

EXPERT INSIGHT

<packt>

The Claude Code Operating Model

Build scalable AI coding systems with Skills,
MCP, Hooks, agent orchestration,
and SDK patterns



Jia Huang

The Claude Code Operating Model

Build scalable AI coding systems with Skills, MCP, Hooks, agent orchestration, and SDK patterns

Jia Huang

<packt>

The Claude Code Operating Model

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Ashwin Nair

Portfolio Manager: Nitin Nainani

Content Engineer: Kinnari Chohan

Project Manager: Vishnu Priya R

Technical Editor: Rohit Singh

Copy Editor: Kinnari Chohan

Indexer: Manju Arasan

Proofreader: Kinnari Chohan

Production Designer: Ajay Patule

Growth Lead: Sayantani Saha

First published: August 2026

Production reference: 1060826

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK.

ISBN 978-1-80808-271-9

www.packtpub.com

To my wife and our children, who remind me that the best systems are the ones we build together.

— Jia Huang

Contributors

About the author

Jia Huang is an AI researcher at A*STAR, Singapore, and a technical author who writes about how AI agents are engineered. He is the author of *Designing AI Agents* and *RAG from First Principles*.

He proposed the dual-axis framework for agent design patterns, which classifies patterns by cognitive function and execution topology, and the Pattern Selection Card, a practical method for choosing patterns under real cost and latency constraints. His work focuses on the engineering that surrounds the model rather than the model itself: context, tools, execution environments, and governance. He writes in both Chinese and English.

My thanks to Lucy Wan, who bridged my Chinese and English markets and brought me to the Packt team, a connection that opened the door to a remarkable period of collaboration. I am also grateful to Kinnari Chohan, whose professional editing made this English edition clearer and stronger than I could have made it alone.

My Chinese publishing team at Posts & Telecom Press has supported this work throughout. Jikang Chen and Sha Li backed me at every turn and helped shape the way I write; Jian Qin edited the Chinese edition; and Chenyu Yuan created the illustrations.

My thanks also go to the GeekTime team, especially editors Yuxin Zhao and Dongqing Wang, and to the many learners in my Claude Code course. Their thoughtful questions, practical feedback, and valuable suggestions helped me refine the ideas in this book and make them more useful to practitioners.

I am grateful to my colleagues in the ADPS community—the Agent Design Patterns Society—Bingsheng Ru, Ning Jiang, Bo Liang, Professor Haofen Wang, Huisheng Yin, Haili Zhang, and Pingping Bai, for building this community with me.

And finally, my deepest thanks to my parents for their unconditional love and support.

About the reviewers

Wei Wei is an AI engineering practitioner and technical author with an MSc in Artificial Intelligence from the University of Aberdeen (UK). He specializes in LLM systems, agentic AI architecture, and Harness Engineering. The author of five-plus technical books spanning Python systems, web architecture, and applied machine learning, he has led big data platform development at Tongcheng Digital Technology and built multimodal intelligent systems as a Senior ML Scientist. As an independent consultant, he has delivered enterprise AI training and advisory to institutions including Citibank, Bank of China, and Ping An Life Insurance, and lectures on LLM engineering at over ten universities.

Yury Vashchylau makes software and automation for a living, and treats the two as the same thing. Long before agentic AI became a category, he ran his engineering work on top of a self-hosted stack of services. He believes that the principles that shape good code also shape the AI harnesses built around it. His work spans banking infrastructure, contributing to open-source, consumer apps at DIGITALBY, and developer tooling. He writes at yuryv.info, and is interested in animation, filmmaking, business processes, and how automated systems evolve over time.

Table of Contents

Preface	xvii
----------------	-------------

Free benefits with your book.....	xxii
-----------------------------------	------

Chapter 1: Claude Code as an Agent Framework: a Technical Architecture Overview	1
--	----------

From Command-Line assistant to agent framework	2
--	---

Four-Layer architecture model.....	4
------------------------------------	---

Memory layer: the CLAUDE.md system • 6

Extension layer: the four major extension components • 8

Integration layer: connecting to the outside world • 12

Programming layer: agent SDK • 14

Underlying perspective: harness and the agentic loop • 16

Component relationships and collaboration.....	18
---	-----------

Comparison of trigger mechanisms • 18

Data flow: the journey of a request • 20

Plugins: packaging and distribution of combinations • 21

Technical selection guide	22
--	-----------

Summary	25
----------------------	-----------

Reflection questions.....	26
----------------------------------	-----------

Chapter 2: Learning From the Past: Engineering Practice of the Memory System	27
---	-----------

Why AI needs "Memory"	28
-----------------------------	----

The Five-Layer memory architecture	30
--	----

CLAUDE.md writing paradigms.....	35
----------------------------------	----

First question: WHY (why do this?) • 36

Second question: WHAT (what to do, what not to do) • 36

Third question: HOW (how to do it step by step) • 36

Conditional Rule System	40
--------------------------------------	-----------

In practice: three typical project configurations.....	43
---	-----------

React frontend project configuration • 43	
Node.js backend project configuration • 45	
Python data project configuration • 46	
Summary	48
Reflection questions	49
 Chapter 3: Teaching a Man to Fish: Engineering Reusable Skills	 51
 From CLAUDE.md to Skills: Two Dimensions of Knowledge.....	 52
Dissecting a Skill: Skeleton and Texture	55
Directory Structure • 55	
YAML Frontmatter: The "ID Card" of a Skill • 57	
Progressive Disclosure: The Return on Investment of Knowledge	59
The Library Model • 60	
The Budget Mechanism of description • 62	
Trigger Mechanism: How Claude Code Decides Which Skill to Invoke	63
Dual-Channel Activation Mechanism • 63	
Description: The Soul of Skills • 64	
Preventing Over-Triggering and Under-Triggering • 66	
Reference Skills vs. Task Skills: Two Skill Philosophies • 67	
SKILL.md Main Content: Router, Not Repository	69
Router Mindset • 69	
Contractual Reference • 71	
The 500-Line Rule • 71	
allowed-tools: Knowledge-Constrained Behavior.....	72
Permission Design Templates • 72	
Fine-Grained Bash Control Syntax • 73	
Parameter Passing and Dynamic Injection	74
\$ARGUMENTS and Positional Parameters • 74	
Dynamic Context Injection • 75	
Scope and Priority • 77	
In Practice: Building 3 Skills from Scratch • 78	
Task-oriented Skill: Intelligent Commit • 81	
Composite Skill: Financial Analysis (Progressive Disclosure Full Case) • 83	
Four Skill Design Patterns • 85	
Skills from a Software Engineering Perspective • 87	

<i>Separation of Concerns</i> • 88	
<i>Dependency Inversion</i> • 88	
<i>Cache Optimization and Lazy Loading</i> • 88	
<i>Principle of Least Privilege</i> • 88	
<i>Open standards</i> • 89	

Summary	89
Reflection Questions	90

Chapter 4: Divide and Conquer: The Art of Sub-Agents and Task Delegation 93

The Context Window Dilemma	94
The Essence of Subagents.....	95
Defining and Configuring Subagents	97
Sub-agent Modes	103
Read-Only: The Safe Observer • 103	
Execution: High-Noise Task Processor • 104	
Parallel: Multi-Expert Workflow • 105	
Pipeline Mode: Serial Processing Chain • 107	
Collaboration between Sub-agents and Skills	113
Token Economics	120
Subagents from a Software Engineering Perspective.....	123
Single Responsibility Principle • 124	
Bulkhead Pattern • 124	
MapReduce Pattern • 124	
Chain of Responsibility Pattern • 125	
Practical Considerations	125
Summary	126
Discussion Questions	127

Chapter 5: From Guidelines to Guardrails: Automating Control with Hooks 129

The Role of Hooks in the Claude Extension System	130
Event Lifecycle: More Than Two Dozen Events	131
Session-Level Events • 131	
<i>SessionStart</i> • 131	

<i>SessionEnd</i> • 132	
<i>PreCompact</i> • 132	
Tool Invocation Events • 132	
<i>PreToolUse</i> (the most powerful event in the entire Hooks system) • 132	
<i>PostToolUse</i> • 133	
<i>PostToolUseFailure</i> • 133	
<i>PermissionRequest</i> • 133	
<i>UserPromptSubmit</i> • 134	
Subagent Events • 134	
<i>SubagentStart</i> • 134	
<i>SubagentStop</i> • 134	
Completion Events • 135	
<i>Stop</i> • 135	
<i>Notification</i> • 135	
Newer Event Types • 135	
<i>TeammateIdle</i> and <i>TaskCompleted</i> • 135	
<i>ConfigChange</i> • 135	
<i>WorktreeCreate</i> and <i>WorktreeRemove</i> • 135	
"Can It Be Blocked?": The Most Critical Dimension • 136	
Configuration System: 6 Locations, 6 Purposes.....	136
Three Types of Processors: A Deterministic Ladder	139
Command Type: Deterministic Rules • 139	
Prompt Type: Single Large Model Evaluation • 140	
Agent Type: Multi-turn Intelligent Verification • 140	
hookSpecificOutput: Protocol for Communicating with Claude	141
Engineering Practice 1: Security Protection System	142
<i>PreToolUse</i> : Dangerous Command Interception • 143	
<i>PreToolUse</i> : Sensitive File Protection • 144	
<i>PostToolUse</i> : Comprehensive Operation Auditing • 146	
Complete Configuration • 147	
Engineering Practice II: Automated Code Quality	148
<i>PostToolUse</i> : Automatic Formatting • 148	
<i>PostToolUse</i> : Lint Feedback Loop • 149	
Stop Hook: Test Quality Gate • 150	
Subagent Hooks: Precise Context Management	152

