

ANTHROPIC SOLUTION ENGINEERING

# Building Intelligent Agents with Claude Agent SDK

*Architecting the "Rei" Engine in Python*

PYTHON SDK

AGENTIC WORKFLOW

ENTERPRISE HARNESS

# The Shift: Chat vs. Agents

## Stateless Chat

Standard API interactions are stateless. You send a prompt, you get a completion. Logic lives in your application code.

- ✗ Complex state management on client
- ✗ Tool execution requires external loops
- ✗ Hard to "contain" mid-flight

## The Agentic Model (SDK)

The SDK brings the "Loop" inside. It allows Claude to:



**Plan & Reason**



**Execute Tools (In-Process)**



**Adhere to Safety Hooks**

# What is the Claude Agent SDK?

It is not just a wrapper around the Messages API. It is the **engine** that powers Claude Code.

It provides a standardized harness for building agents that can:

- Execute terminal commands safely
- Edit files with permissioning
- Maintain persistent context
- Fork sessions for exploration



## The Bundle

The Python SDK automatically bundles the **Claude Code CLI** binary.

No separate Node.js or binary installation required. `pip install` handles the binary fetch and setup.

# The Stack Decoded

To understand the SDK, we must distinguish between the components.



## 1. Python Script

**Your Entry Point.** This is the `main.py` you write. It imports the SDK and initiates the connection.



## 2. Python App

**The Wrapper.** The ecosystem hosting your script (e.g., a FastAPI backend or CLI tool).



## 3. The Agent

**The Logic.** The `ClaudeSDKClient` instance. It holds state and manages the loop.

## 4. The CLI Binary (The Engine)

A pre-compiled binary (managed by the SDK) that runs as a subprocess. It handles the heavy lifting: API communication and sandboxed execution.



# Visualizing the Harness

The SDK acts as a bridge between your application logic and the Anthropic execution engine.



## Control Protocol

A JSON-RPC style protocol flows over the transport layer, handling interrupts, permissions, and tool calls.

## Isolation

If the Agent generates a "crash" (e.g., runs a bad bash command), it happens in the Subprocess, protecting your main Python App.

# Getting Started

*Installation & Basic Concepts*

# Installation & Prerequisites

## Requirements

- Python 3.10+
- Valid Anthropic API Key

## Setup

```
pip install claude-agent-sdk
```

*Note: The SDK will automatically download the platform-specific Claude Code CLI binary during the first run or build process.*

## Environment Variables

```
export ANTHROPIC_API_KEY="sk- ... "
```

The SDK uses this key to authenticate the underlying bundled CLI against the Anthropic API.

# Hello World: The query() Function

---

For simple, one-shot interactions where you don't need complex state management.

```
import anyio
from claude_agent_sdk import query, AssistantMessage, TextBlock

async def main():
    # Simple one-shot query
    async for message in query(prompt="What is 2 + 2?"):
        if isinstance(message, AssistantMessage):
            for block in message.content:
                if isinstance(block, TextBlock):
                    print(block.text)

anyio.run(main)
```

## Key Characteristics

- **Unidirectional:** Send prompt, stream response.
- ↗ **Stateless:** Each query is a fresh start.
- ⟳ **Async Iterator:** Yields messages as they arrive.

# The Control Center: ClaudeAgentOptions

This object controls the environment, permissions, and capabilities of the agent.

## Context & Identity

- `system_prompt`
- `cwd` (Working Directory)
- `model` (e.g., Sonnet 3.5)

## Safety & Limits

- `max_budget_usd`
- `max_turns`
- `permission_mode`

## Capabilities

- `allowed_tools`
- `mcp_servers`
- `hooks`

# Defining the Working Environment

Agents interact with the filesystem. Setting the `cwd` is critical for file operations.

```
from pathlib import Path

options = ClaudeAgentOptions(
    # Set the root of the agent's world
    cwd="/path/to/project",

    # Define the persona
    system_prompt="You are a DevOps engineer."
)

# Pass options to query or client
await client.query("Check git status", options=options)
```



## Sandbox Implications

By default, tools like `ls`, `grep`, or `Edit` will operate relative to this CWD.

The SDK ensures operations are grounded in this path.

# Interactive Agents

*The ClaudeSDKClient & State*

# The ClaudeSDKClient

---

## Bidirectional & Stateful

Unlike `query()`, the Client maintains a persistent connection to the agent process.

- ✓ Send follow-up messages.
- ✓ Interrupt generation.
- ✓ Change settings mid-flight.
- ✓ Receive tool events and hooks.

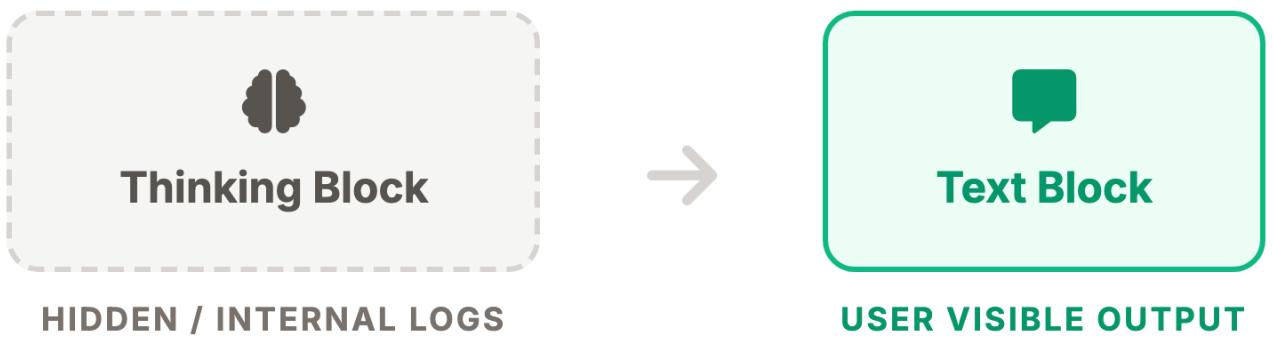
```
async with ClaudeSDKClient() as client:  
    # First turn  
    await client.query("Check the logs")  
    async for msg in client.receive_response():  
        print(msg)  
  
    # Second turn (maintains context)  
    await client.query("Fix the error you found")  
    async for msg in client.receive_response():  
        print(msg)
```

# Streaming Deltas

To build rich UIs, you need to show the agent "typing" via `StreamEvent`.

```
options = ClaudeAgentOptions(  
    include_partial_messages=True, # Enable deltas  
)  
  
async with ClaudeSDKClient(options=options) as client:  
    await client.query("Write a story ...")  
  
    async for msg in client.receive_response():  
        if isinstance(msg, StreamEvent):  
            # msg.event contains: message_start,  
            # content_block_delta, etc.  
            delta = msg.event.get('delta', {})  
            if delta.get('type') == 'text_delta':  
                print(delta['text'], end="", flush=True)
```

# Transparency: Auditing "Thinking"



Agents using **Extended Thinking** models produce reasoning steps before the final answer.

The SDK exposes these as **ThinkingBlock** objects, allowing you to audit the agent's logic chain.

```
if isinstance(msg, AssistantMessage):
    for block in msg.content:
        if isinstance(block, ThinkingBlock):
            # Hidden reasoning
            logger.info(f"Reasoning: {block.thinking}")

        elif isinstance(block, TextBlock):
            # Final output to user
            print(block.text)
```

# Control Flow: Interrupts

Agents can get stuck in loops or go down wrong paths. The SDK allows you to signal an interrupt to the underlying process.

This stops the generation but keeps the session alive for a new prompt.

```
# Start a long running task
await client.query("Count to 1 million")

# ... some logic decides to stop it ...
await asyncio.sleep(2)

# Send interrupt signal
await client.interrupt()

# Send new instruction
await client.query("Actually, just count to 10")
```

# Dynamic Runtime Control

You can change the agent's brain or permissions without restarting the session.

## Switch Models

```
await client.set_model(  
    "claude-3-5-haiku"  
)
```

Swap to a faster/cheaper model  
for simple sub-tasks.

## Change Permissions

```
await client.set_permission_mode(  
    "acceptEdits"  
)
```

Grant write access dynamically after a  
review phase.

## Budgeting

```
options.max_budget_usd = 0.50
```

Hard stop the agent if it burns too  
many tokens.

