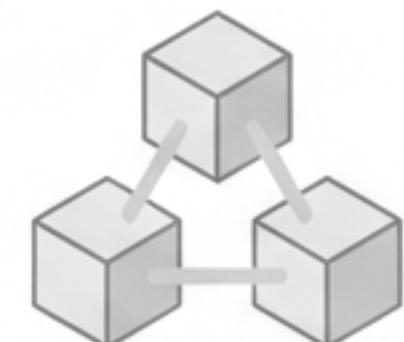


# The Anthropic Way to Build Agents

A Deep Dive into the Claude Agent SDK  
and its Core Principles



# From Simple Features to Autonomous Agents

## Single LLM Feature

e.g., "Categorize this text."

## Structured Workflow

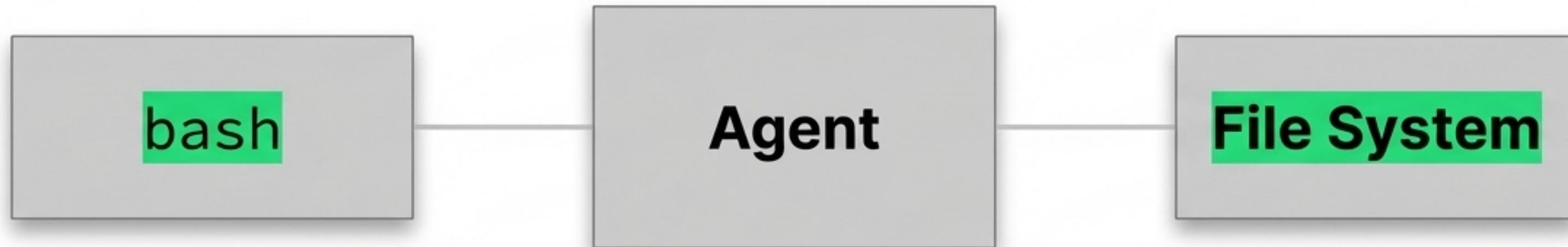
e.g., "Index this codebase via RAG."

## Autonomous Agent

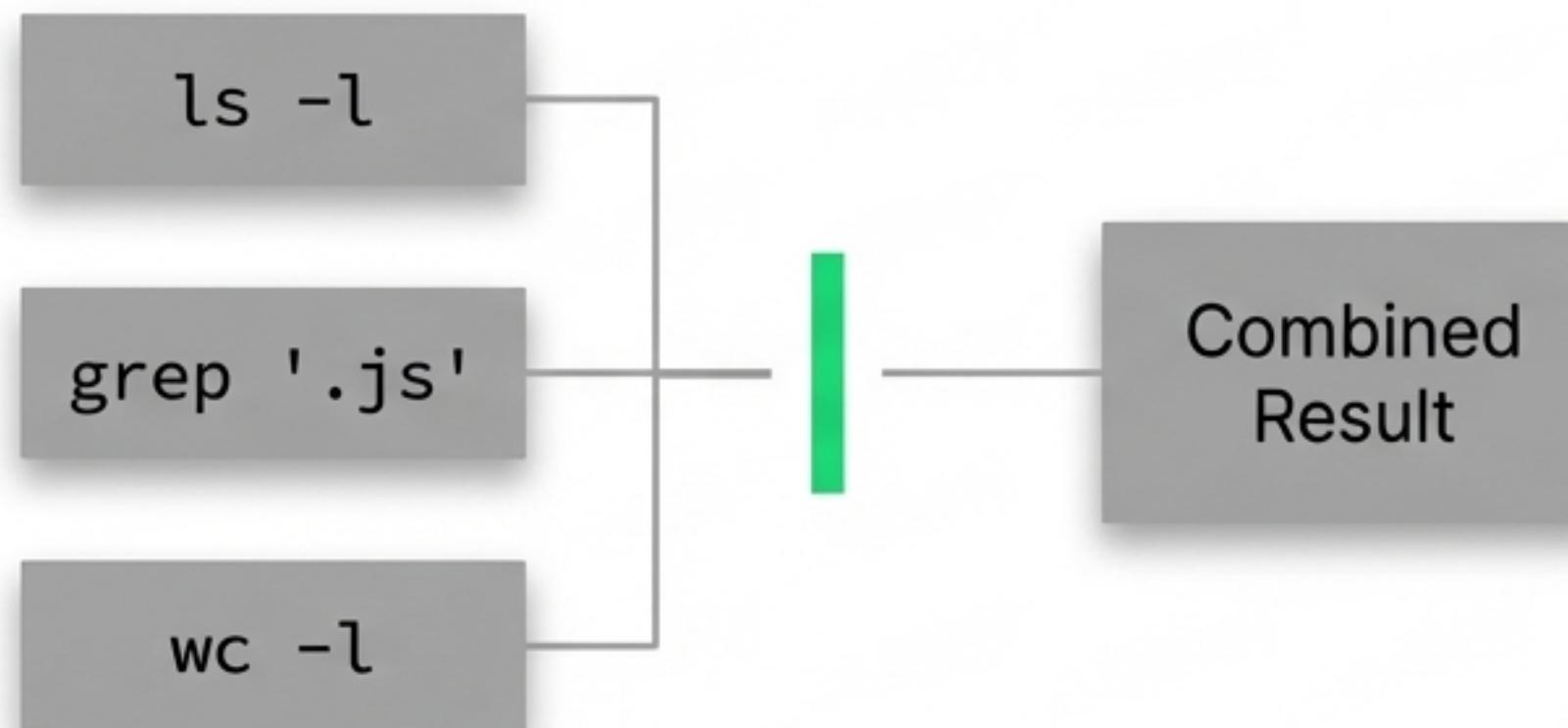
e.g., "An AI coding partner that builds its own context and decides its own trajectory."