Global vs. Frontmatter: The Precision Issue • 152	
SubagentStart: Automatic Context Injection • 153	
SubagentStop: Verifying Output Quality • 153	
Asynchronous Hooks: Background execution without blocking	155
Environment Variables and Debugging.....	156
Environment Variables Available to Hooks • 156	
The Three Debugging Techniques • 156	
Common Pitfalls • 157	
Engineering Design Methodology	157
Summary	158
Discussion questions.....	159
 Chapter 6: Connecting Everything: Integrating External Tools with MCP	 161

From N×M to N+M: The Power of Standardization.....	162
Architecture: Client-Server and JSON-RPC.....	163
Transmission Methods: Three Forms of Connection	164
Configuration Details: From CLI to Configuration File.....	165
Quick CLI Configuration • 165	
.mcp.json Configuration File • 166	
Configuration File Location and Scope • 167	
Practical Example 1: Connecting to a Database	167
Practical Example 2: Building a Custom MCP Server	168
Common MCP Server Ecosystem.....	172
Official Servers • 172	
Popular Third-Party Servers • 173	
Practical Configuration Combinations • 174	
Security Mechanisms: The Boundaries of Trust	175
Three-Layered Defense Mechanism • 175	
Security Risks and Countermeasures • 176	
MCP+Skills: Kitchen and Recipe.....	177
Enterprise Deployment	179
Debugging and Troubleshooting.....	179
Common Debugging Methods • 179	
Token Management for MCP Tool Output • 180	

Common Issues • 180	
Summary	181
Discussion Questions	182
Chapter 7: Headless Mode and CI/CD Integration	183
From Interactive to Unattended: A Key Architectural Evolution.....	184
Core Parameter System: Control Across Four Dimensions.....	185
Output Format Control • 186	
text format • 186	
json format • 187	
stream-json format • 188	
Cost Guardrails • 189	
Security Boundaries: Tool Permissions • 190	
Execution Control: Model, Prompt, Structured Output • 191	
<i>Model Selection • 191</i>	
<i>System Prompt Customization • 191</i>	
<i>Structured Output • 192</i>	
Unix Pipelines: Integrating Claude into Command Line Workflows	193
Practical Example 1: Automated Code Review with GitHub Actions	194
Using the Official Action • 194	
Direct CLI Invocation • 196	
Practical Example 2: Multi-Stage CI Pipeline	199
Streaming Output: Real-Time Monitoring of Long-Running Tasks.....	202
Session Management: Maintaining Context Across Steps	203
CI Environment Configuration: Production-Grade Checklist.....	205
Required Environment Variables • 205	
Performance Tuning Parameters • 206	
Complete Configuration Template for GitHub Actions • 206	
MCP Server Configuration in CI • 207	
Cross-Platform CI/CD Integration.....	207
GitLab CI/CD Configuration • 207	
Jenkins Pipeline Integration Configuration • 208	
Local Automation Scripts • 209	
Security Best Practices: From RBAC to Container Isolation	210
Principle of Least Privilege • 211	

Secrets Management • 212	
Container Isolation • 212	
Cost Protection • 213	
Audit Logs • 213	
From CLI to Agent SDK: Programming Interfaces for Headless Mode.....	214
Progressive Implementation Strategy	216
Summary	217
Discussion Questions	217
 Chapter 8: Building with Claude as the Engine: Agent SDK	 219
Intelligent Agent Development Kit	
 Positioning of the Agent SDK: From Tool to Component.....	 220
Core API: The query Function.....	221
Python Version • 221	
TypeScript Version • 222	
Message Types: Interpreting Claude's Output Stream	223
system/init—Session Initialization • 223	
assistant—Claude's Response • 223	
user—Tool Execution Results • 224	
result—Task Completion • 224	
ClaudeAgentOptions: Fine-Grained Behavior Control	226
Detailed Explanation of Permission Modes • 228	
Pattern Matching for Tool Permissions • 229	
Session Management: Context Continuity Across Calls.....	229
Session Continuity Based on session_id • 229	
Conversation Forking • 230	
Custom Tools: Extending the Capability Boundaries of Claude.....	231
Defining Tools with the @tool Decorator • 231	
Register and Enable Custom Tools in Agent Configuration • 233	
Using Pydantic Models for Parameter Validation • 233	
Hooks in the Agent SDK: Programmatic Interception.....	235
PreToolUse: Pre-execution Interception • 235	
PostToolUse: Execute Post-Processing • 237	
Relationship Between Agent SDK Hooks and Shell Hooks • 238	
Four Layers of Security Defense.....	239

Structured Output: Enforcing JSON Schema	242
In Practice: Building a Code Analysis Web Service.....	243
Agent SDK and Headless CLI: How to Choose.....	247
Summary	248
Questions for Reflection.....	249

Chapter 9: From Personal Craft to Shared Asset: The Plugin Ecosystem **251**

The Role of Plugins: Encapsulation and Distribution of Capabilities	252
Physical Structure of a Plugin.....	253
plugin.json: The Plugin's Identity Card • 255	
Specific Format of Components • 256	
<i>Command Files</i> • 256	
<i>Sub-agent Files</i> • 257	
<i>Hooks Configuration File</i> • 257	
<i>MCP Configuration File</i> • 258	
Installation and Lifecycle Management	258
Installation Sources • 258	
Routine Management • 259	
Local Development and Testing • 259	
Storage Location • 259	
Namespaces: Multiple Plugins Coexisting.....	260
In Practice: Building a Team Capability Pack	260
Complete Directory Structure • 261	
plugin.json • 261	
Security Scanning Sub-Agent • 262	
Hooks: Security Check Script • 263	
Release Process • 263	
Private Marketplace and Enterprise Management	264
Building a Private Marketplace • 264	
Using a Private Marketplace • 265	
Organization-Level Plugin Management • 265	
Plugin Design Principles	266
Single Responsibility • 266	
Incremental Iteration • 266	

Least Privilege • 266	
Documentation Is Essential • 267	
When to Package Capabilities as a Plugin	267
LSP Support and Future Evolution	268
Summary	269
Discussion Questions	270
 Chapter 10: From Individual to Team: Engineering Practices for Claude Code	 271
Cost: Make Tokens Work for You, Not Burn Your Money	272
Understanding the Pricing Structure • 272	
Model Selection Strategy • 273	
Context Management Is the Core of Cost Control • 274	
Prompt Caching • 275	
Cost Monitoring • 276	
Debugging: Opening the Black Box	277
--debug Mode: An "X-ray" of the Execution Process • 277	
stream-json: Real-Time Insight into the "Thought Stream" • 278	
PostToolUse Hook: Building Persistent Audit Logs • 278	
Session Archiving and Breakpoint Debugging • 279	
Diagnosing Abnormal Claude Behavior • 280	
Security Guidelines: Least Privilege Is the Bottom Line	281
Permission Modes: The First Line of Defense • 281	
Tool Permissions: Fine-Grained Control • 282	
.claudeignore and File Access Control • 283	
Hook Security Scanning • 284	
Trust Assessment of MCP Servers • 284	
API Key Lifecycle Management • 285	
Large Codebases: Navigating Millions of Lines of Code Within a Limited Context Window..	286
Hierarchical CLAUDE.md • 286	
Guide Searches, Not Exhaustive Reading • 288	
Sub-Agents: Building Modular Context Boundaries • 289	
Context Compression and Fine-Grained Management • 290	
Prompt Engineering: Writing Effective CLAUDE.md and Prompts	290
Principles for Writing CLAUDE.md • 290	

Techniques for Precise Expression • 291	
<i>Quantify everything</i> • 291	
<i>Example-driven</i> • 292	
<i>Define boundaries clearly</i> • 292	
Stepwise Confirmation • 292	
Team Adoption: The Leap from Individual to Organization	293
Shared Configuration: The Cornerstone of Team Unity • 293	
Governance Standards for CLAUDE.md • 294	
Organization-Level Policy Management • 295	
Progressive Capability Building Roadmap • 295	
Claude Code Onboarding Training Program • 296	
Integration of the SDD Ecosystem and Workflow	298
Four-Layer SDD Ecosystem Architecture • 298	
Recommended Workflow Combination Solutions • 299	
Global View: The Orchestration of Nine Core Mechanisms	300
Summary	302
Reflection Questions	303
 Afterword: The Way of the Blade	 305
Other Books You May Enjoy	309
Index	315

Preface

For two years, the industry asked one question above all others: which model is the strongest? It was the right question for its time. But somewhere in 2025, the ground shifted. The frontier models became so capable and so close to one another that raw model quality stopped being what separated a working system from a frustrating one. What separated them was everything around the model: the context you fed it, the tools you gave it, the environment you ran it in, and the guardrails you placed on it. That surrounding structure has a name in this book: the Harness. The central argument of these pages is simple to state and demanding to practice, an agent is a Model plus a Harness, and once models are commoditized, the Harness is where the engineering lives.

Claude Code is the clearest embodiment of that idea available today. On the surface, it is a command-line coding assistant. Underneath, it is a complete, inspectable Harness: `CLAUDE.md` injects context, Skills and sub-agents extend capability, Hooks and permissions enforce control, and Headless mode, the Agent SDK, and Plugins carry all of it into real execution environments. This book takes Claude Code apart mechanism by mechanism, not to admire it, but to teach the underlying discipline, so that what you learn transfers to any agent you build, in any framework, long after the specific tool has changed.