# Session Forking

**Session A (Original Context)**  
Read Repo • Analyze Bugs • Plan Fixes

FORK POINT

**Session B (New Branch)**

Try Fix Attempt #1 (Isolated)

## Exploration Pattern

Start a session, establish context (e.g., read a repo), and then "fork" the session to try multiple parallel experiments without re-reading the repo.

```
options = ClaudeAgentOptions(  
    resume="session-id-123",  
    fork_session=True  
)  
  
# This creates a NEW session ID but keeps  
# the history of session-123.  
async with ClaudeSDKClient(options) as client:  
    await client.query("Try approach A")
```

# Tools & MCP

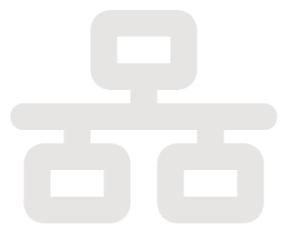
*The Agent's Hands and Eyes*

# Model Context Protocol (MCP)

MCP is the standard way to connect AI models to data and tools.

Instead of hardcoding API calls, you expose "Servers" that provide:

- ✓ **Tools:** Executable functions (e.g., "query\_db")
- ✓ **Resources:** readable data (e.g., "logs://")
- ✓ **Prompts:** Reusable templates



Standardized Interface

# Game Changer: In-Process MCP

Traditionally, MCP servers run as separate processes (stdio/SSE). The SDK allows you to run MCP servers **directly inside your Python application.**



## No Overhead

No IPC serialization lag. Function calls are direct Python calls.



## Shared State

Tools can access your app's global variables, database connections, and memory.



## Simple Debugging

Set breakpoints in your tool code and step through it alongside your agent logic.

# Tooling Pattern: Input Sanitization

You can intercept tool inputs before execution to "fix" or redirect them transparently.

This is crucial for sandboxing writes or preventing agents from touching forbidden directories without breaking the flow.

```
async def permission_callback(tool, input, context):
    if tool == "Write" and "forbidden" in input["file_path"]:
        # Redirect to safe location silently
        new_input = input.copy()
        new_input["file_path"] = "/tmp/safe_write.txt"

    return PermissionResultAllow(
        updated_input=new_input
    )
```

# Architecture: External vs. In-Process

## External (Standard MCP)

**Mechanism:** Spawns a new process. Communicates via Stdio pipes.

- ✓ Isolated environment
- ✓ Language agnostic
- ✗ High latency (IPC)
- ✗ Hard to share state

## In-Process (SDK)

**Mechanism:** Registers Python functions directly within the SDK client memory.

- ✓ Zero Latency
- ✓ Shared Memory Access
- ✓ Easy Debugging
- ! Python only (for now)

# Why use In-Process MCP?

For Solution Engineers integrating Claude into existing platforms, In-Process is often superior.



## Reuse Connections

Don't open a new DB connection for every tool call. Use your existing SQLAlchemy session.



## Speed

Eliminating JSON serialization/deserialization over pipes saves milliseconds per turn.



## Complexity

No need to package/dockerize a separate "server". It's just a module in your app.

# Implementation: The `@tool` Decorator

---

Creating an In-Process server is as simple as decorating a Python function.

```
from claude_agent_sdk import tool, create_sdk_mcp_server

# 1. Define the tool
@tool("get_user_data", "Fetch user info", {"user_id": str})
async def get_user_data(args):
    # Direct access to your app logic
    user = await db.fetch_user(args['user_id'])
    return {
        "content": [{"type": "text", "text": str(user)}]
    }

# 2. Create the server
server = create_sdk_mcp_server(
    name="internal-api",
    tools=[get_user_data]
)
```

## Key Concepts

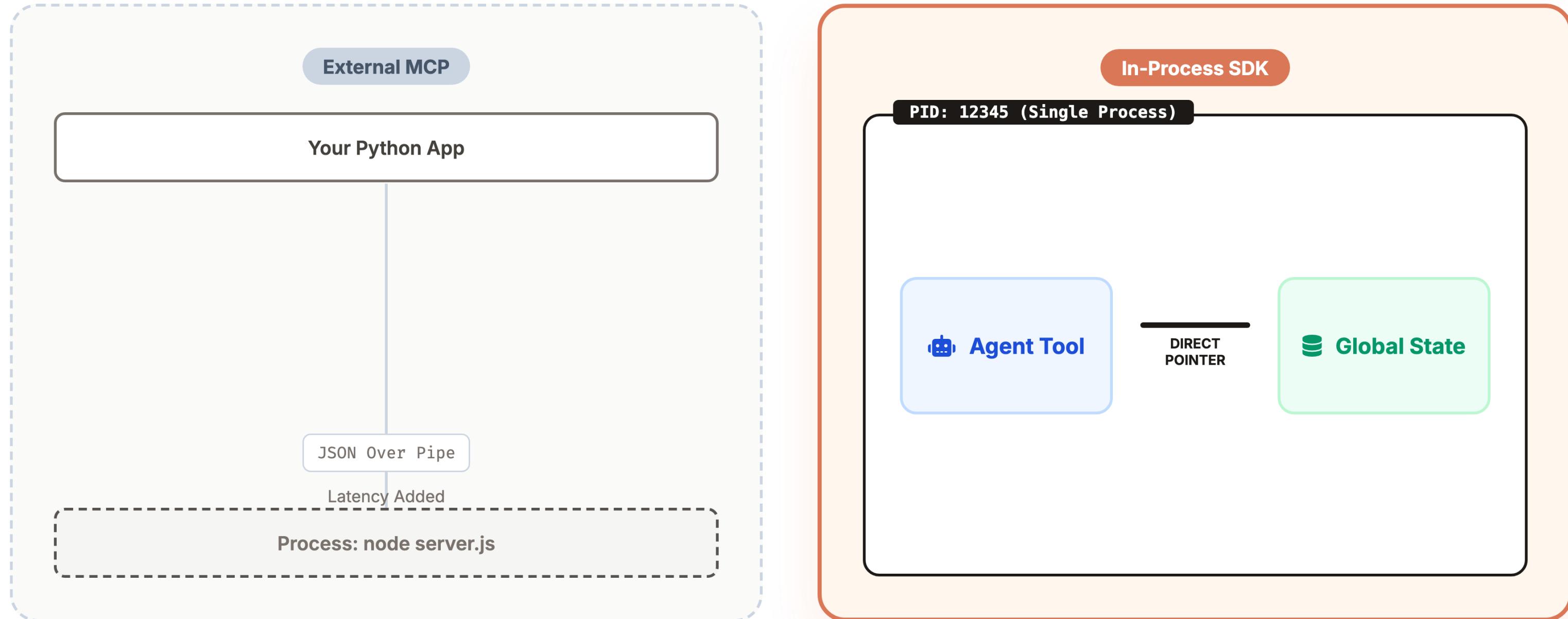
- **Type Hints:** Used for validation.
- **Async:** Tools must be async functions.
- **Return Format:** Standard MCP content structure.

# Pattern: Shared State Access

The killer feature: Your tool can see your application's memory.

```
class ApplicationState:  
    def __init__(self):  
        self.feature_flags = {"beta_access": False}  
  
app_state = ApplicationState()  
  
@tool("toggle_feature", "Toggle a feature flag", {"feature": str})  
async def toggle_feature(args):  
    # Modify state directly!  
    feat = args['feature']  
    app_state.feature_flags[feat] = not app_state.feature_flags.get(feat, False)  
    return {"content": [{"type": "text", "text": "Flag updated"}]}  
  
# The agent effectively has "Root Access" to this specific state object.
```

# Concept: Zero-Latency State



# Real-World Examples

---

## Internal API Gateway

Expose your company's private microservices (User Service, Billing) as tools without building public wrappers.

## RAG & Vector DB

Pass your pre-loaded Vector Search client into the tool. Agent queries Pinecone/Weaviate directly.

## DevOps Control

Expose Kubernetes client methods to allow the agent to inspect pods running in the current cluster context.

# Registering the Server

---

Once created, you pass the in-process server to the Agent Options.

```
options = ClaudeAgentOptions(  
    # Register the server  
    mcp_servers={"internal_tools": server},  
  
    # Whitelist the tools you want available  
    # Format: mcp_{server_name}_{tool_name}  
    allowed_tools=["mcp_internal_tools_greet"]  
)  
  
async with ClaudeSDKClient(options=options) as client:  
    await client.query("Greet Alice")
```

# Hybrid Tooling Strategy

You can mix In-Process tools with External MCP servers.

