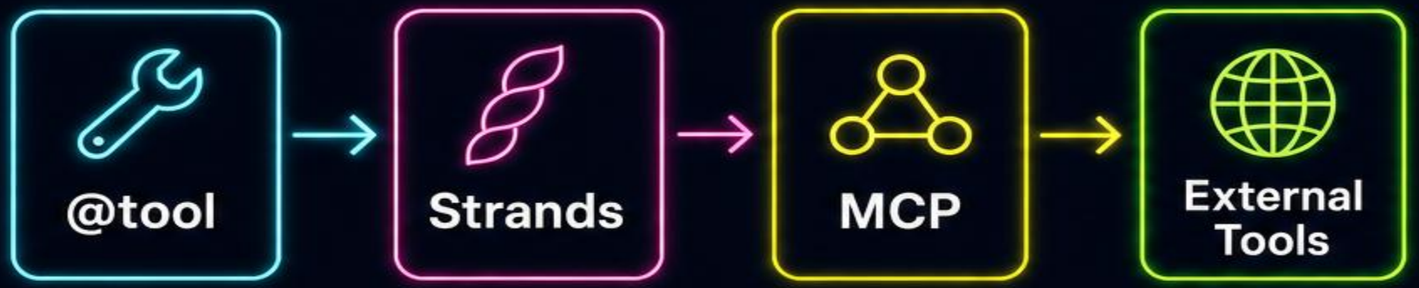
**The model** reasons, then answers or calls a tool

**Tool results** feed back, and the loop iterates until the task is done



# Strands Tools and Providers

Strands keeps tool definitions and model choice deliberately simple.




**Decorate a function** with @tool and the spec is built from your hints

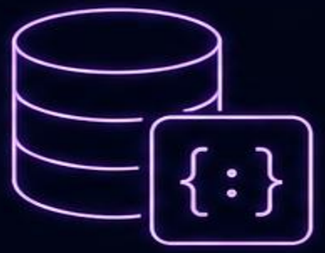
**It consumes** MCP servers for external tools

**Providers include** Bedrock as native default, plus the Anthropic API

</> TOOL DEFINITION

```
from anthropic import Anthropic
client = Anthropic()
tool = {
  "name": "get_weather",
  "description": "Get the current weather for a city",
  "input_schema": {
     "type": "object",
     "properties": {"city": {"type": "string", "description": "City name, e.g., San Francisco"}},
     "required": ["city"],
  }}
```

# Strands State Management




## Agent state

A key-value store that lives outside the model context. Your tools and app logic read and write it directly.



## Conversation manager

Handles the message history, using a sliding-window manager as its default strategy.



## Session manager

Persists and restores a conversation and its state, to a file or cloud storage.

# PydanticAI Building Blocks



## Agent

The container for instructions, tools, a model, and dependencies. Generic in its deps and output types.




## output_type

The parameter that names the validated output type. Pydantic checks the result against it.




## RunContext

How tools and functions receive typed dependencies, usage, and retry information at run time.