# Our Core Thesis: The Power of Primitives

We believe the most powerful, flexible, and generalizable agents are built not on a vast library of rigid tools, but on the timeless primitives of computing: a powerful shell (`bash`) and a versatile file system.



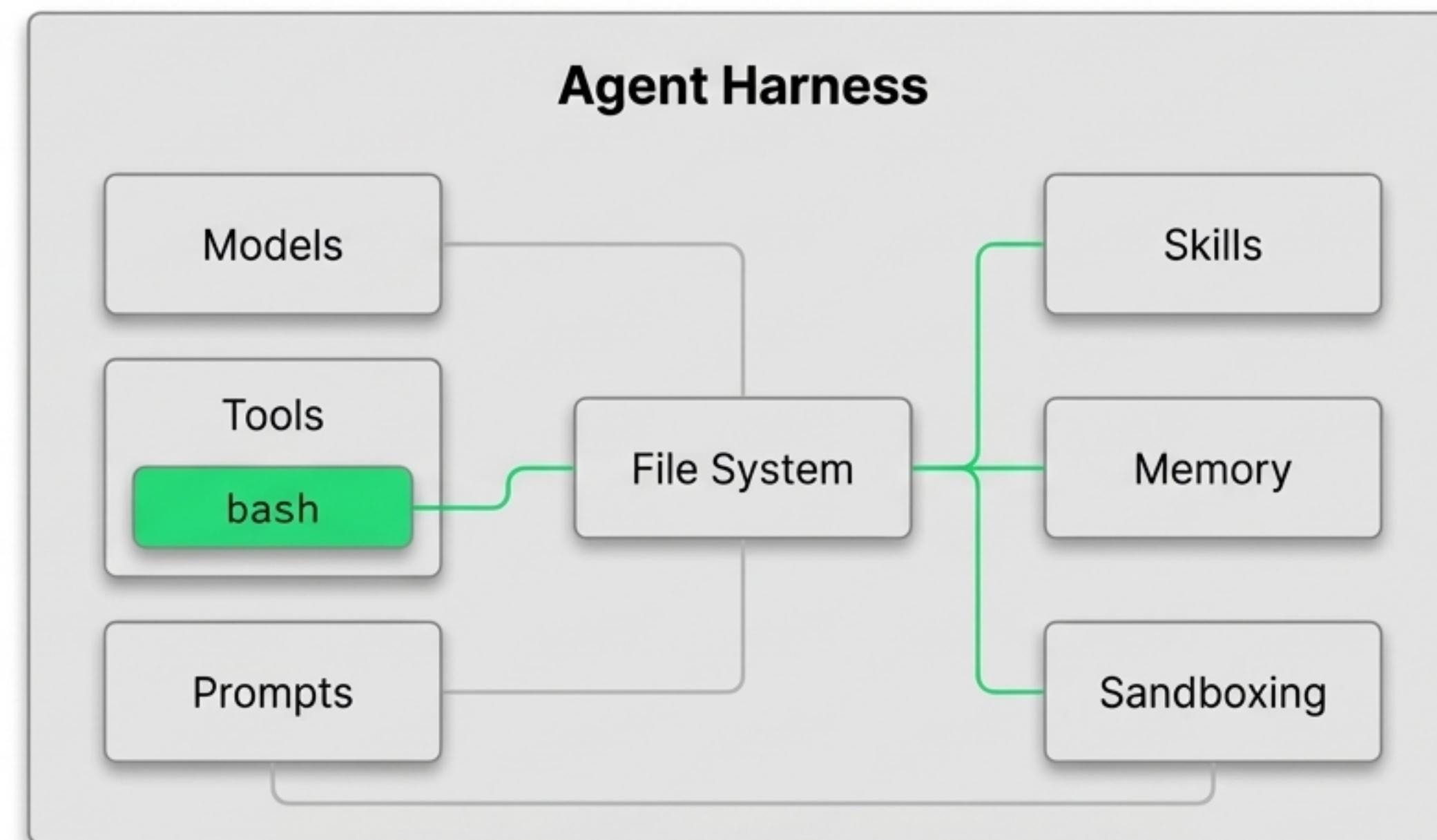
# Bash is the Original ‘Code Mode’