## Internal (SDK)

- Business Logic
- Database Writes
- App Control

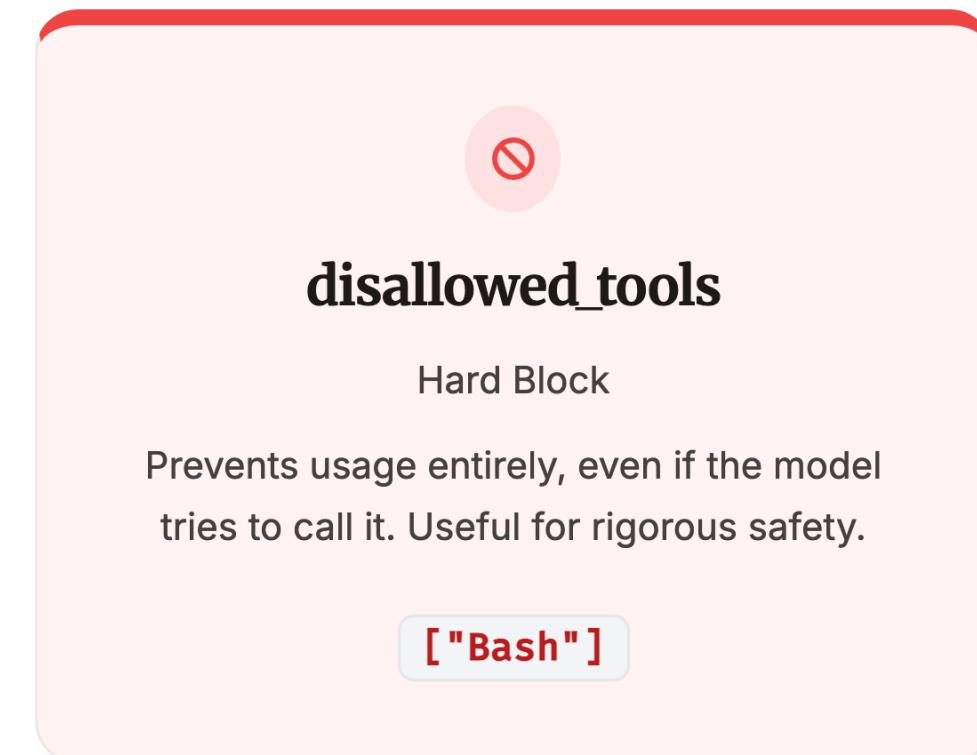
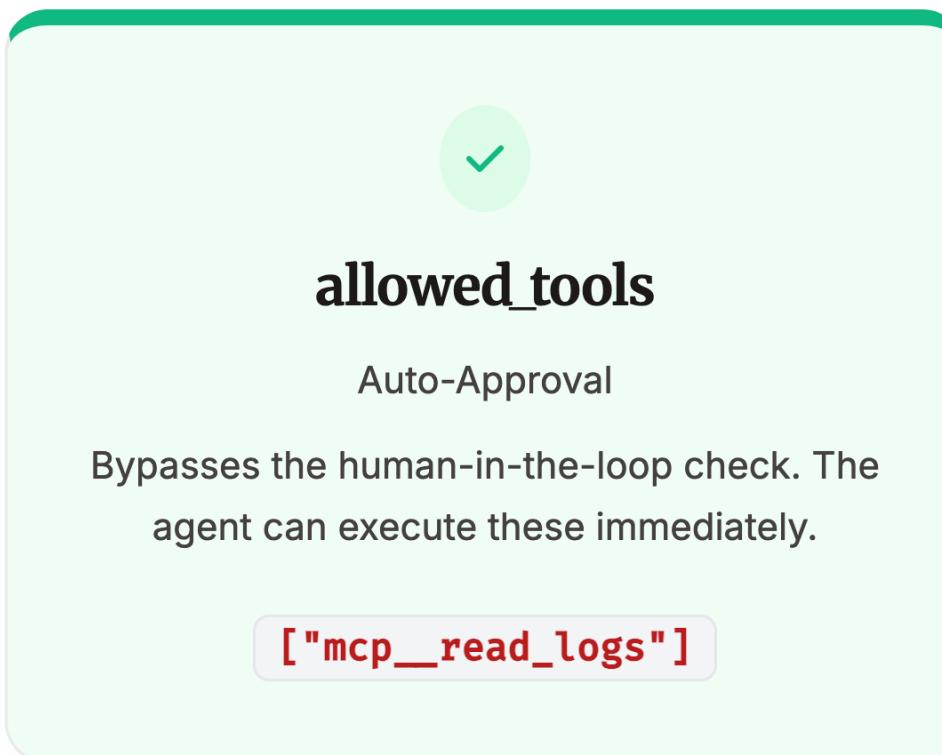
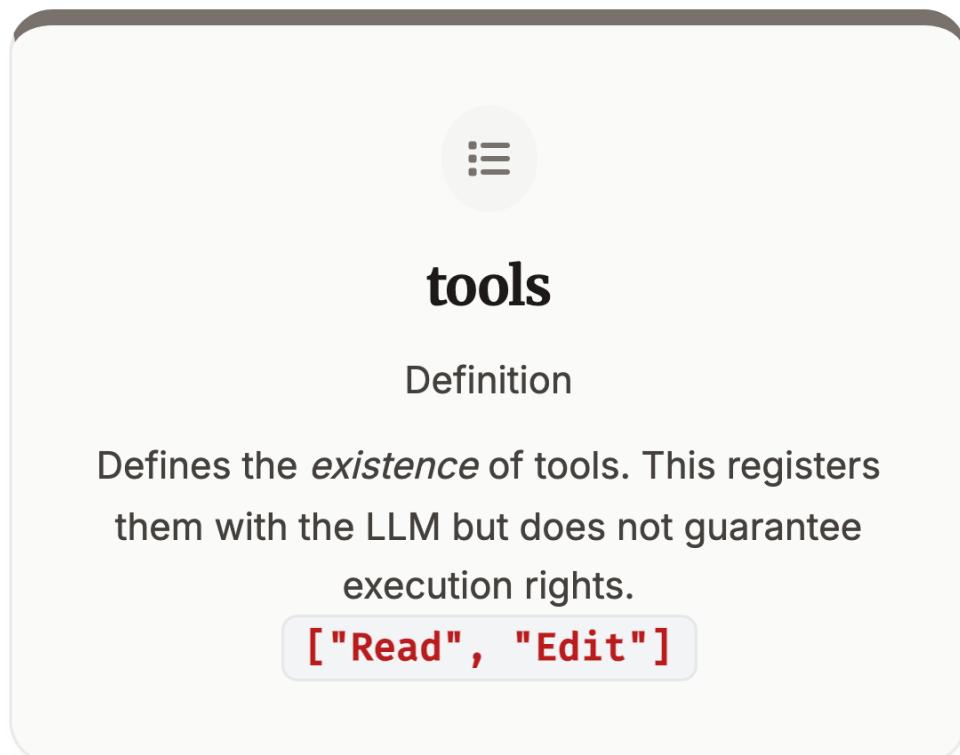
## External (Stdio/SSE)

- PostgreSQL MCP Server
- GitHub MCP Server
- Slack MCP Server

```
options = ClaudeAgentOptions(  
    mcp_servers={  
        "internal": sdk_server,  
        "github": {"type": "stdio", "command": "docker", "args": [ ... ] }  
    }  
)
```

# Controlling Tool Access

Just because a tool exists doesn't mean the agent should use it freely. The SDK offers three levels of granularity.