This is a hands-on engineering book, not a survey. Every mechanism is introduced through a concrete problem, worked in real code, and pushed until it either holds up or reveals its limits. The chapters build on one another: the early chapters establish the four layers of a Harness, context, tools, execution, and governance, and the later chapters compose them into headless pipelines, custom SDK agents, shareable plugins, and, finally, the team practices that let a whole organization work this way without chaos.

A note on the cast. Throughout the book, you will meet three recurring characters in short dialogues. Xiaobing is an engineer applying Claude Code to real projects; Xiaoxue brings a data-analysis perspective and asks the questions a practitioner actually runs into; and Kage is the senior technical mentor who helps them reason through architecture decisions. They are not decoration, each conversation stages a design tension you will face yourself, and the chapter that follows resolves it in code.

The tools in this space move quickly, and some specifics in these pages will age. The mechanisms will not. Injection is still injection, context windows still fill, and a well-drawn boundary between what an agent may perceive and what it may do is still the difference between a system you can trust and one you cannot. My aim is that when you finish, you will

not merely know how to operate Claude Code, you will know how to think about the Harness beneath any agent, and how to build one that earns its trust.

Who this book is for

This book is for software engineers, technical architects, and engineering leads who have used an AI coding assistant and now want to build with one seriously, reliably, at scale, and as a team. If you have run Claude Code or a similar tool and felt both its power and its rough edges, and you want to understand the machinery well enough to shape it rather than fight it, this book is written for you. A working familiarity with the command line, Git, and at least one programming language (the examples lean on Python, JavaScript/TypeScript, and shell) is assumed. You do not need prior experience with agent frameworks or LLM internals; the book builds those ideas from first principles as it goes.

What this book covers

Chapter 1, Claude Code as an Agent Framework: a Technical Architecture Overview, establishes the book's central lens, an agent as Model plus Harness, and maps the four layers (context, tools, execution, and governance) that the rest of the book fills in.

Chapter 2, Learning from the Past: Engineering Practice of the Memory System, examines how Claude Code manages context through the CLAUDE.md hierarchy and its memory scopes, and how to keep that context lean, layered, and loading in the right order.

Chapter 3, Teaching a Man to Fish: Engineering Reusable Skills, introduces Skills, declarative capability packages defined in SKILL.md, and shows how to design, scope, and trigger them so Claude reaches for the right capability at the right moment.

Chapter 4, Divide and Conquer: The Art of Sub-Agents and Task Delegation, covers sub-agents and context isolation: how to decompose work, delegate it to focused agents, and aggregate the results deterministically.

Chapter 5, From Guidelines to Guardrails: Automating Control with Hooks, turns soft guidelines into enforced guardrails using the event-driven Hooks mechanism, including how a hook blocks a dangerous action before it runs.

Chapter 6, Connecting Everything: Integrating External Tools with MCP, opens Claude Code to the outside world through the Model Context Protocol, connecting external tools, data sources, and credentials safely.

Chapter 7, Headless Mode and CI/CD Integration, takes Claude Code off the interactive terminal and into automation: headless invocation, budget and turn limits, and full CI/CD pipelines on GitHub Actions, GitLab, and Jenkins.

Chapter 8, Building with Claude as the Engine: Agent SDK Intelligent Agent Development Kit, moves from driving Claude by parameters to orchestrating it in code, custom tools with the `@tool` decorator, structured output, permission layers, and a complete SDK-based service.

Chapter 9, From Personal Craft to Shared Asset: The Plugin Ecosystem, packages Skills, sub-agents, Hooks, and MCP into distributable Plugins, and shows how private marketplaces turn one engineer's setup into a team standard.

Chapter 10, From Individual to Team: Engineering Practices for Claude Code, addresses the governance layer at organizational scale, cost, debugging, security, scale, instructions, and collaboration, so a whole team can work this way without chaos.

The Afterword, The Way of the Blade, steps back to the book's guiding metaphor and the Harness thesis, and points toward the wider methodology and design-pattern work this book connects to.

To get the most out of this book

To follow the examples, you will need a working Claude Code installation and access to Claude (an Anthropic API key or an equivalent Claude subscription). A recent Node.js runtime is required, since Claude Code runs on Node; Python 3.10+ and Git are used throughout the examples. The book assumes comfort with the command line and a code editor. Everything else, Skills, sub-agents, Hooks, MCP servers, the Agent SDK, is introduced and set up as it is needed, so you can start from a plain installation and build up.

Keep the official Claude Code documentation open alongside the book. The mechanisms are stable, but exact flags, model names, and pricing evolve; when a specific detail has shifted since publication, the documentation is the source of truth.

Download the example code files

This book includes a complete downloadable code bundle containing all the example projects and files used throughout the chapters. We recommend downloading the bundle so you can follow along smoothly and experiment with the examples.

Use the bundle as a practical starting point. Modify it, extend it, and apply what you learn by creating your own variations as you progress through the chapters.

Get the code bundle

If you bought the book directly from Packt:

1. Go to **packtpub.com**
2. Click your **profile picture** and select **Your Orders**
3. Find this book and click **Download Code**

If you bought this book from Amazon or any other channel partner:

1. Go to packtpub.com/unlock or scan the following QR code:



2. Search for this book
3. Sign up or log in to your free Packt account
4. Upload your proof of purchase and download the code bundle locally

Usage note: You're free to use and modify this code for personal learning and non-commercial projects.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, command names, file and directory names, and environment variables. For example: "The `CLAUDE.md` file is loaded automatically at the start of every session."

A block of code is set as follows:

```
{
  "hooks": {
    "PreToolUse": [
      { "matcher": "Bash", "hooks": [ { "type": "command",
        "command": "./check-bash.sh" } ] }
    ]
  }
}
```

Any command-line input or output is written as follows:

```
claude -p "Review the code changes in this MR" --output-format json --max-turns 10
```

Bold: Indicates a new term, an important word, or words that you see on the screen. Terminal labels, slash commands, and key combinations appear in the text like this. For example: "First, go to your GitHub repository, then click **Settings** | **Secrets** | **Actions**."

Note

Warnings or important notes appear like this.

Tip

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at customercare@packt.com and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packt.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packt.com/>.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



Complete Code Bundle

Download the full source code for this book's examples and projects.



DRM-Free PDF Version

Download DRM-free PDF and ePub copies of this book.



7-Day Packt Library Access

Get 7-day unlimited access to 8,000+ books and videos. No credit card required.

Available for first-time Packt+ trial users only.



Next-Gen Reader Access

Read this book on Packt Reader with progress sync, dark mode and note-taking.

How to unlock

Scan the QR code (or go to packtpub.com/unlock). Search for this book by name, confirm the edition, and then follow the steps on the page.



UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Share your thoughts

Once you've read *The Claude Code Operating Model*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



<https://packt.link/r/1808082710>

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Claude Code as an Agent Framework: a Technical Architecture Overview

"We shape our tools, and thereafter our tools shape us."

— Marshall McLuhan

Recently, Xiaobing led an AI Agent project and chose Claude Code as the core development tool. At first, its performance was astonishing: as long as the requirements were clearly stated, code could be generated rapidly, with efficiency far surpassing manual coding. However, after two weeks, a series of issues erupted all at once: every time a new session started, Claude seemed unable to remember the project's tech stack (such as FastAPI + SQLAlchemy), forcing developers to repeatedly restate background information; the generated code lacked stylistic consistency, with the Black formatting standard being applied inconsistently, resulting in three **pull requests (PRs)** being rejected during review. More critically, when executing a refactoring task involving six files, limited by the context window capacity, Claude "forgot" the earlier logic by the time it reached the fourth file, leading to code output that was inconsistent and incoherent.

"It feels like managing an intern with amnesia every day," Xiaobing complained to Xiaoxue and Kage, *"incredibly capable, but with no memory of the project context."*

Kage didn't rush to comfort her, but instead fired off three questions:

First, did you configure CLAUDE.md in your project root directory?

Second, for that refactoring task involving six files, why didn't you break it down for sub-agents to handle collaboratively?

Third, since code standards are so important, why rely only on verbal instructions instead of packaging them as a Skill so the model can load them automatically?"

Xiaobing was left speechless, unable to answer a single question.

"The root of the problem isn't that the Claude model isn't smart enough," Kage pointed out, "but that you've only scratched the surface of its potential by using only its basic conversational ability. In fact, Claude Code itself is a complete Agent framework, with a built-in memory system, skill modules, sub-agent delegation mechanism, event hooks, and external tool protocols. Every pain point you've encountered has a corresponding solution at the framework level—you just didn't know about them yet."

This conversation lasted a full two hours. Near the end, Xiaobing sighed, *"I thought I was harnessing an AI assistant, but in reality, I was just using a fancy chat window."*

From Command-Line assistant to agent framework

The issues Xiaobing encountered: memory loss, style drift, and context overflow in long tasks are bottlenecks that nearly everyone applying Claude Code to real-world engineering development will face. These phenomena may appear to be three separate bugs (defects), but in fact, they all point to the same root cause: mistaking an **Agent** framework for a simple chat tool.

The essence of Claude Code is that it is a programmable, extensible, and composable **Agent** framework.

This is not just marketing hype. These three keywords correspond to core engineering capabilities.

- **Programmable:** This means you can drive it directly with Python or TypeScript code, seamlessly embedding it into CI/CD pipelines or automation scripts, fully liberating human resources without the need for manual terminal supervision
- **Extensible:** This means you can inject capabilities such as memory modules, skill libraries, and event hooks via configuration files, without touching or modifying the core code
- **Composable:** This means these extension modules can be flexibly orchestrated, like Lego bricks freely assembled, enabling the construction of complex and efficient workflows

When combined, these three aspects outline not a single tool, but a robust infrastructure framework, a cornerstone for developers to build their own AI work systems.