- Compose Functionality: Pipe outputs and chain commands (`grep`, `awk`, `jq`).
- Dynamic Scripting: Generate and execute scripts on the fly to solve novel problems.
- State Management: Store tool call results and memory in files, which can then be manipulated.
- Leverage Existing Software: Utilize powerful, pre-existing CLIs like `ffmpeg` or `npm` without custom tool definitions.

# Introducing the Claude Agent SDK: A Production-Grade Harness

The SDK packages the essential components we found ourselves rebuilding constantly while developing agents like Claude Code.



# The Agent's Spectrum of Action: Choosing the Right Primitive

## Tools

**Use For:** Atomic, reliable, high-control actions.  
Non-reversible changes.

### Example:

`send_email`, `write_file`

**Trade-offs:** High context usage, not composable.

## Bash

**Use For:** Composable actions, file system operations, discovery.

### Example:

`git diff | grep 'TODO'`

**Trade-offs:** Requires discovery (`--help`), slightly lower call rates.

## Code-Gen

**Use For:** Highly dynamic logic, composing APIs, complex data analysis.

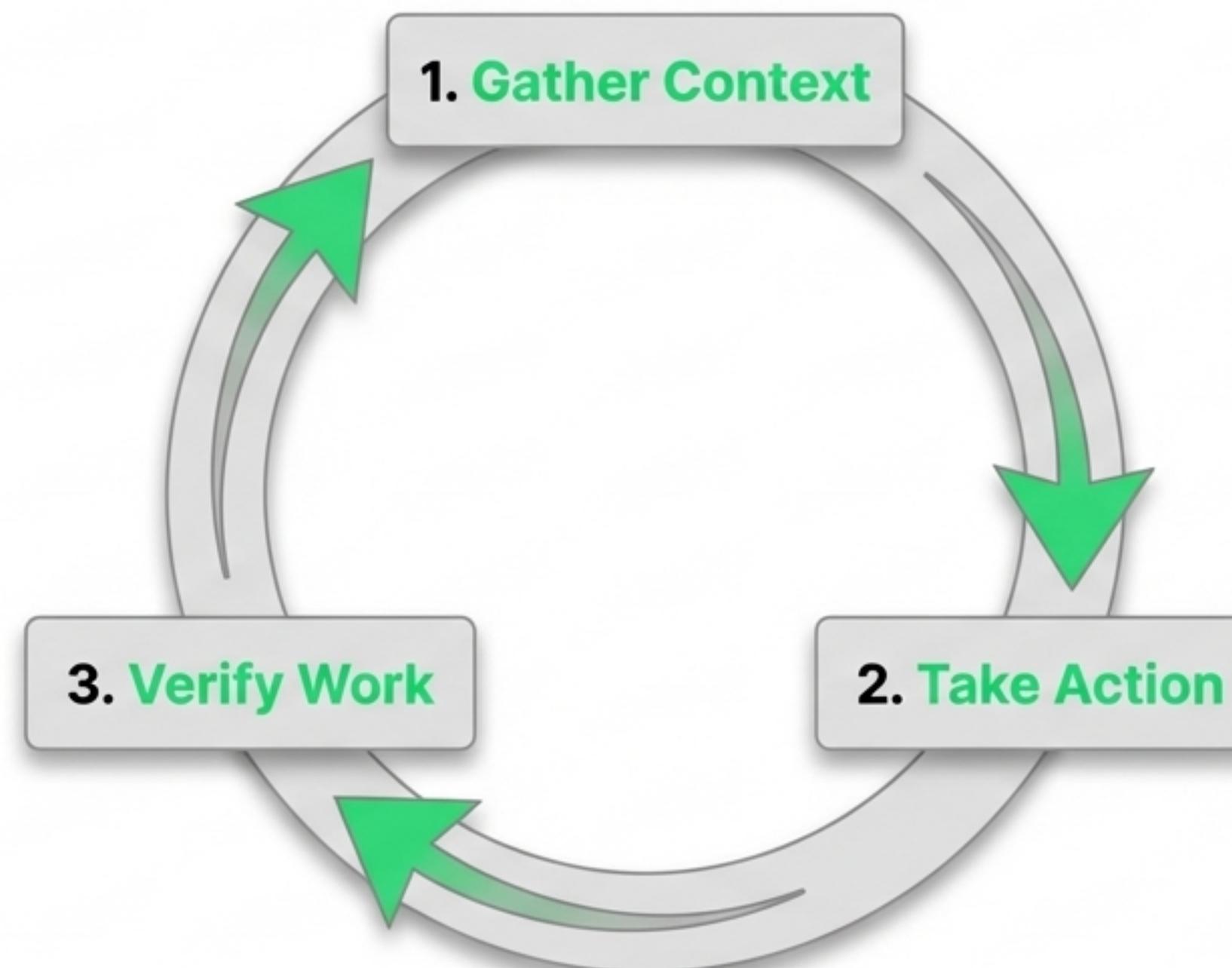
### Example:

Generating a script to query a novel API

**Trade-offs:** Longest execution time (linting, compilation).

# The Core Agent Loop: A Disciplined Methodology

Every agent task can be deconstructed into a simple, iterative loop. This provides a mental model for structuring agent behavior.

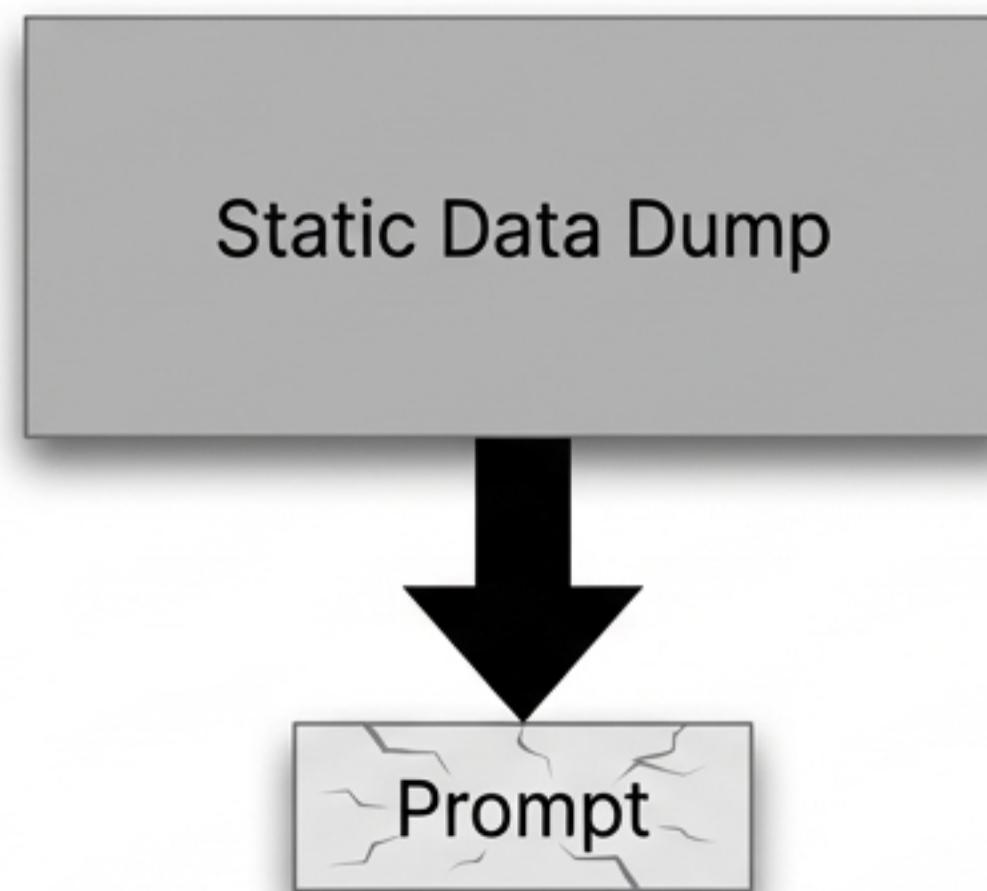


# Step 1: Let the Agent Discover Its Own Context

The most effective agents aren't just given context; they are given the tools to \*discover\* it.

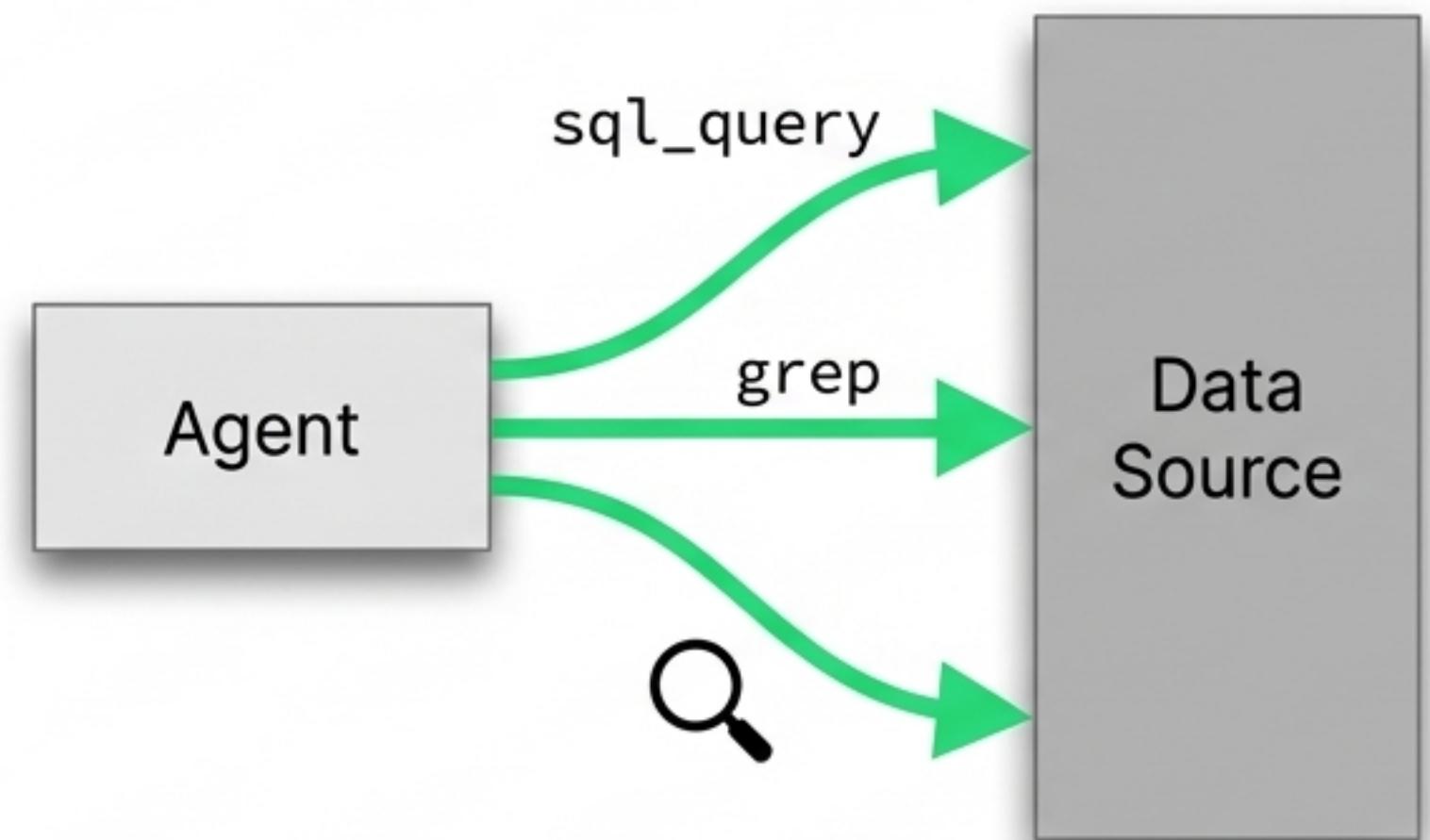
## Less Effective: Static Context

Stuffing the prompt with massive, static data dumps.