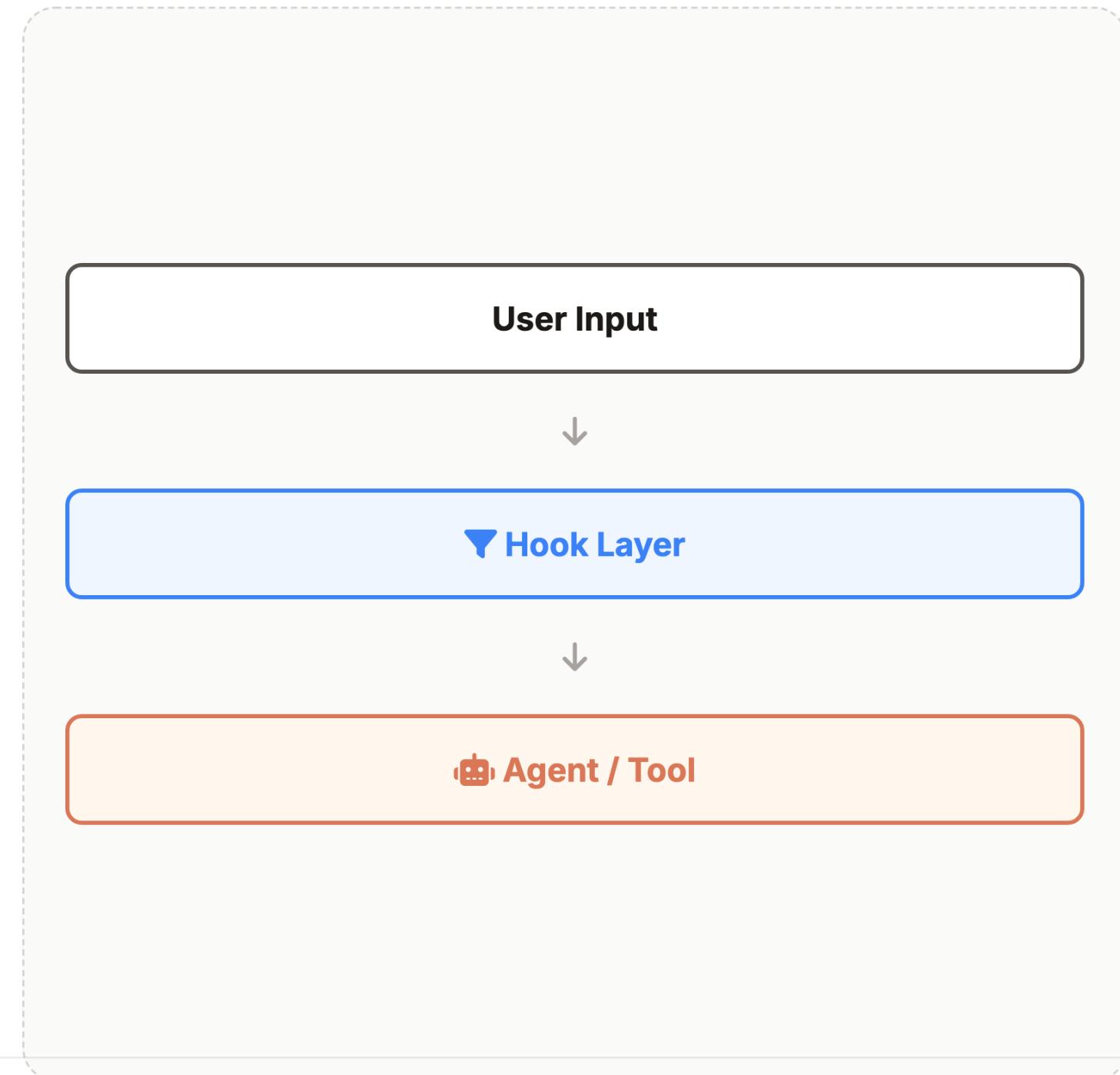
# The Harness

*Hooks, Safety & Permissions*

# The Safety Harness: Hooks

Hooks act as a **middleware layer** between Claude's intent and the actual execution.

- **PreToolUse:** Intercept before execution (Block/Allow).
- ✓ **PostToolUse:** Inspect results after execution.
- 👤 **UserPromptSubmit:** Modify user input before LLM.



# Human-in-the-Loop: Async Hooks

```
async def approval_hook(input, tool, ctx):
    # Keep connection open, pause loop
    return {
        "async_": True,
        "asyncTimeout": 300000
    }
    # App handles approval out-of-band
```

ENTERPRISE PATTERN

SLACK APPROVAL

Agent Request → PAUSED



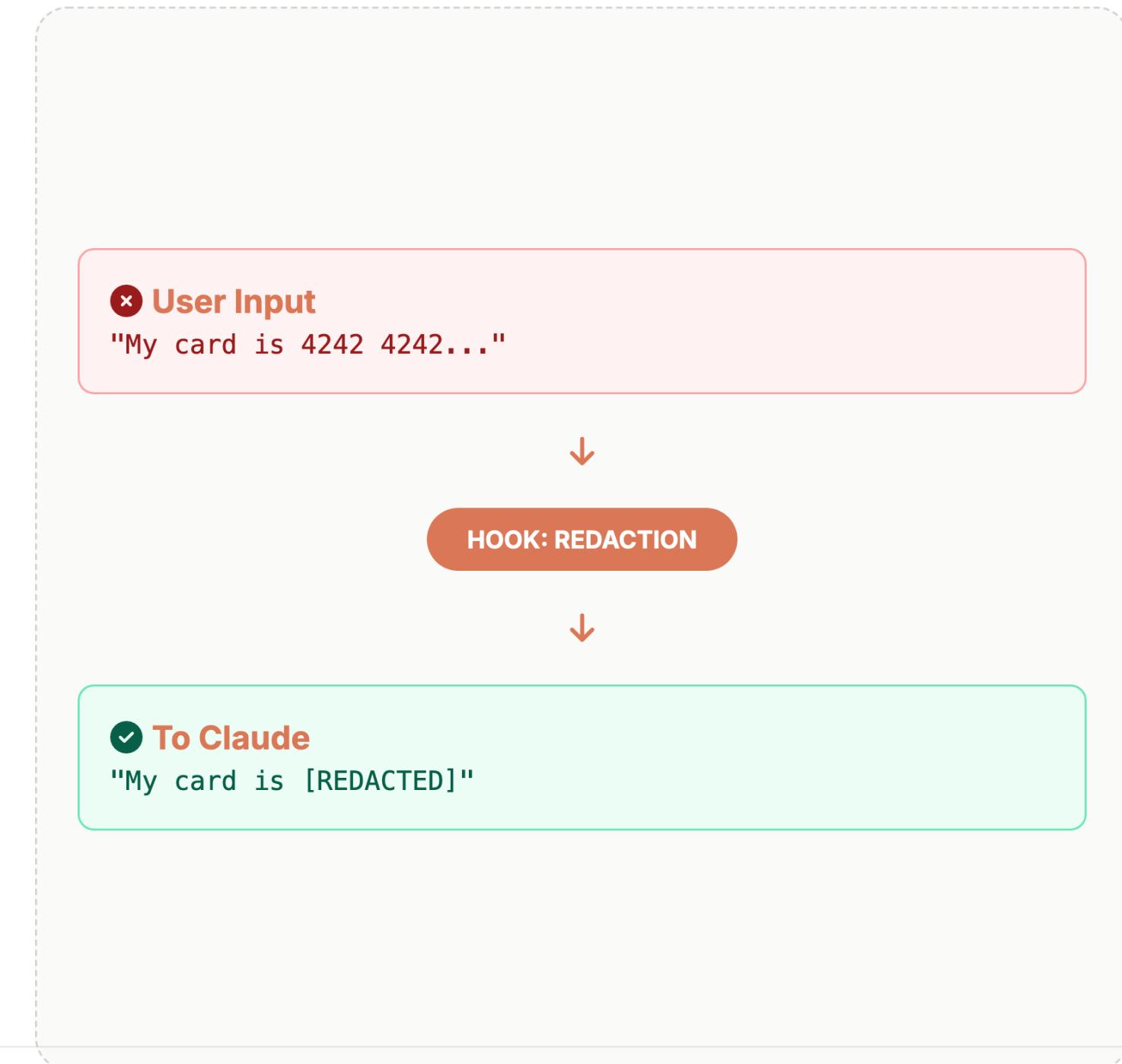
Human approves  
via Slack

RESUMED → Tool Runs

# Data Privacy: PII Redaction

Intercept user prompts **before** they reach the LLM to redact sensitive info.

```
async def redact_pii(input, tool, ctx):
    p = input["prompt"]
    # Regex to find Credit Cards
    cleaned = re.sub(
        r'\b(?:\d[ -]*?){13,16}\b',
        '[REDACTED_CC]', p
    )
    return {
        "hookSpecificOutput": {
            "hookEventName": "UserPromptSubmit",
            "prompt": cleaned
        }
    }
```