Kage vividly compared Claude Code to an iceberg and drew it on the whiteboard, as shown in *Figure 1.1*.

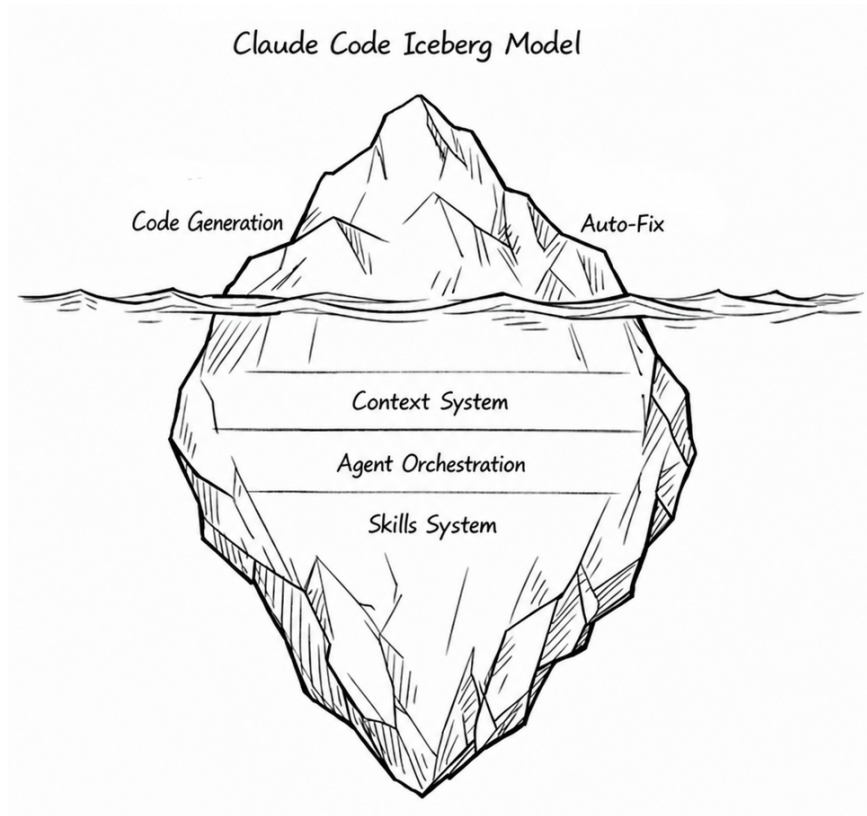


Figure 1.1: Claude code and the iceberg analogy

Figure 1.1 Claude Code is like an iceberg, with many core features hidden below the surface. Above the waterline is just the tip of the iceberg, corresponding to Xiaobing's daily usage scenarios: terminal conversations, code generation, and command execution. The massive foundation hidden below the waterline is a series of key technical architectures that precisely address the challenges Xiaobing encountered.

- **CLAUDE.md**: Cures the "amnesia" problem by writing project standards into a **configuration file** once, which is then automatically loaded in every session, eliminating the need for repeated restatement
- **Skills**: Ends style drift by making code review standards configurable and institutionalized, completely replacing the uncertainty of verbal reminders
- **Sub-agents**: Resolve context overflow by breaking a six-file refactoring task into six independent context units, enabling parallel processing and logical isolation

- **Hooks:** Automatically trigger security checks or log recording during tool invocation, building a defensive programming mechanism
- **MCP (Model Context Protocol):** Breaks data silos, giving Claude the ability to call external databases and APIs
- **Headless mode:** Supports unattended operation in CI/CD pipelines, enabling true automated delivery
- **Agent SDK:** Allows complex multi-step Agent workflows to be orchestrated via code, enhancing task execution flexibility
- **Plugins ecosystem:** Packages the above capabilities for efficient internal team distribution and reuse

"Every pain point you've encountered," Kage pointed to the massive foundation beneath the iceberg, "has a corresponding mechanism here to resolve it. The problem isn't that the tool itself isn't powerful enough, but that you haven't yet unlocked its true potential. Now, I hope you can take a step back and view Claude Code from a global perspective, mastering its full scope and unleashing its true power."

Xiaobing, hearing this, reflected: *"The way I was using it before was essentially inefficient 'manual scripting', repeatedly feeding Claude the same information, manually breaking down tasks, and checking style by eye; whereas the framework-based approach turns all of this into configurable and automated processes."*

Kage nodded in confirmation: *"Exactly. Every time you start a new conversation, you have to restate the tech stack; every code review, you have to describe the standards manually; every code submission relies on personal memory for checks—all of these are essentially inefficient 'manual scripting' operations. The framework-based approach is: solidify the tech stack in CLAUDE.md once and for all; package review standards into a Skill for automatic loading; configure submission checks as Hooks to ensure nothing is ever missed."*

This paradigm shift from "step-by-step instructions" to "system configuration" not only greatly improves efficiency but also profoundly reshapes the quality of human-AI collaboration. Once established, configurations are never forgotten, never neglected, and never compromised by late-night overtime. This is the essence of engineering: distilling individual wisdom into institutional standards.

Next, we will dissect the internal structure of this framework in depth.

Four-Layer architecture model

Before any complex system can be truly understood, a clear "map" is needed. Just as an architect draws blueprints before construction, and a software engineer establishes

architecture before coding, the complete technical stack of Claude Code can also be abstracted as a four-layer architecture model, as shown in *Figure 1.2*. From bottom to top, these four layers are: the memory layer, the extension layer, the integration layer, and the programming layer. Deeply understanding the meaning and boundaries of these four layers is the first step to mastering the entire Claude Code system.

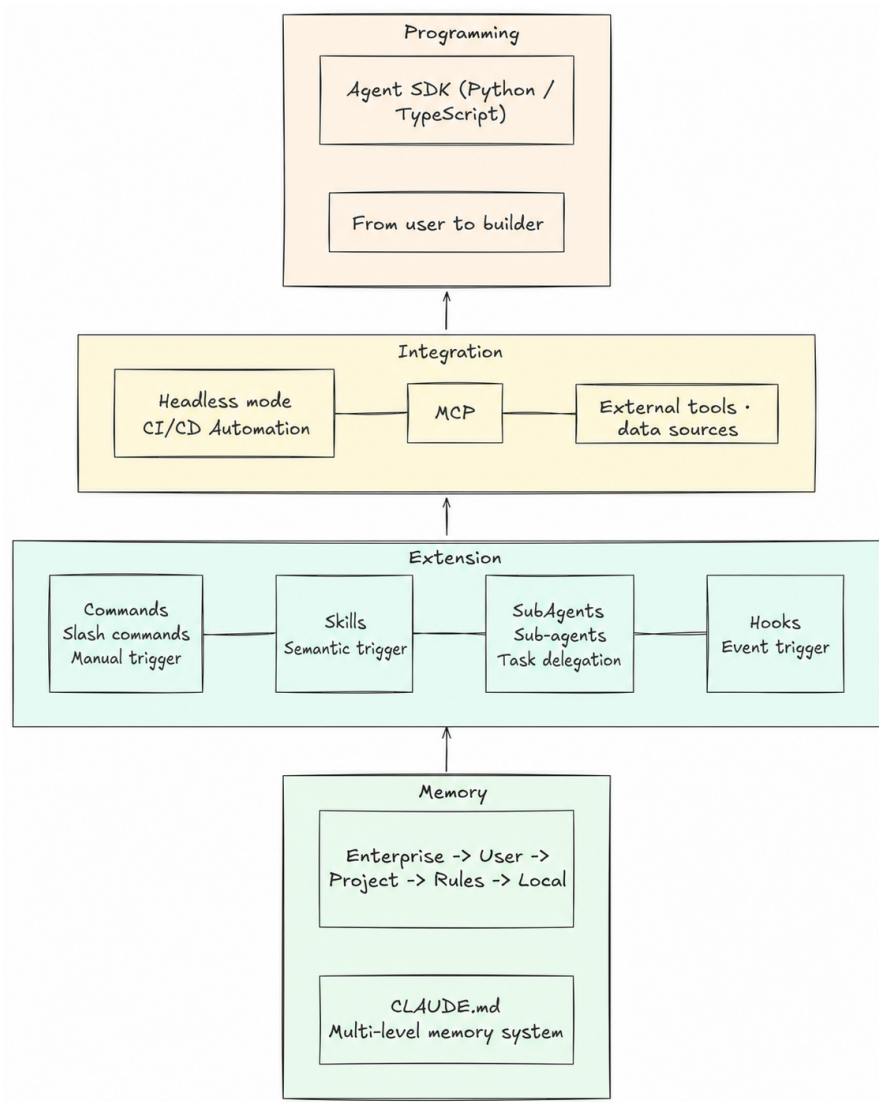


Figure 1.2: The four-layer architecture model of Claude Code

Kage vividly compared these four layers to a skyscraper.

- The memory system of the memory layer is the foundation buried deep underground. Without this foundation, everything built above would be at risk of collapse at any moment.
- The four major extension components of the extension layer form the main floors of the building, carrying the core functions of daily operations and representing the most directly perceived value for users.
- The integration capabilities of the integration layer are like the hidden but crucial water and electrical networks, tightly connecting the building to the vast external infrastructure, ensuring a continuous flow of energy and information.
- The Agent SDK of the programming layer is the architect's studio on the top floor. Here, you are no longer just a resident, but have the supreme authority to design new buildings and restructure spatial logic.