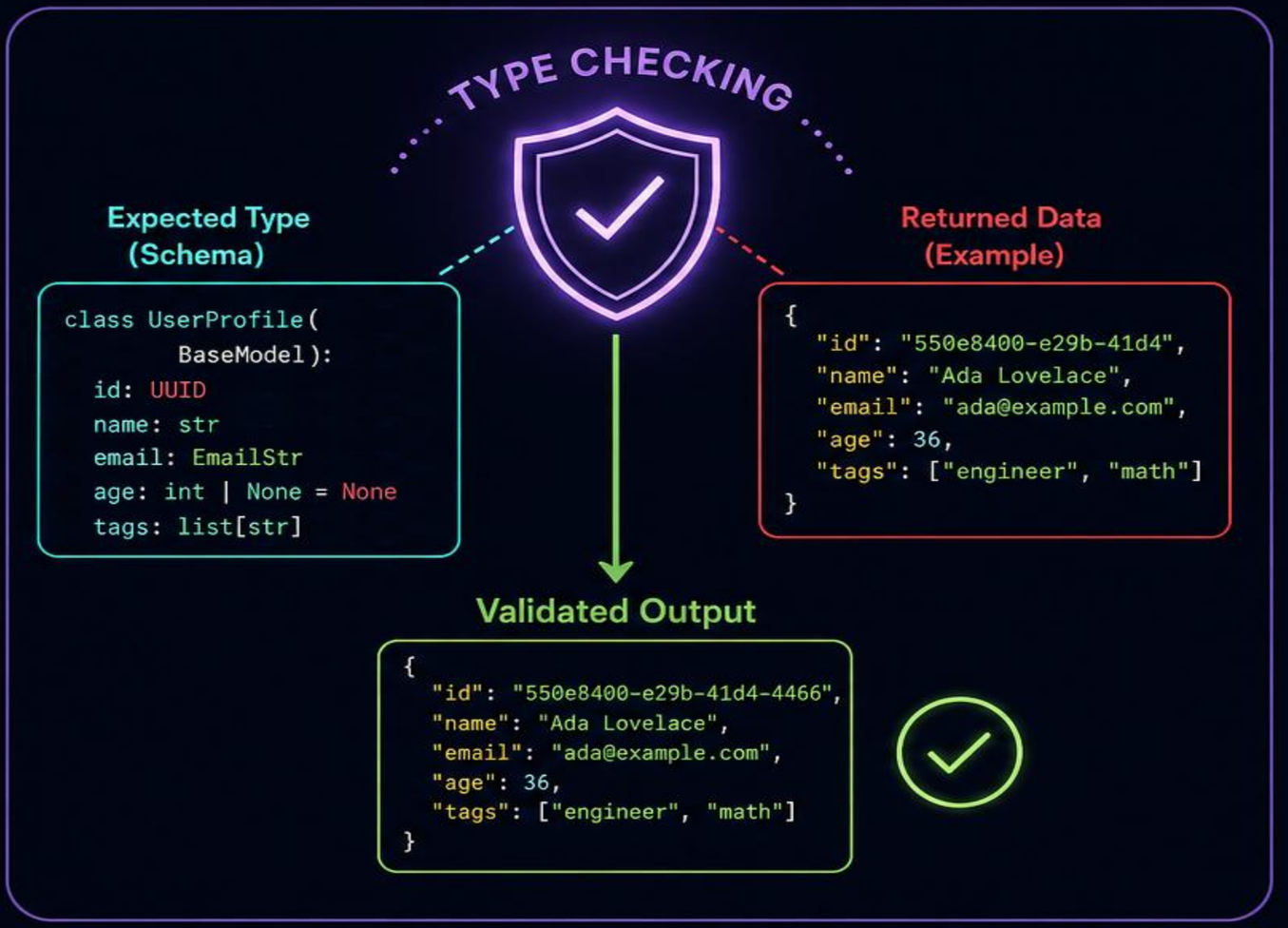
# Typed, Validated Outputs

PydanticAI's differentiator is type-safe, validated output.

**You set output_type,**
and Pydantic validates
the result against it

**The returned result**
is statically typed from
that output type

**Whole classes of errors**
move from run-time
to write-time



# PydanticAI Tools and Claude

Tools and model access stay typed and explicit here too.

RETRY HANDLER

```python
import time
from anthropic import Anthropic

client = Anthropic()

for attempt in range(4):
    try:
        resp = client.messages.create(
            model="claude-opus-5",
            max_tokens=1024,
            messages=[{"role": "user", "content": "Hello"}],
        )
        break
    except Exception as e:
        if attempt == 3:
            raise
        sleep_for = 2 ** attempt
        time.sleep(sleep_for)
```



### @agent.tool receives RunContext, @agent.tool_plain does not

RunContext provides per-call state, dependencies, and config that typed tools can access.



### A tool can raise ModelRetry to ask the model to try again

Use ModelRetry to signal "not now" or "need more info" so the model can refine its plan.



### Reach Claude via AnthropicModel, plus Bedrock and Vertex

PydanticAI supports multiple providers through a consistent model interface.

# Same Loop, Different Emphasis



## Strands

A lean model-driven loop of model, tools, prompt.
Minimal scaffolding and Bedrock-native defaults,
with a ReAct-style loop that just runs.
Best when you want an agent up fast.



## PydanticAI

Also model-driven, but built around typed,
validated outputs and dependency injection.
Best when the correctness of the structured
result is what you can't compromise.