# Data Safety: Output Auditing

Use PostToolUse to ensure tools aren't leaking secrets to the context window.

```
async def audit_output(input, tool, ctx):
    res = str(input["tool_response"])

    if "sk-ant-" in res:
        return {
            "hookSpecificOutput": {
                "hookEventName": "PostToolUse",
                "tool_response": "[REDACTED]"
            }
        }
```

{"key": "sk-ant-123"}

RAW



{"key": "[REDACTED]"}

SAFE

# PreToolUse: Implementing Guardrails

Intercept a tool call before it executes to inspect arguments.

```
async def check_bash(input, tool, ctx):
    cmd = input["tool_input"].get("command")

    if "rm -rf" in cmd:
        return {
            "hookSpecificOutput": {
                "hookEventName": "PreToolUse",
                "permissionDecision": "deny"
            }
        }
```



> rm -rf /

**BLOCKED**

Policy Violation Detected

# PostToolUse: Validation & Feedback

Inspect output. If it failed, intervene with helpful context.

```
async def validate(input, tool, ctx):
    res = input["tool_response"]

    if "Error" in str(res):
        return {
            "systemMessage": "Tool failed.",
            "hookSpecificOutput": {
                "hookEventName": "PostToolUse",
                "additionalContext": "Check syntax."
            }
        }
```

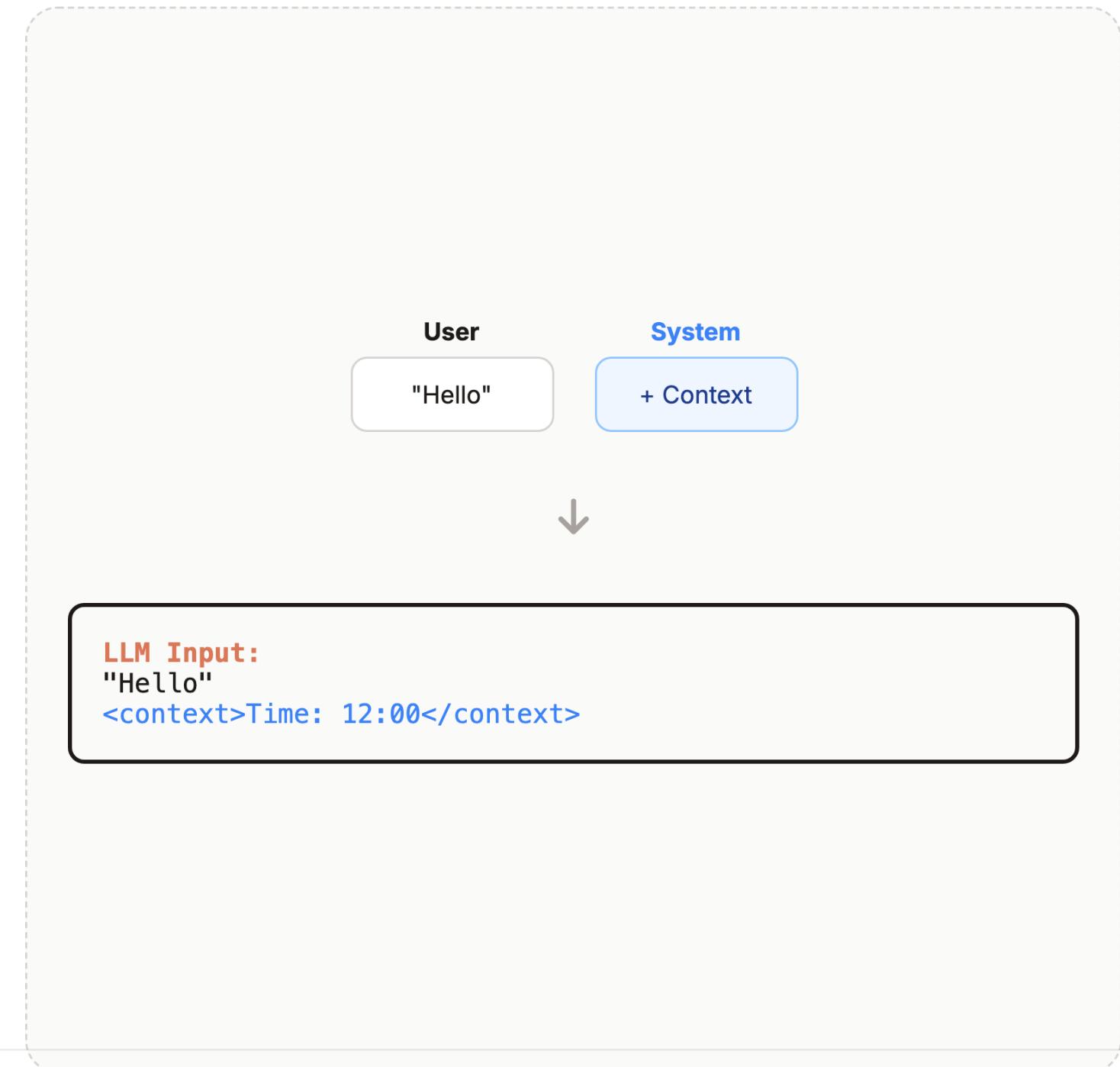
⚠ Tool Error → 🔎 Hook

💡 Feedback to Agent:  
"You made a syntax error. Try adding quotes."

# UserPromptSubmit: Context Injection

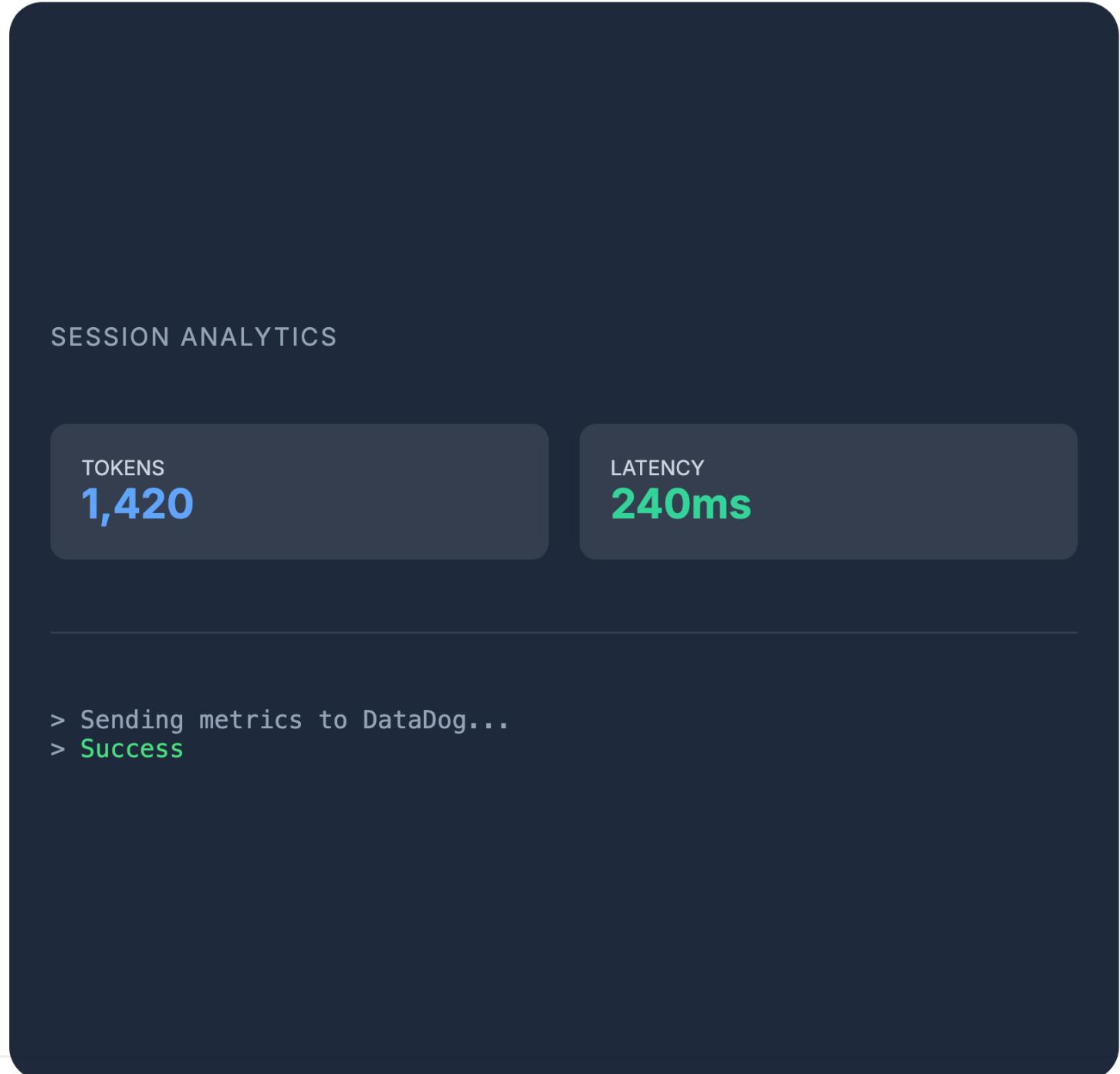
Inject secret context or instructions without the user typing them.

```
async def inject(input, tool, ctx):
    return {
        "hookSpecificOutput": {
            "hookEventName": "UserPromptSubmit",
            "additionalContext": "Time: 12:00."
        }
    }
```