"Looking from the bottom up is the builder's perspective, focusing on foundations, and support; looking from the top down is the user's perspective, focusing on function, and experience," Kage emphasized. "This is a principle worth remembering."

Memory layer: the CLAUDE.md system

The foundation of a building determines the load-bearing capacity of the entire structure. The memory layer of Claude Code is the same: it directly defines the depth of Claude's "understanding" of a project.

In essence, Claude is a stateless model. Every time a new conversation starts, it knows nothing about the project background, tech stack choices, code standards, or team conventions. If no configuration is made, developers must repeatedly restate basic settings such as "use TypeScript instead of JavaScript," "prefer pnpm over npm," or "set Vitest as the testing framework." This mode, akin to introducing yourself anew every day, is obviously highly inefficient.

The memory system thoroughly resolves this pain point through the `CLAUDE.md` file. Essentially, it serves as an "employee handbook for AI," which is automatically loaded into Claude's context at the start of every conversation. What's even more ingenious is that this memory mechanism is not monolithic, but rather constructs a five-level memory hierarchy (see *Figure 1.3*), with each level corresponding to a different scope of application.

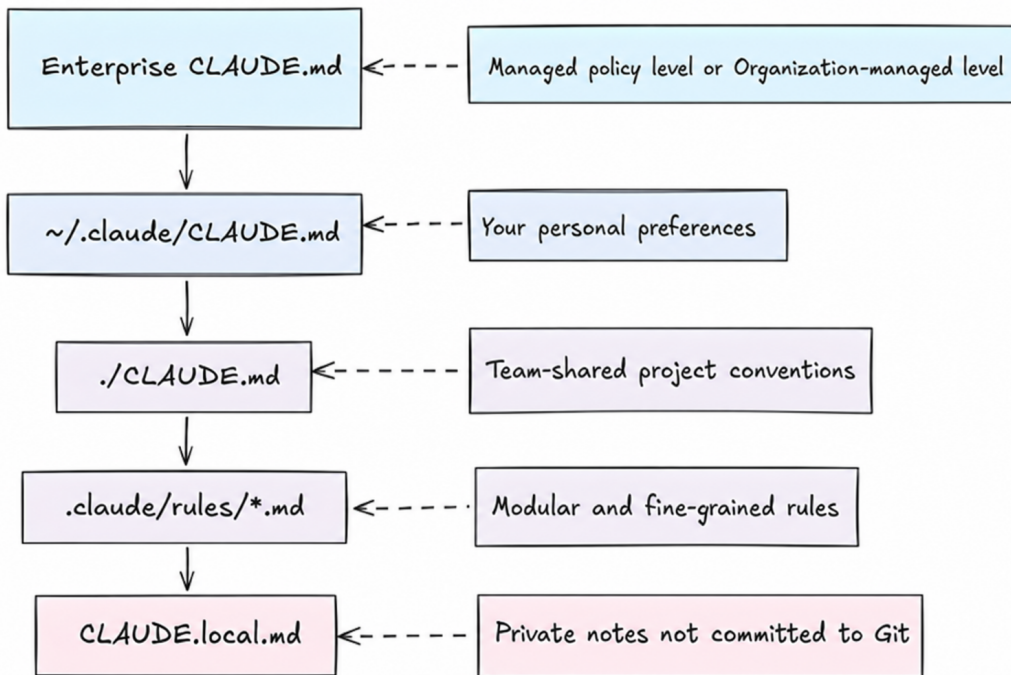


Figure 1.3: The Five-Level Memory System of Claude Code

The engineering philosophy behind this layered design is very similar to the cascading priority mechanism in CSS, or the four-level override model in software configuration—"global–user–project–local": each level can override the settings of the previous one, and the more specific the level (such as local), the higher its priority.

A typical project-level CLAUDE.md file is as follows:

```

# Project Memory CLAUDE.md

## Tech Stack
- Language: TypeScript 5.x (strict mode enabled)
- Framework: Fastify + Prisma
- Testing: Vitest (all public functions must include unit tests)
- Package Management: pnpm (npm or yarn are prohibited)

## Code Standards
- Component Style: Prefer functional components; class components are forbidden
- Error Handling: Always return using the Result type; throwing exceptions
  
```

```
directly is strictly prohibited
- Commit Convention: Follow the <type>(<scope>): <description> format

## Common Commands
- pnpm dev — Start the development server
- pnpm test — Run the test suite
- pnpm lint — Run code linting
```

With the `CLAUDE.md` file, Claude will automatically "study the manual" every time a conversation starts, thereby eliminating misuse of the tech stack and guesswork about code style. As Xiaoxue put it: *"Ever since we created the CLAUDE.md file, I no longer have to repeatedly remind Claude in every conversation that 'we use pnpm, not npm.'"*

This seemingly simple mechanism is, in fact, the cornerstone of the entire framework. All upper-layer configurations and behaviors, such as Skills, Commands, and Hooks, are built upon Claude's precise understanding of the project context. Just as a building's foundation may not be as eye-catching as its exterior decoration, without it, all decoration would have nothing to attach to. *Chapter 2* will delve into this seemingly plain yet crucial memory system.

Extension layer: the four major extension components

If the memory layer is the foundation, then the extension layer is the main structure of the building—it is what gives Claude its richest capabilities in daily use. This layer consists of four major components: Commands (slash commands), Skills, SubAgents, and Hooks. They operate independently, yet can be flexibly combined.

Among them, **Commands** are the most intuitive extension method. When you enter commands like `/review`, `/commit`, or `/deploy` in the terminal, Claude will execute a standardized workflow based on predefined templates. Essentially, this encapsulates high-frequency, fixed-step operations into "one-click" shortcuts—much like quick actions on a smartphone, where a single tap triggers a series of preset actions.

From a design pattern perspective, Commands are a textbook application of the Command Pattern: they encapsulate requests as objects, allowing the client to handle different requests in a parameterized way. Command definition files are stored in the `.claude/commands/` directory, with each Markdown file representing an independent command. For example, a file named `review.md` might contain a code review checklist and output format specifications; simply entering `/review` prompts Claude to strictly follow the checklist and review the current code item by item.

Note**Kage's notes**

In the current Claude Code, custom commands are implemented through the Skills system. This chapter still names Commands separately as a conceptual trigger pattern: manually invoked slash workflows, as opposed to semantically triggered Skills.

Skills follow a completely different design path. Skills live in `.claude/skills/<skill-name>/SKILL.md` (the legacy `.claude/commands/` location still works); sub-agent definitions live in `.claude/agents/<agent-name>.md`. They do not require manual triggering by the user; instead, Claude automatically determines and loads the relevant skill based on the semantic context of the current task. Each Skill is essentially a directory containing a `SKILL.md` file. In this file, the skill's name, description, and the list of allowed tools are declared via YAML Frontmatter (metadata at the top). Claude uses the content of the description field to dynamically decide "whether this Skill needs to be activated for the current task." This implicit triggering mechanism makes Skills an ideal carrier for domain knowledge—whether it's financial analysis methods, API design standards, or best practices for a specific framework, all can be encapsulated within.

For example, when you discuss financial metrics with Claude, it will automatically load the "Finance Skill"; once the topic shifts to code architecture, that Skill will quietly step aside. This "load on demand, remain silent when idle" mechanism greatly enhances the fluidity and intelligence of interactions.

Below is a typical `SKILL.md` example.

```
---
name: code-reviewing
description: >
Review code for best practices and potential issues.
Use when the user asks for code review or mentions
reviewing changes.
allowed-tools:
- Read
- Grep
- Glob
---
```

Code Review Guidelines

You are a professional code reviewer...

Sub-agents are designed to address a deeper bottleneck: the limitation of the context window.

Claude's context window is like a desk of fixed size: the workspace is limited, and the more files you pile on, the more cluttered it becomes. When a task involves a massive number of files (for example, reviewing all code changes in an entire PR), forcing all the details into the main conversation's context quickly leads to a "desk overflow," causing Claude to lose track and even miss critical information.

The sub-agent's solution is to create an independent context space for specific tasks—this is akin to setting up a new workstation next to the main desk and assigning a specialist to handle the subtask there. Once the task is complete, only the conclusion (a concise report) needs to be relayed back to the main desk, rather than transferring all the working drafts. This is much like a manager delegating tasks to subordinates: the manager doesn't need to scrutinize every line of code, only the review conclusions and improvement suggestions.

Sub-agent definition files are located in the `.claude/agents/` directory, with each Markdown file detailing a sub-agent's role, scope of authority, and behavioral guidelines. More importantly, you can set strict tool permissions for sub-agents—for example, specifying that the "code reviewer" only has permission to read files and is strictly prohibited from making any modifications. This design powerfully embodies the Principle of Least Privilege at the core of the AI Agent architecture.

Hooks are the only component among the four major modules with "interception" capabilities. If Commands define "what" Claude does, and Skills guide "how" Claude does it, then Hooks decide "whether" Claude is allowed to do it.

They are automatically triggered at specific event nodes, the most commonly used hooks including before tool invocation (`PreToolUse`), after tool invocation (`PostToolUse`), and before response generation ends (`Stop`). The full event list is documented in the Claude Code hooks reference. Developers can insert checks, interception, or enhancement logic at these critical moments. This mechanism is strikingly similar to the concept of middleware in web frameworks.

For example, by configuring a Hook in `.claude/settings.json`, you can achieve the following: whenever Claude attempts to execute `git commit`, a lint check is automatically run; if the check fails, the commit operation is directly blocked. This "invisible guardian" role makes Hooks a powerful tool for ensuring code quality and security compliance.