## More Effective: Dynamic Discovery

Providing an interface (e.g., a SQL query tool for a CSV, or grep for a codebase) and letting the agent explore.



# Steps 2 & 3: Taking Action and Ensuring Correctness

## Take Action

Leverage the full spectrum (Tools, Bash, Code-Gen) based on the task's requirements.

## Verify Work

This is **critical** for reliability.



**Deterministic Verification:** A powerful first line of defense. Examples: Linting, compiling, schema validation, checking if a file was read before being written to.



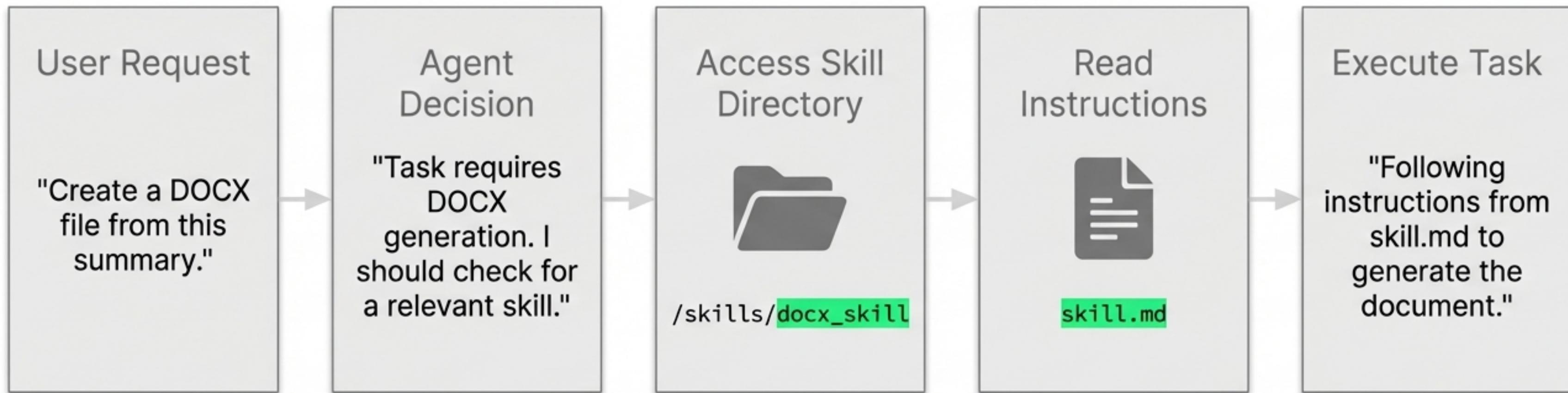
**Model-Based Verification:** Use sub-agents for adversarial checks or critiques of the primary agent's work.

## Core Insight

The more verifiable and reversible a task is, the better it is for today's agents. (e.g., code with git history vs. complex UI state).

# Advanced Technique: "Progressive Context Disclosure" with Skills

Skills are collections of prompts and files that give an agent expert, just-in-time knowledge. They are a pattern for the agent to discover how to perform complex tasks.