# ROI & Usage Tracking

The **ResultMessage** contains precise cost and token usage data.



# Wiring it Together

Hooks are registered using **HookMatcher**.

```
options = ClaudeAgentOptions(  
    hooks={  
        "PreToolUse": [  
            # Only run for "Bash"  
            HookMatcher(  
                matcher="Bash",  
                hooks=[check_bash]  
            ),  
        ],  
    }  
)
```

## ⚙️ Configuration

- PreToolUse → check\_bash
- UserPrompt → inject\_context

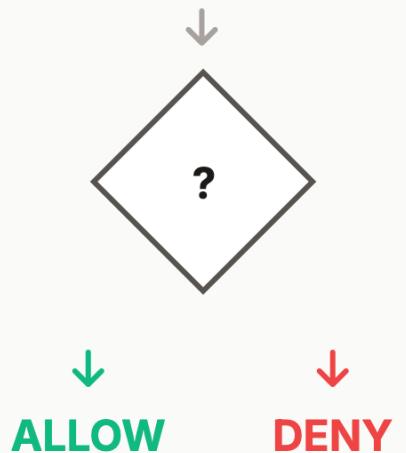
# Advanced: can\_use\_tool Callback

For complex logic beyond simple blocking.

```
async def perm_cb(tool, input, ctx):
    if tool == "Write" and "/etc" in input:
        return Deny("No system writes")

    # Redirect write
    if tool == "Write":
        new = input.copy()
        new["path"] = "/tmp/safe.txt"
        return Allow(new)
```

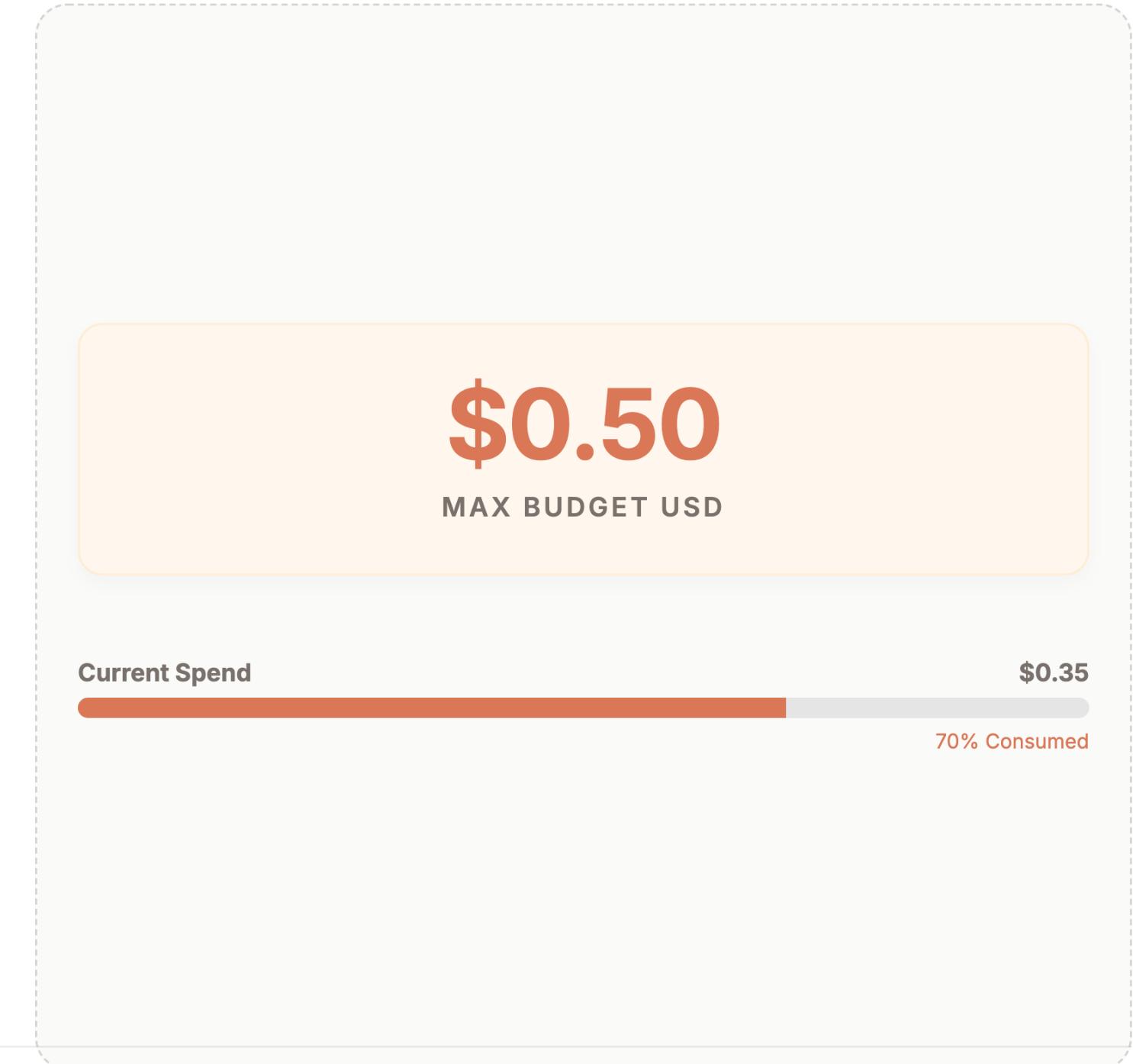
Incoming Tool Call



# Enterprise Safety: Budgeting

Prevent runaway agents from consuming infinite tokens.  
The SDK terminates the session if the limit is breached.

```
options = ClaudeAgentOptions(  
    max_budget_usd=0.50,  
    max_turns=20  
)
```



# Reliability: The Testing Pyramid

## Unit Tests

Test your `@tool` functions directly in Python.

## Integration

Mock the Transport layer to simulate Claude.