Below is a typical configuration example.

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "python .claude/hooks/safety_check.py"
          }
        ]
      }
    ]
  }
}
```

The meaning of the above configuration is this: whenever Claude is about to invoke the Bash tool, the `safety_check.py` script runs first. Note the shape of the entry — a matcher paired with a hooks array, each element declaring its own type and command. There is no "blocking" field; a hook blocks the tool call by exiting with code 2, and whatever it writes to `stderr` is fed back to Claude as the reason. *Chapter 5* covers this contract in full.

This event-driven guardian mechanism has mature counterparts in software engineering—whether it's Git Hooks, database triggers, or message queue consumers, their essence all follows a unified pattern: when a specific event occurs, preset logic is automatically executed.

These four major components together form the core foundation of Claude Code's extensibility. It is important to emphasize that they are orthogonal: they support independent learning and standalone use, but can also be flexibly combined to build more powerful workflows.

This orthogonal design closely follows the **Single Responsibility Principle (SRP)** in software engineering: each component focuses on solving a specific type of problem ("do one thing well"), yet through organic combination, they can unleash a system with seemingly limitless capabilities.

Note**Kage's notes**

At the beginner stage, do not rush to master all four components at once. It is recommended to follow this step-by-step learning path: first, write a solid `CLAUDE.md` file; next, learn **Commands**, which is the most intuitive way to extend functionality; then, master **Skills** to experience the magic of semantic auto-triggering; after that, introduce **Hooks** to establish a safety and quality safeguard mechanism; finally, take on the challenge of sub-agents to tackle large-scale complex tasks. For each component you master, be sure to practice it in real projects for two weeks before moving on to the next stage. After all, the saying "biting off more than you can chew" applies here as well.

Integration layer: connecting to the outside world

In the skyscraper of the four-layer architecture, the integration layer is like the water and power grid that keeps it running—it extends the core capabilities of **Claude Code** to external systems, allowing it to truly integrate into the development ecosystem.

Among these, **Headless** is the key. It enables Claude Code to run automatically in unattended environments, detached from interactive terminals. By passing the task description via the `-p` parameter and specifying the output format with `--output-format json`, **Claude Code** can be seamlessly embedded into any **CI/CD pipeline**, such as **GitHub Actions**, **Jenkins**, or **GitLab CI**.

A typical use case is automated **PR review**: whenever a developer submits a Pull Request, the CI pipeline automatically invokes Claude Code for code review and writes the results back as a comment directly on the PR page. The entire process requires no human intervention, truly achieving "unattended" intelligent collaboration.

Below is an example command for invoking Claude Code in a CI/CD pipeline.

```
# Invoke Claude Code in a CI/CD pipeline
claude -p "Review the most recent code changes, focusing on security risks and
performance issues" \
--output-format json \
--max-turns 10 \
--allowedTools Read,Grep,Glob
```

Each parameter in the above command carries a clear engineering intent: `-p` activates Headless mode for non-interactive operation; `--output-format json` ensures the output can be automatically parsed by scripts; `--max-turns` precisely controls costs by limiting the maximum number of interaction rounds; and `--allowedTools` uses a whitelist mechanism to strictly limit the range of tools Claude can invoke, thereby establishing a strong security barrier.

MCP, on the other hand, unlocks another dimension of integration—it empowers Claude to deeply interact with external tools and data sources. If Headless enables Claude to "go out" and integrate into pipelines, then MCP allows external capabilities to "come in" and empower the model.

You only need to configure the MCP server in the `.mcp.json` file at the project root to easily extend Claude's capability boundaries: whether it's accessing databases, calling third-party APIs, operating on GitHub Issues, or querying Jira tickets, all can be seamlessly connected with a single configuration. Anthropic vividly describes MCP as the "USB-C of the AI era"—a standardized interface protocol that allows any tool or data source to be integrated into the Claude Code ecosystem in a unified and regulated way.

Below is an example configuration for a GitHub MCP server.

```
{
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_TOKEN": "your-token-here"
      }
    }
  }
}
```

The design philosophy behind MCP is worth deep consideration. In traditional AI tool integration, connecting to each new external system often requires writing a large amount of customized adapter code. MCP, however, achieves "interface unification" between tool providers and AI consumers by defining a standardized protocol (covering the three core capabilities of Tools, Resources, and Prompts). This is reminiscent of the transformative impact REST APIs had on unifying frontend and backend communication in the Web era—standardization does not impose limitations, but rather fosters a thriving ecosystem.

Headless and MCP together form the two pillars of the integration layer.

- Headless is oriented toward automated pipelines, enabling seamless embedding of Claude Code into existing CI/CD systems
- MCP is oriented toward the tool ecosystem, bringing external data and services into Claude Code's toolbox of capabilities

Together, they propel Claude Code from a standalone terminal tool into a foundational component that can be flexibly integrated into any tech stack. If the extension layer strengthens Claude Code's core functions "internally," then the integration layer builds a network of connections "externally"—the two complement each other and are both indispensable.

Programming layer: agent SDK

The top floor of a skyscraper is the architect's studio—here, you are no longer just a resident, but a creator designing new blueprints.

The Agent SDK sits at the pinnacle of the Claude Code technology stack and serves as a crucial "watershed." In the three layers below (memory, extension, and integration), developers primarily extend Claude's behavior boundaries through configuration files and command-line operations, essentially as users. But once you cross the threshold of the Agent SDK, you officially enter the realm of builders: developers can use Python or TypeScript to write code, directly invoking Claude Code's core capabilities to build entirely new AI Agents from scratch.

Below is an example of using the Agent SDK to build a code health check Agent.

```
import asyncio
from claude_agent_sdk import query, ClaudeAgentOptions,
    AssistantMessage, ResultMessage

async def main():
    options = ClaudeAgentOptions(
        allowed_tools=["Read", "Grep", "Glob"],
        max_turns=15,
    )

    async for message in query(
        prompt="Analyze the code quality of all Python files
            under src/ and provide a health score.",
        options=options,
    ):
        if isinstance(message, AssistantMessage):
```