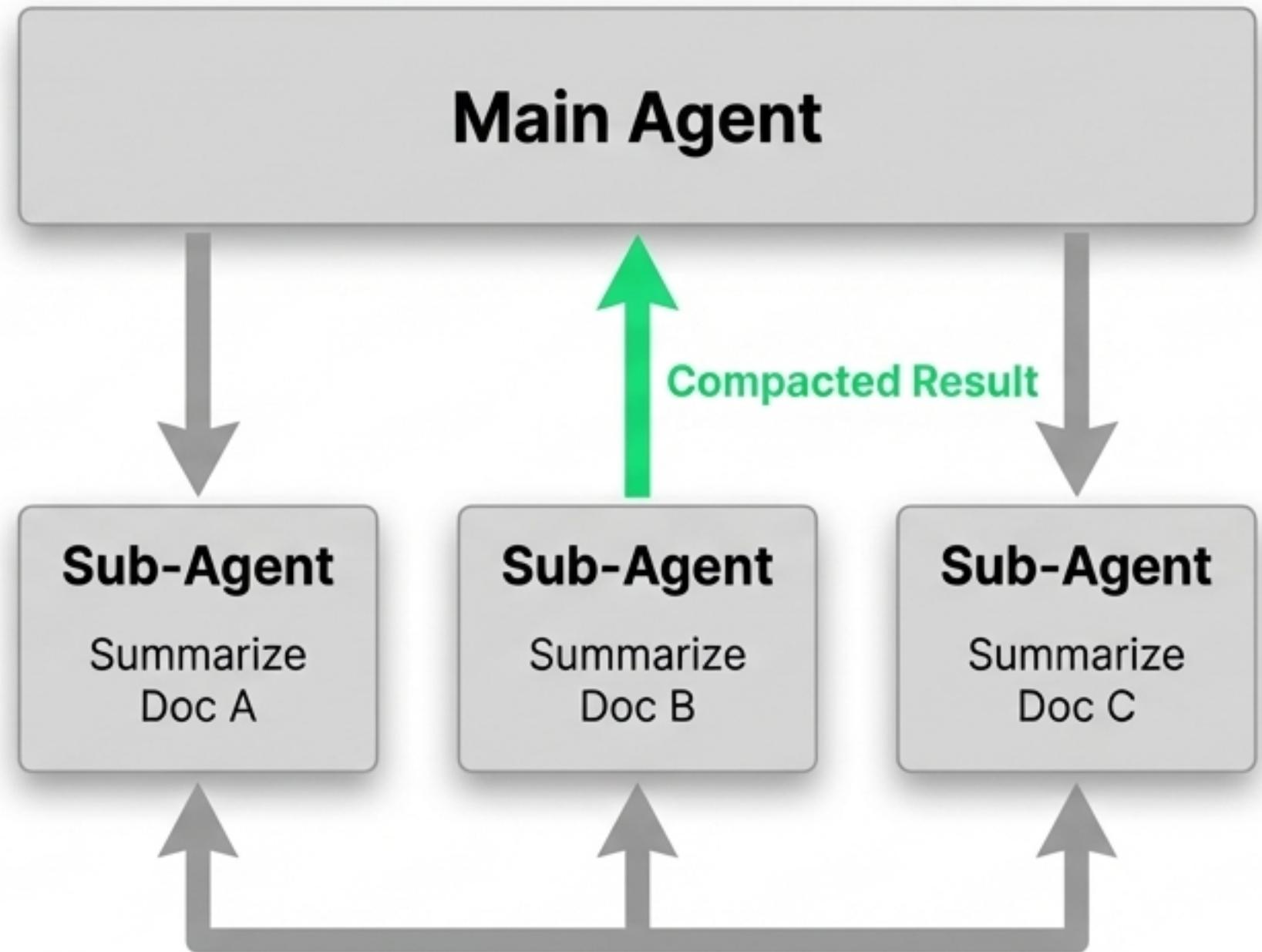
# Advanced Technique: Managing Complexity with Sub-Agents

## Benefit 1: Preserves Context

The main agent's context window isn't polluted with intermediate steps. It only receives the final, compacted result.