# Model-Driven, Not Graph-Based



**Strands** is model, tools, and prompt, Bedrock-native and lean



**It uses @tool** functions and manages state outside the context



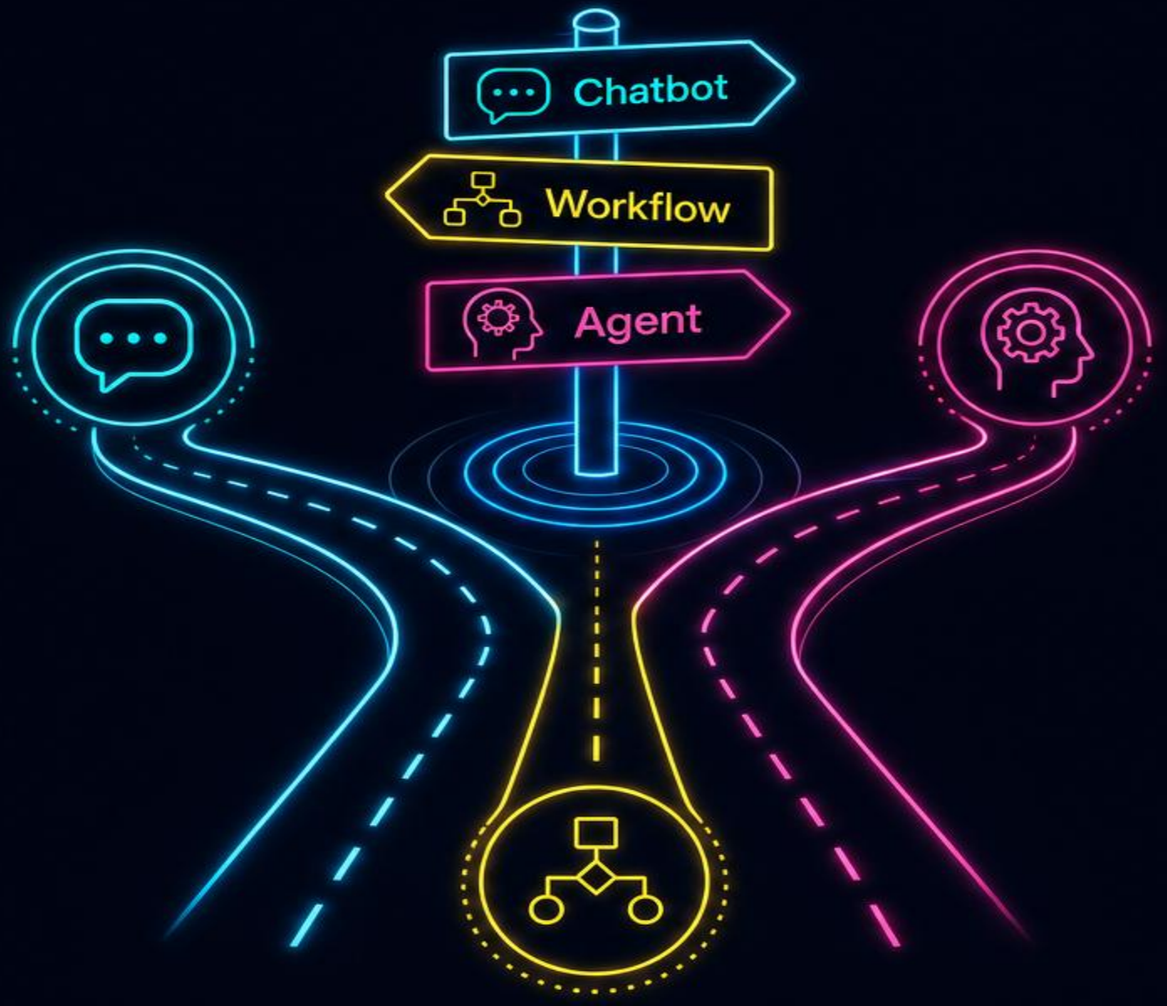
**PydanticAI** is model-driven with typed, validated outputs



**output_type and RunContext** give it write-time type safety

# Choosing a Framework

You've met all three frameworks. Now the real skill is picking the right one and keeping their identities straight.






# Three Frameworks, Three IDs



## LangGraph

The graph-based one. An explicit state machine with StateGraph, nodes, edges, reducers, and checkpointers.



## Strands

A model-driven loop of model, tools, and prompt. It is not graph-based, and not defined by typed output.



## PydanticAI

Also model-driven, but differentiated by its type-validated, structured outputs and full type safety.

# Reach for LangGraph When

Pick LangGraph when your control flow is the hard part.

**You need an explicit state machine with visible transitions**

Complex flows need precise states, guards, and transitions you can see and test.

**You want checkpointing, human-in-the-loop, or time-travel**

Pause, resume, inspect history, and roll back to any prior state safely.

**Cycles, retries, and branching belong in code, not prompts**

Express loops, fallbacks, and parallel paths with code-level control and robust error handling.

**If flow is complex and must be inspectable, this is your framework.**

# Reach for Strands When

Pick Strands when you want a lean, model-driven agent fast.