## E2E

Run real loops against a sandbox.

```
@pytest.mark.e2e
async def test_agent_completes_task():
    async with ClaudeSDKClient(options) as client:
        await client.query("Echo 'success'")
```

# Advanced Architectures

*Sub-Agents & Orchestration*

# Defining Sub-Agents

Claude can "handoff" tasks to named agents.

```
options = ClaudeAgentOptions(  
    agents={  
        "code-reviewer": AgentDefinition(  
            description="Reviews code",  
            prompt="You are an auditor.",  
            tools=["Read", "Grep"],  
            model="claude-3-opus"  
        )  
    }  
)
```



## code-reviewer

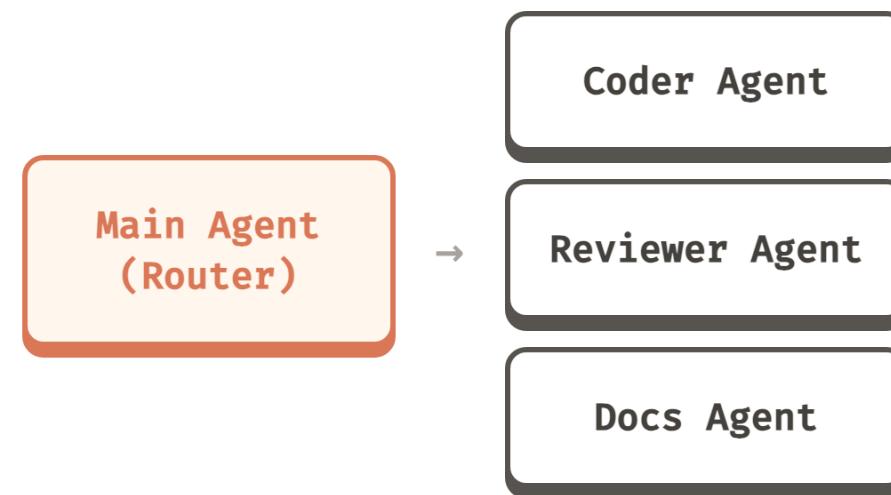
Specialized Sub-Agent

OPUS

STRICT MODE

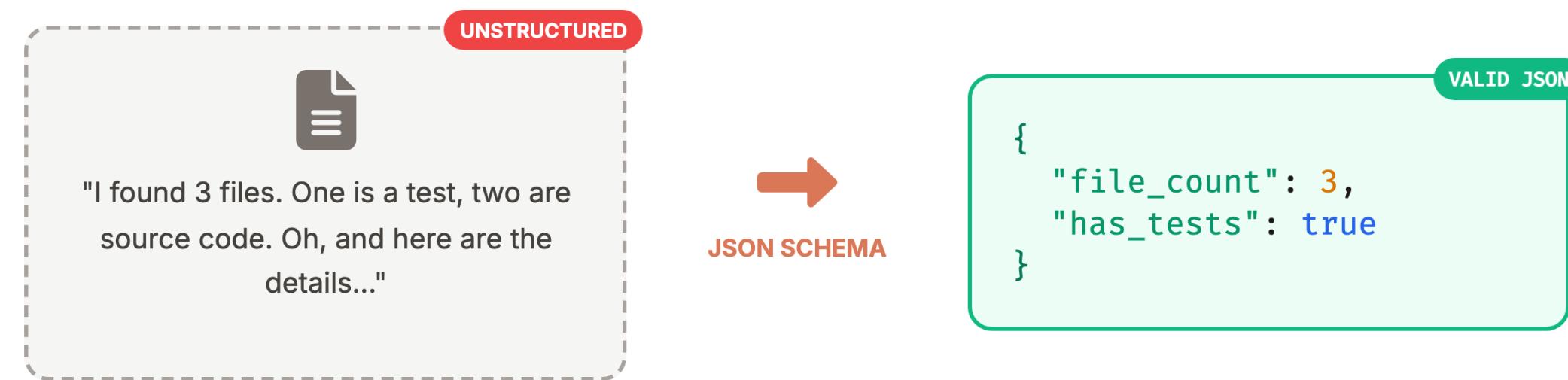
*"I only review code. I do not write it."*

# The Orchestrator Pattern



The Main Agent uses the "**agents**" tool (automatically provided by SDK) to delegate work to the definitions provided in **AgentDefinition**.

# Structured Output: From Chaos to JSON



The SDK forces the LLM to output only data that matches your strictly defined schema.

# Security: Network Sandbox

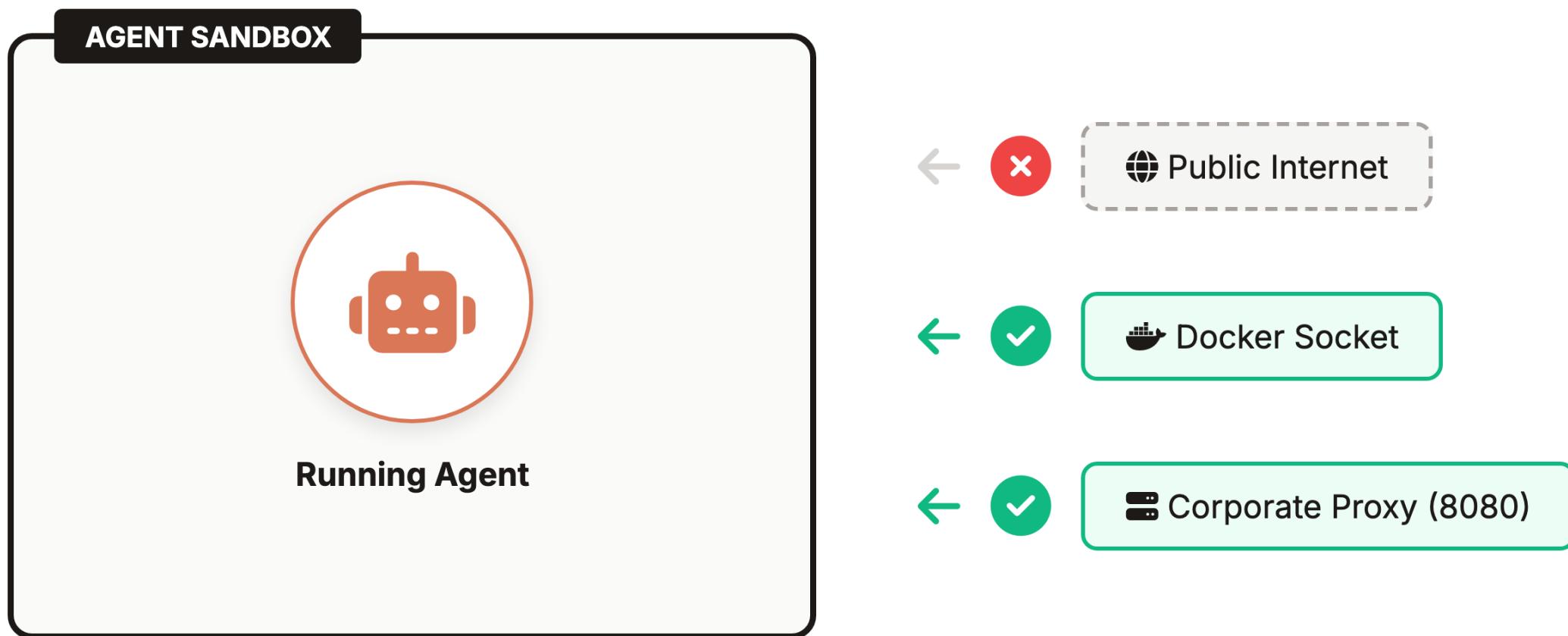
Enterprise deployments often require strict network isolation. The `SandboxSettings` object controls exactly what the agent can access.

- Block all internet access.
- Allow specific Unix sockets (e.g., Docker).
- Force traffic through proxies.

```
sandbox_config = {
    "enabled": True,
    "network": {
        "allowAllUnixSockets": False,
        "allowUnixSockets": ["/var/run/docker.sock"],
        "httpProxyPort": 8080
    }
}

options = ClaudeAgentOptions(sandbox=sandbox_config)
```

# Visualizing Network Isolation



# Robust Error Handling

Production agents must handle failures gracefully. The SDK provides specific exception types.

## **CLINotFoundError**

Agent runtime binary is missing or corrupt.

## **CLIConnectionError**

The transport layer failed (process crash).

## **ProcessError**

The underlying command exited with a non-zero code.

```
try:  
    async for msg in query( ... ):  
        pass  
except CLIConnectionError:  
    # Handle retry logic or fallback  
    logger.error("Agent transport died, restarting ... ")
```

# Settings Isolation

Control where Claude reads its configuration from. Vital for testing and secure environments.

## User

Global settings ( `~/.claude/` ).

## Project

Repo settings ( `./.claude/` ).