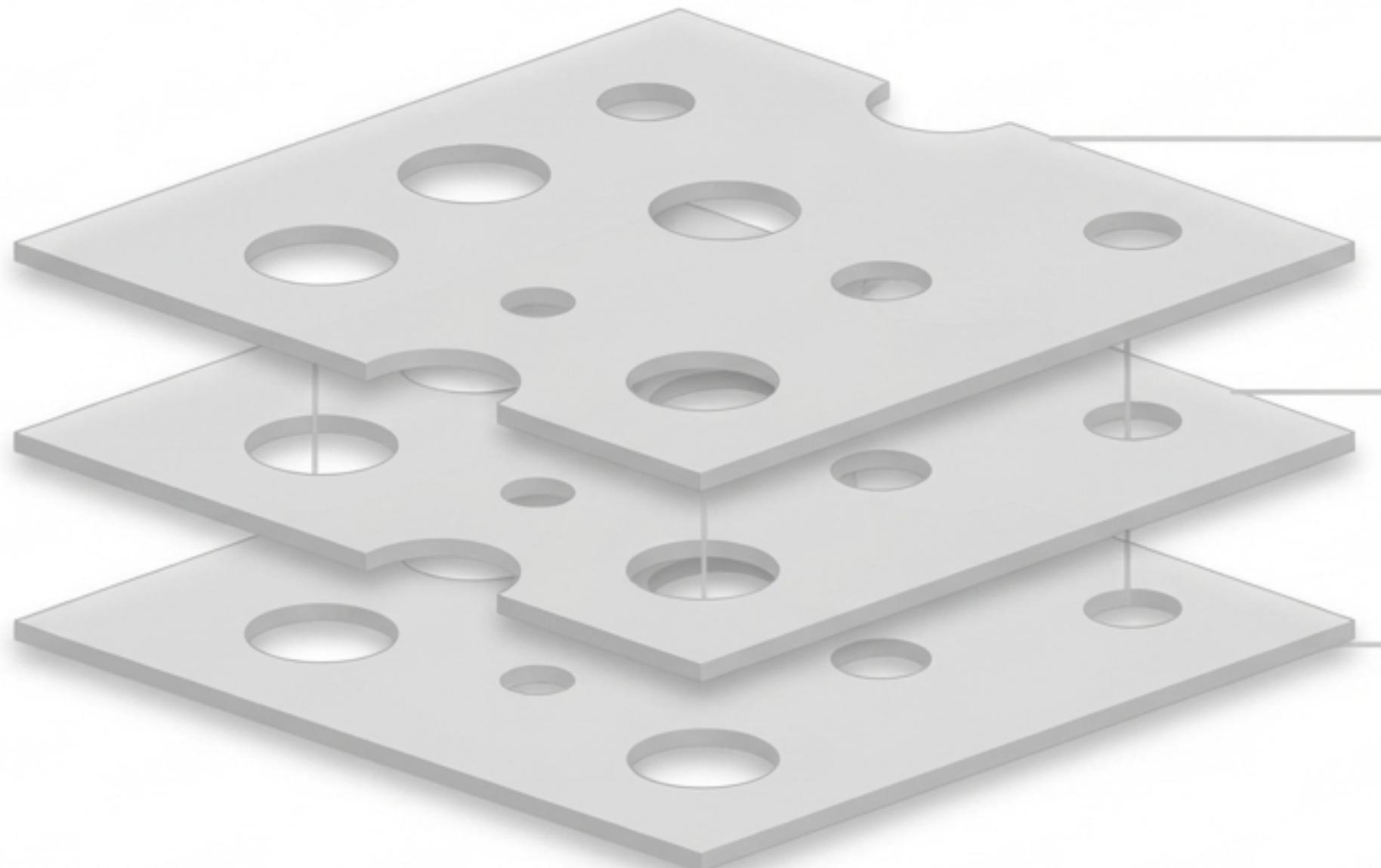
## Benefit 2: Enables Parallelism

Spin up multiple sub-agents to perform tasks concurrently (e.g., summarize different documents simultaneously).



# Security: A ‘Swiss Cheese’ Defense Model

Giving an agent access to `bash` requires a multi-layered security approach.  
No single layer is perfect, but together they provide robust protection.



- 3. Sandboxing:** Isolating the execution environment to control file system access and network requests.
- 2. Harness & Permissions:** Parsing `bash` commands, enforcing rules (e.g., read-before-write), and managing permissions within the SDK.
- 1. Model Alignment:** Training models to be helpful and harmless as a baseline.

# How to Prototype Your First Agent

The **fastest way** to prototype is to bypass building a harness initially. Start with Claude Code.

1. Open Claude Code.
2. Provide it with your libraries, scripts, and API documentation as files.
3. Define the high-level task and constraints in a `claude.md` file.
4. Instruct it to perform the task and—most importantly—**read the transcript**. Observe its behavior, identify where it struggles, and iterate on your prompts and APIs.

---

*“Simple is not the same as easy. Your final agent code should be elegant and simple, but getting there requires thoughtful iteration.”*

# The Anthropic Way: Four Core Principles

## Embrace Primitives

Start with bash and the file system for maximum flexibility and power.

## Think in Loops

Structure agent logic around Gathering Context, Taking Action, and Verifying Work.

## Let Agents Discover

Design interfaces that allow agents to find their own context, rather than being force-fed information.

## Prototype Rapidly

Use Claude Code as your initial development environment to focus on the core agent logic first.

# Resources & Further Reading

- Claude Agent SDK Documentation
- Official GitHub Repository
- Anthropic Cookbook Examples
- Follow on X (Twitter): TRQ212