## Strands

Lean - Model-driven - Fast




### Live Example

```
{
   "task": "Get Q1 revenue for Acme Corp and summarize trends",
   "tools": ["athena_query", "s3_read", "calculate_trend"],
   "model": "claude-opus-5",
   "context": { "region": "us-east-1", "account": "123456789012" }
}
```

### You'd rather the model direct its own steps than draw a graph

Let the model decide the path in real time instead of predefining every step.

### You're on AWS and want Bedrock-native defaults

Strands uses Bedrock-native settings for auth, models, and tool integrations.

### Minimal scaffolding beats fine-grained control for your task

Ship a working agent quickly with less setup and maintenance.

# Reach for PydanticAI When

Pick PydanticAI when the shape of the output has to be correct.

### You need validated, typed structured results every time

PydanticAI guarantees the output matches your schema - or it fails.

### You want errors caught at write-time, not in production

Schemas are enforced immediately, so bad data never ships.

### Typed dependency injection fits how your team already builds

Providers, tools, and configs are strongly typed and easy to compose.

## Typed Output Validation in Action




**Schema** (Pydantic)

```python
from pydantic import BaseModel, Field
from typing import List

class CurrentWeather(BaseModel):
    temperature_c: float = Field(..., ge=-100, le=60)
    condition: str
    humidity_pct: int = Field(..., ge=0, le=100)
    wind_kph: float = Field(..., ge=0)

class WeatherReport(BaseModel):
    location: str
    retrieved_at: str  # ISO 8601
    current: CurrentWeather
    forecast: List[CurrentWeather]
```

**Example: Valid Output**

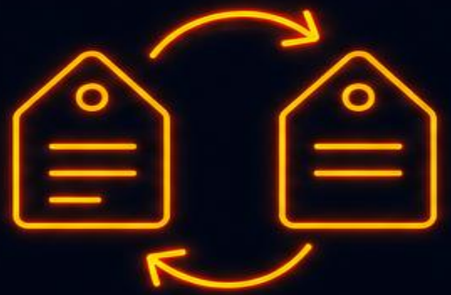
```json
{
  "location": "San Francisco, CA",
  "retrieved_at": "2025-05-23T10:30:00Z",
  "current": {
    "temperature_c": 18.2,
    "condition": "Partly cloudy",
    "humidity_pct": 72,
    "wind_kph": 12.4
  },
  "forecast": [
    {"temperature_c": 19.0, "condition": "Sunny", "humidity_pct": 68, "wind_kph": 10.1},
    {"temperature_c": 17.5, "condition": "Cloudy", "humidity_pct": 75, "wind_kph": 11.3}
  ]
}
```

**Example: Invalid Output (Rejected)**

```json
{
  "location": "San Francisco",
  "retrieved_at": "yesterday",
  "current": {
    "temperature_c": "cold",
    "condition": 123,
    "humidity_pct": 150,
    "wind_kph": -5
  }
}
```

If the output doesn't match the schema, PydanticAI raises a ValidationError.

# The Cross-Attribution Trap



## StateGraph is LangGraph

Only LangGraph has StateGraph, reducers, and checkpointers. Never attribute those primitives to Strands or PydanticAI.



## Typed output is PydanticAI

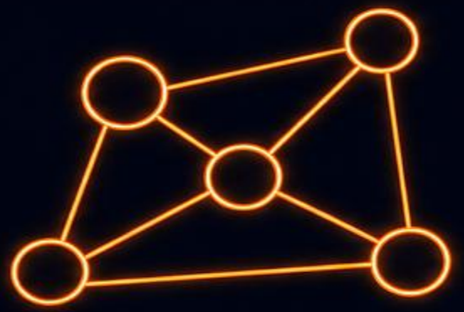
Type-validated, structured output is PydanticAI's signature. Strands does not build its identity on it.



## Strands is neither

Strands is purely model-driven, neither graph-based nor defined by typed, validated outputs at all.

# Graph-Based vs. Model-Driven



## Graph-based

LangGraph alone. You author the control flow yourself as an explicit graph, and Claude does the reasoning inside the nodes you have defined.