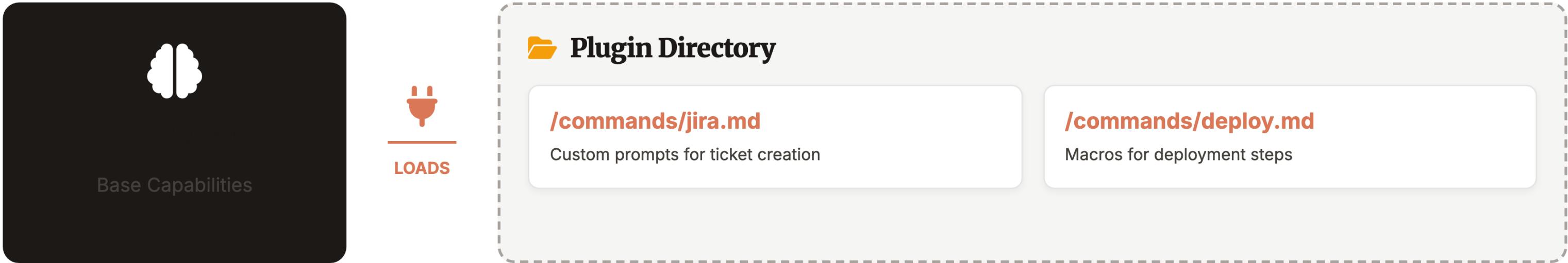
## Local

Git-ignored local overrides.

```
# Load NOTHING by default (Clean slate)
options = ClaudeAgentOptions(setting_sources=[])

# Load only User settings
options = ClaudeAgentOptions(setting_sources=["user"])
```

# Plugins Architecture



```
options = ClaudeAgentOptions(  
    plugins=[  
        {  
            "type": "local",  
            "path": "/path/to/my/plugins"  
        }  
    ]  
)
```

# Debugging: The Stderr Callback

---

Since the CLI runs in a subprocess, you need a way to see its internal logs.

```
def my_logger(line):
    if "[ERROR]" in line:
        logger.error(line)
    else:
        logger.debug(line)

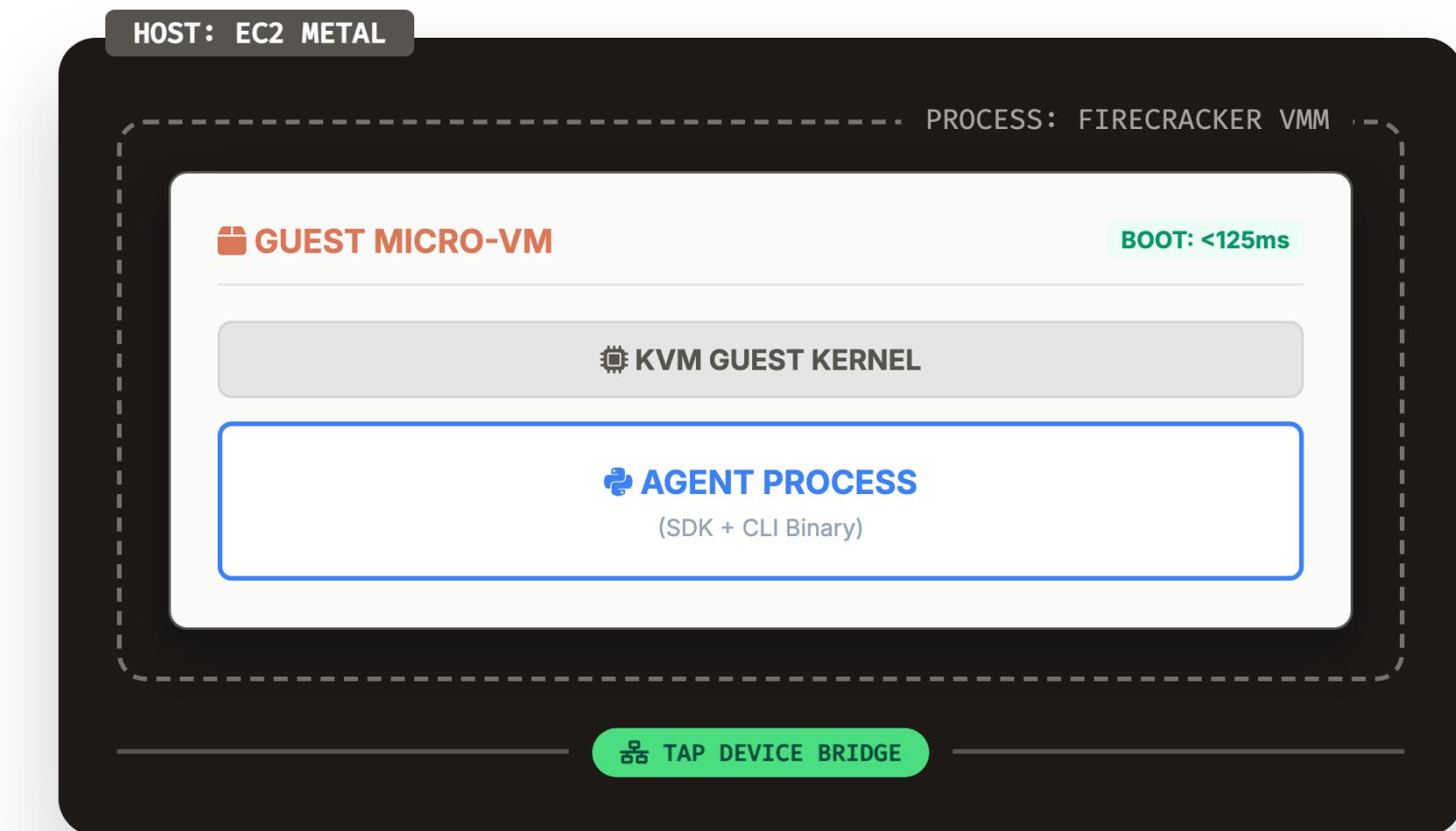
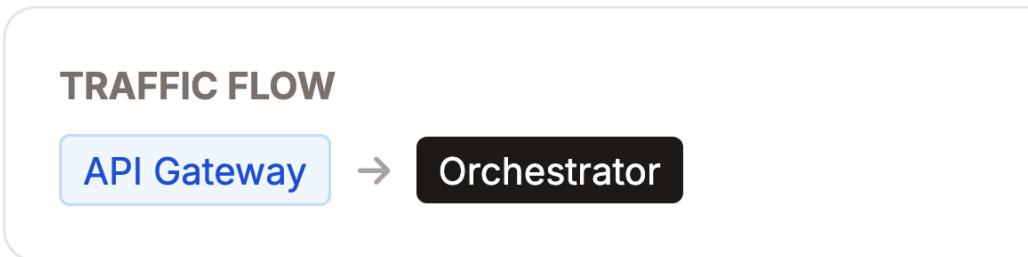
options = ClaudeAgentOptions(
    stderr=my_logger,
    extra_args={"debug-to-stderr": None} # Enable debug mode
)
```

# Enterprise Deployment

*Sandboxing & Production*

# Deep Sandboxing: Firecracker VMs

The SDK's internal sandbox protects the *filesystem*, but for running untrusted code from users, you need **Kernel Isolation**.



# Defense in Depth

---

Layering protections ensures maximum security for enterprise agents.

## Layer 1: SDK Permissions

Controls what tools (Edit/Read) can be called. Prevents accidental file overwrites.

## Layer 2: Bash Sandbox

SDK [SandboxSettings](#). Restricts network access and blocks dangerous syscalls.

## Layer 3: Firecracker/Docker

OS-level isolation. Prevents the agent from accessing host resources if Layer 1 & 2 fail.

# Sandboxing (Bash)

The SDK supports configuring the Bash sandbox for file and network isolation.

```
sandbox_config = {
    "enabled": True,
    "network": {
        "allowUnixSockets": ["/var/run/docker.sock"],
        "allowLocalBinding": True
    },
    "excludedCommands": ["git"] # Git runs outside sandbox
}

options = ClaudeAgentOptions(sandbox=sandbox_config)
```

**Note:** Filesystem restrictions (Read/Write) are handled via Permissions, while Sandboxing handles process isolation.

# Deployment Architecture



## Containerization

Wrap your Python script + SDK in a Docker container. The SDK will pull the CLI binary inside the container.



## Secret Management

Inject `ANTHROPIC_API_KEY` via environment variables. Do not hardcode.



## Headless Mode

Use `query()` for background jobs.  
Use `ClaudeSDKClient` for WebSocket/API backends.

# Build with Confidence

*The Claude Agent SDK provides the harness you need to move from Chatbots to Autonomous Agents.*

```
pip install claude-agent-sdk
```

[docs.anthropic.com/en/docs/clause-code/sdk](https://docs.anthropic.com/en/docs/clause-code/sdk)