```
    for block in message.content:
        if hasattr(block, "text"):
            print(block.text)
    elif isinstance(message, ResultMessage):
        print(f"\n[done] cost=${message.total_cost_usd}")
asyncio.run(main())
```

The deeper significance of the above code is this: developers are no longer confined to the terminal interface of Claude Code, but can instead drive Claude to perform tasks directly within their own applications. This means that Claude's analytical and reasoning capabilities can be seamlessly embedded into any scenario—scheduled tasks, web services, data processing pipelines—wherever your code can reach, Claude's capabilities can extend.

For the vast majority of developers, the first three layers (Memory, Extension, Integration) are sufficient to handle everyday development scenarios with ease. However, if your vision is to build a team-level AI development platform or design a multi-agent collaborative automation system, then the Agent SDK is an essential path.

With the Agent SDK, you can build batch code review tools, project health dashboards, automated test report generators, or even orchestrate multiple Claude Agents into an efficient collaborative team. For example, the first Agent specializes in code reading, the second in writing tests, and the third in conducting reviews, together forming a fully automated intelligent pipeline.

The existence of the Agent SDK also reveals an intriguing fact: **Claude Code and the Agent SDK run the same agent loop, tools, and context management. The SDK packages that runtime as a library so developers can embed the same harness that powers Claude Code in their own applications.** Everything you see in the terminal—reading files, writing code, executing commands—is, in essence, a tool invocation at the Agent SDK layer. Once you grasp this, your understanding of Claude Code will elevate from "a product" to "a concrete implementation of an architectural pattern."

In subsequent chapters, we will delve into practical uses of the Agent SDK. For now, it is crucial to establish one core understanding: the Agent SDK is the pinnacle of this architectural edifice and marks the starting point for the leap from "configuration-driven" to "code-driven" development.

Underlying perspective: harness and the agentic loop

Having understood the "external perspective" of the four-layer architecture, let's dig deeper into a more fundamental concept: the **Harness**.

Anthropic's official documentation offers a succinct definition: "Claude Code serves as the **agentic harness** around Claude: it provides the tools, context management, and execution environment that turn a language model into a capable coding agent."

In short, Claude Code is an **intelligent orchestration framework** wrapped around the Claude model. The native model is only capable of generating text. It is the Harness that endows it with the "hands and feet" to read files, write code, retrieve codebases, and execute commands in the terminal. Without the Harness, Claude is merely a "brain" with intelligence but unable to act.

The original meaning of "harness" is the set of straps and reins fitted to a horse. While the horse has strength, without the harness's guidance, it cannot pull a vehicle. The harness does not change the horse's strength itself, but rather the way that strength is **transmitted and controlled**.

Expressed as a formula: **Agent** = Model + Harness.

If the model is the engine, then the Harness is all the infrastructure built around it: tool systems, permission controls, context management, session persistence, event hooks, and the core loop that drives it all: the **Agentic Loop**.

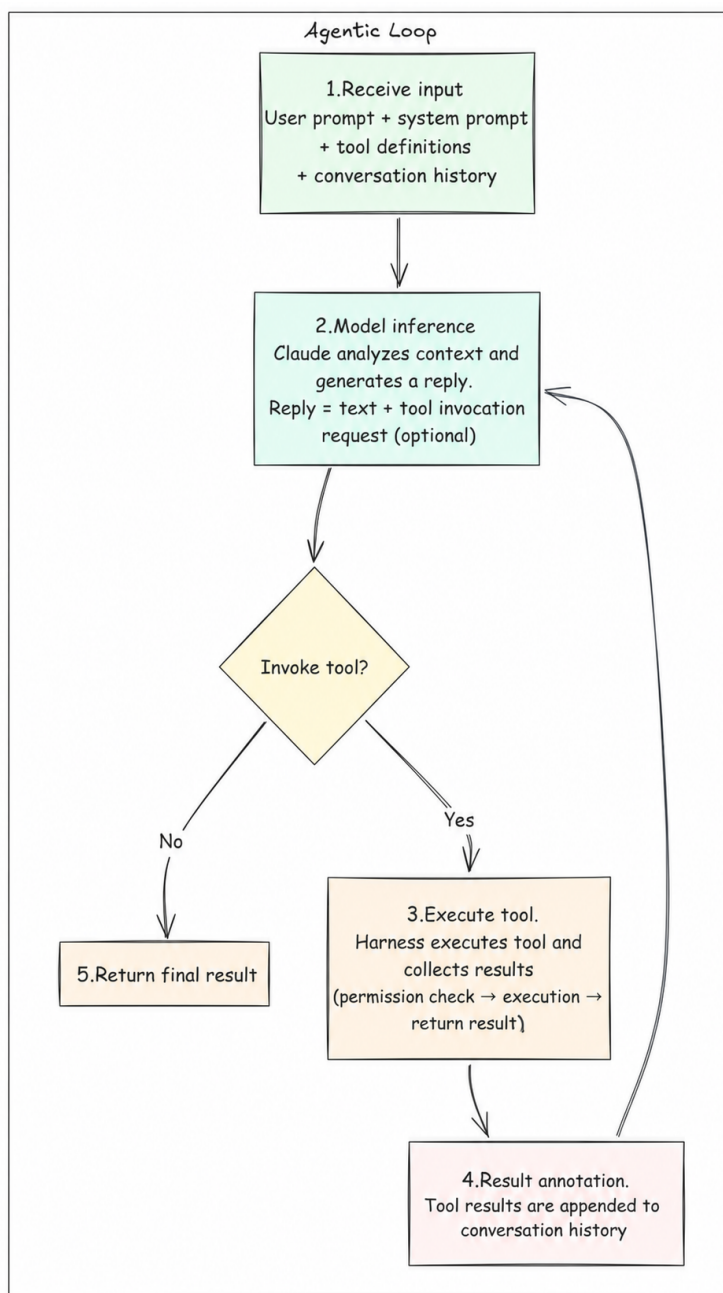


Figure 1.4: The Agentic Loop—the heart of the Harness

This looping mechanism profoundly reveals the source of Claude Code's core capabilities. When you submit a bug, it does not simply "take a glance and guess the answer." Instead, it undergoes a repeated process of observation, hypothesis, and verification: first consulting error logs, then searching relevant code, deeply understanding the context, and only then proceeding to fix the issue.

When does the loop terminate? It depends on two conditions: first, the model actively stops (i.e., determines the task is complete and no longer initiates tool calls); second, the maximum number of turns is reached (set by the `--max-turns` parameter to prevent infinite loops).

Recent benchmark results have made this point concrete: changing the harness can substantially change an agent's performance even when the underlying model stays fixed (<https://www.langchain.com/blog/improving-deep-agents-with-harness-engineering>).

In the TerminalBench benchmark, simply optimizing the **Harness** enabled the same model to rise from 30th place to the Top 5 on Terminal-Bench 2.0 (52.8% → 66.5%, a 13.7-point gain). In other words, the Harness is more important than the model itself. Every link in the four-layer architecture you've learned—whether writing `CLAUDE.md`, setting tool permissions, integrating hooks, or connecting the MCP, is, in essence, a meticulous refinement of the Harness.

Upon hearing this, Xiaobing suddenly understood: *"I used to think Claude Code was just a shell for the Claude model. Turns out, this 'shell' is where its true value lies."*

Component relationships and collaboration

After gaining a deeper understanding of each component in the four-layer architecture, the more critical question is: how do they work together? How should you choose between them in different scenarios? How does a real task flow through this system?

Comparison of trigger mechanisms

The most easily confused aspect of the four major extension components is their trigger mechanisms. *Table 1.1* compares them across five dimensions.

Trigger	Commands	Skills	Human or AI	Hooks
Is memory required?	Yes (needs to know the command name)	Financial analysis, API design	Yes (needs to know the agent name)	No (effective upon configuration)

Typical scenario	CLI command	AI	Code review, log analysis	Pre-commit checks, post-save formatting
Engineering analogy	Claude automatically matches based on semantics	Strategy pattern (automatic algorithm selection)	Thread pool (isolated execution)	Middleware/ Git Hooks
User manual input	/review, /deploy	Sub-agent	System event auto-trigger	Application scenario
Human	No (automatic discovery)	User-specified or proactively delegated by Claude	System	Recommended component

Table 1.1: Comparison of the Four Major Extension Components

A straightforward mnemonic is as follows:

- **Commands:** "You tell it what to do" (responds to user commands)
- **Skills:** "It knows what to do itself" (autonomously judges and executes)
- **Sub-agents:** "It delegates tasks to others" (coordinates sub-agents to complete tasks)
- **Hooks:** "No matter who does it, checks are executed at this step" (automatically intercepts or processes at specific workflow nodes)

After reading the above summary, Xiaoxue insightfully pointed out: *"Both Commands and Hooks are deterministic triggers—the former is explicitly triggered by user commands, the latter is automatically triggered by specific events; whereas Skills and sub-agents involve AI's autonomous judgment—the former is enabled if the AI deems it necessary, the latter if the AI decides to delegate the task."*