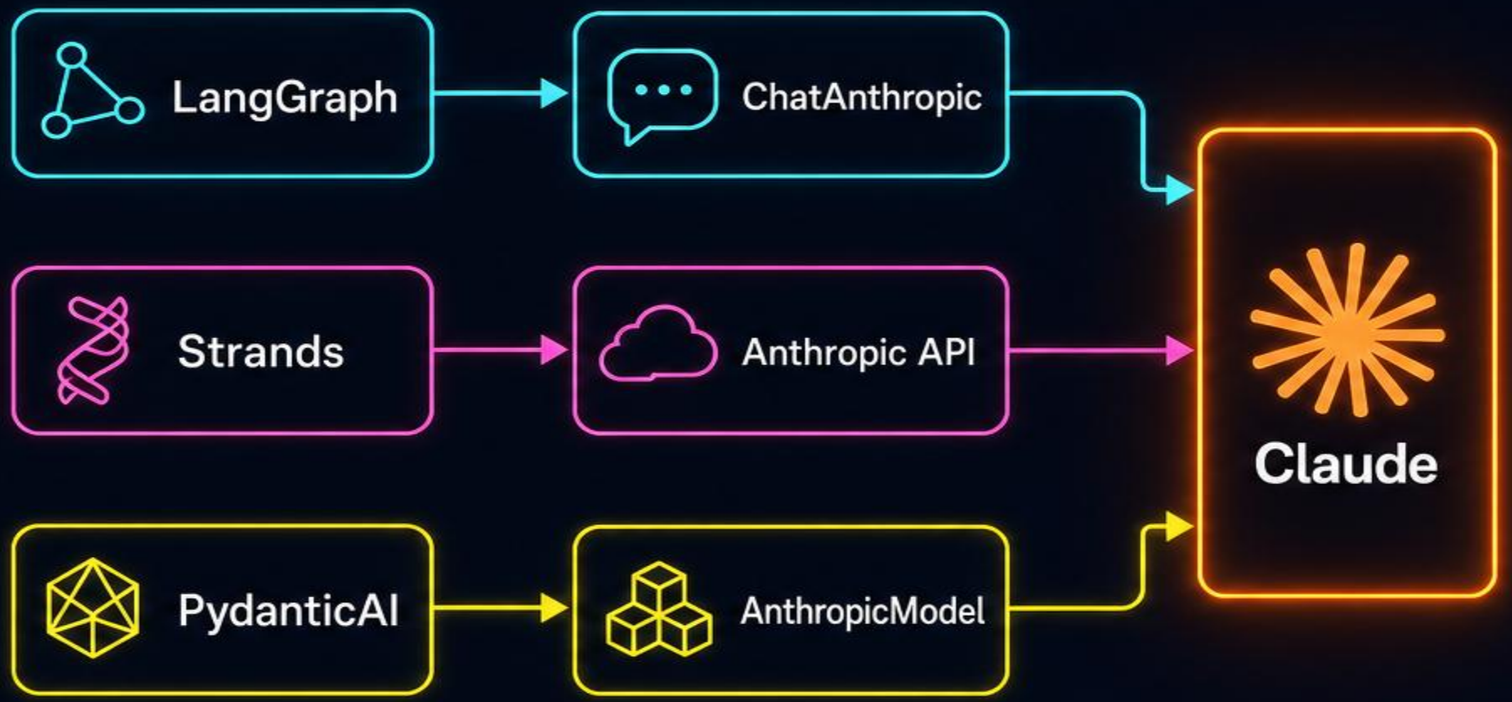


## Model-driven

Strands and PydanticAI. The model directs its own steps at run time, with no developer-authored graph guiding the path at all.

# All Three Wire to Claude

Framework choice never blocks you from using Claude.



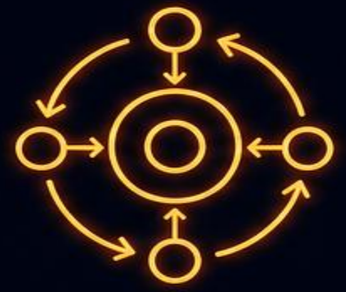

**LangGraph** brings Claude in through ChatAnthropic

**Strands** reaches Claude via the Anthropic API, alongside Bedrock

**PydanticAI** uses AnthropicModel, plus Bedrock and Vertex

**So pick on flow and output needs, not on Claude support.**

# Match the Build to the Tool



## Complex flow

When the control flow is the hard part, explicit state machines and inspectable transitions point you at LangGraph.



## Lean loop

When you want a minimal, model-driven agent up fast with little scaffolding, that points at Strands.



## Typed output

When correct, validated, write-time-safe results matter most, that points at PydanticAI.

# Picking the Right Framework



**Explicit graphs and checkpointing** point to LangGraph



**Lean model-driven loops** point to Strands



**Typed, validated outputs** point to PydanticAI



**Never cross-attribute primitives** between the three