Kage nodded approvingly: *"This dichotomy of 'deterministic vs. AI judgment' is indeed another key perspective for understanding the differences among the four major extension components. In engineering practice, for scenarios with extremely high security requirements (such as sensitive information interception), deterministic Hooks should be prioritized; for scenarios requiring greater flexibility (such as domain knowledge matching), Skills triggered by AI judgment are more suitable."*

Data flow: the journey of a request

To more intuitively understand the collaborative mechanism of each component, let's trace the complete journey of a request through the Claude Code system.

Suppose a developer enters the command in the terminal: "Help me review the recent code changes in the src/payment/ directory to ensure there are no security risks."

The processing flow of Claude Code is shown in *Figure 1.5*.

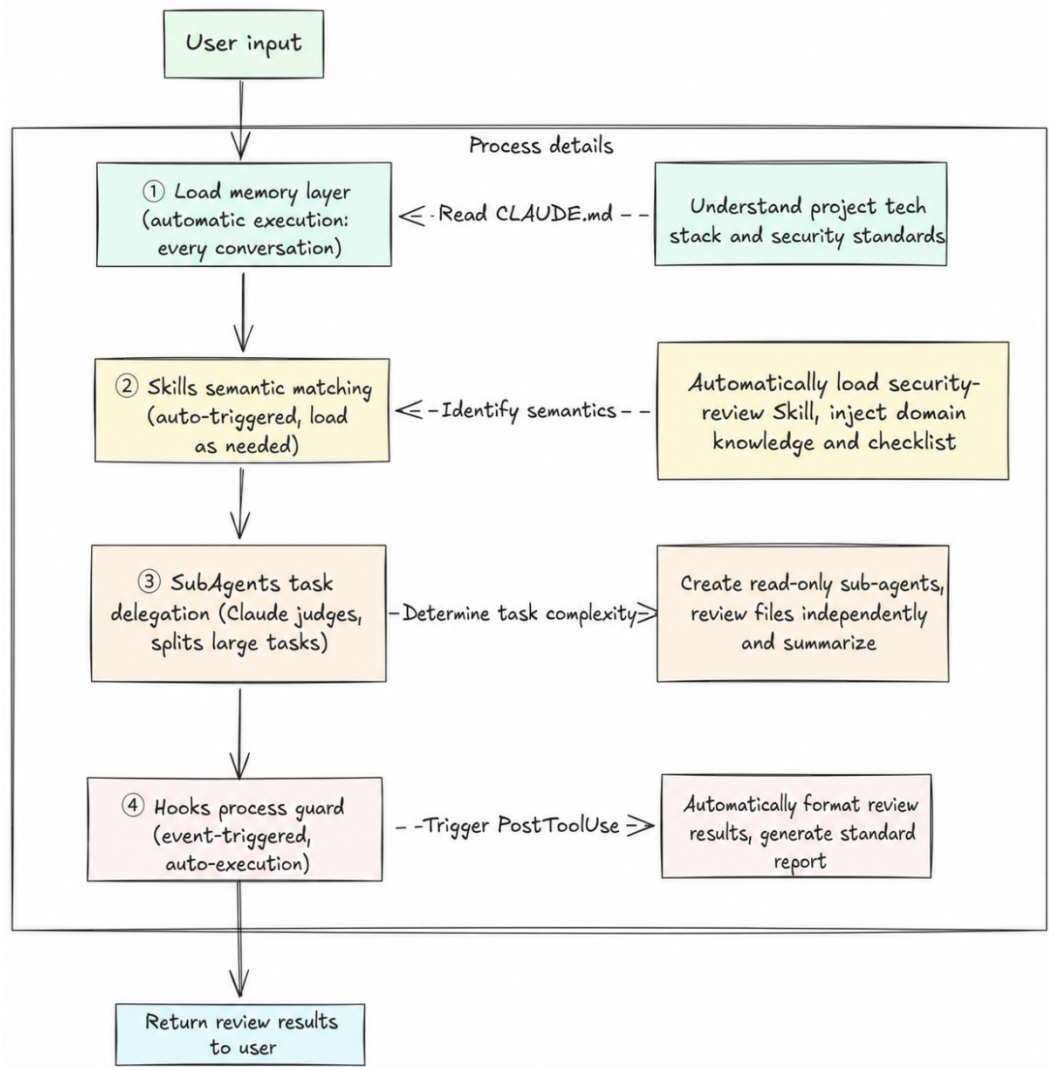


Figure 1.5: Processing Flow of Claude Code

In the process shown in *Figure 1.5*, each component in the Memory and Extension layers plays its role: the Memory layer (CLAUDE.md) builds the project context, Skills inject domain knowledge, sub-agents provide isolated execution environments, and Hooks strictly guard quality checkpoints. The entire process requires no additional user intervention—everything is automatically orchestrated after the user inputs that single natural language command.

This is the true essence of "composability." Each component is like an independent building block; once combined, they form a complete automated pipeline. This perfectly aligns with the classic software engineering principle of "Separation of Concerns": each module focuses on a single responsibility and collaborates through clear interfaces. The component design of Claude Code is a model example of this principle.

- CLAUDE.md focuses on "know-what"
- Skills focus on "know-how"
- Sub-agents focus on "who-does-it"
- Hooks focus on "guard-rails"

It is precisely because of these clear boundaries of responsibility that these components can be freely combined without interfering with one another.

It is worth noting that the above example demonstrates an "all-in-one" collaboration scenario. In actual engineering practice, most tasks do not require all components: simple code generation may only need the cooperation of memory and Skills, while routine commit operations might involve only Hooks. The core value of components lies not in "using them all," but in "calling them as needed, ready for use at any time."

Plugins: packaging and distribution of combinations

"Xiaobing, you once asked me: 'What exactly is the relationship between Skills and Plugins?'" Kage drew a big box on the whiteboard and then revealed the answer: *"It's simple—Plugins are not a new capability, but a packaging mechanism."*

Imagine you spent three weeks meticulously building a Claude Code extension system for your team: it includes two Skills (code review, API design), one sub-agent (security auditor), and three Hooks (pre-commit lint check, post-save formatting, sensitive information interception). Now, a new colleague joins and urgently needs to reuse this setup. If you were to copy files manually, just figuring out "which files go in which directories" could easily take half a day.

The Plugins mechanism elegantly solves this problem. Essentially, it is a standardized packaging format: through a `plugin.json` manifest file, Skills, Commands, sub-agents, Hooks, and MCP configurations are all encapsulated together. Once packaged, a single installation command automatically deploys all configurations in place.

This is akin to what npm is for Node.js or Docker is for container images—the core value of Plugins lies in distribution and reuse, not in creating a new dimension of capability.

Clarifying this concept is crucial. Beginners often mistakenly see Plugins as some kind of "more advanced Skills," but in fact, the two operate on completely different dimensions: Skills define **types of capabilities**, while Plugins define **the form of distribution**.

A Plugin is like a container, capable of encapsulating Skills, Commands, Hooks, sub-agents, and other components. Its core concern is not "what task to execute," but "how to package and share."

For individual developers, there may be no need to deliberately create Plugins; configuring components locally in the project may suffice.

For team leaders, however, Plugins are a powerful tool for standardizing best practices and enabling one-click configuration for new team members.

Technical selection guide

Faced with a variety of components, a core question arises: for your specific needs, which one should you choose?

To help with this, here is the Claude Code component selection decision tree (see *Figure 1.6*) and application scenario comparison (see *Table 1.2*) to help you make quick judgments.

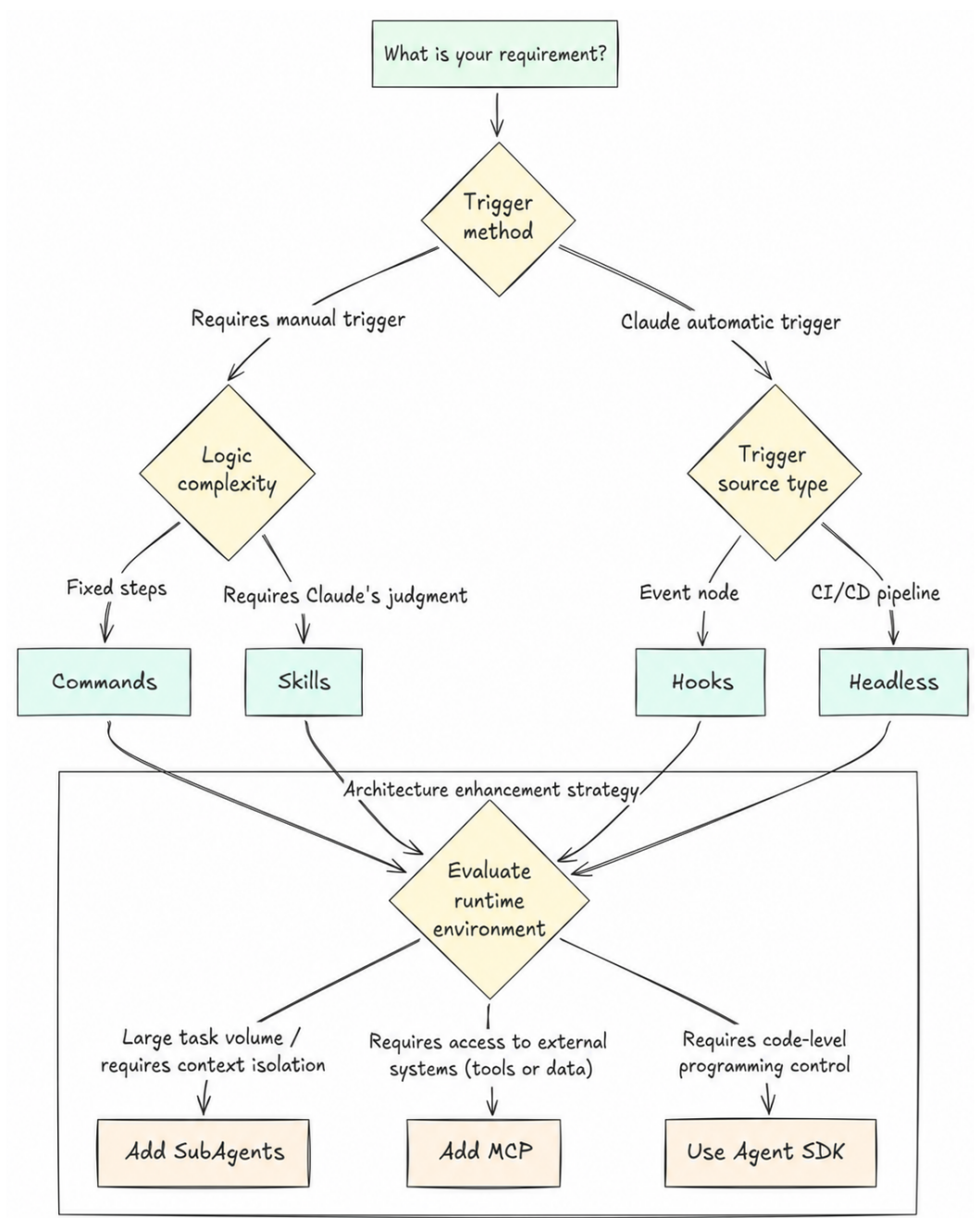


Figure 1.6: Claude Code Component Selection Decision Tree

Core reason	Execute a fixed checklist for each release	Steps are clear and fixed, suitable for one-click triggering via manual command
Automatically receive professional guidance when discussing financial issues	Commands	Domain knowledge is automatically matched based on semantics and loaded as needed; there is no need for users to manually specify or remember commands
Review large PRs to avoid context overflow	Skills	Sub-agent
Use independent sub-agents to isolate context, only returning concise conclusions to the main session, effectively managing token consumption	Automatically block sensitive information leaks before code submission	Event-driven mechanism ensures mandatory interception and checks before specific operations (such as tool invocation), with no omissions
Automatically trigger code review after PR submission	Hooks (PreToolUse)	Suitable for unattended environments, can be embedded in CI/CD pipelines, automatically executing review tasks via command
Let Claude operate GitHub Issues	Headless Commands	Seamlessly connect to external systems (such as GitHub API) via standard protocols, extending the tool boundaries of the model
File type	MCP	Path

Table 1.2: Claude Code Component Selection Application Scenario Comparison

For the **decision tree**, follow this self-questioning path. First, ask yourself, "How is this task triggered?" If it requires human initiation, continue by asking, "How complex is the logic?" If the steps are fixed, choose Commands; if Claude needs to make judgments, choose Skills. If it is triggered automatically by Claude, then ask, "Is it triggered at a specific event node, or in a CI/CD

pipeline?" If it's an event node, choose Hooks; if it's in a CI/CD pipeline, choose Headless mode. Next, ask yourself, *"What is the runtime environment?"* If the task load is heavy or requires context isolation, add sub-agents. If you need to access external systems (such as tools or data), add MCP. If you need code-level programming control, use the Agent SDK.

It is important to note that these components are often used in combination rather than as mutually exclusive choices.

A mature team configuration will typically coordinate all components to build a robust automation system.

- The `CLAUDE.md` file lays the foundational context
- Skills inject domain expertise
- Commands provide standardized entry points for manual operations
- Hooks build an automated security defense line
- Sub-agents break down and handle highly complex tasks
- MCP breaks through data barriers with external systems
- Headless mode seamlessly integrates capabilities into the CI/CD pipeline

This is just like a highly efficient enterprise: each role (component) has clear responsibilities and boundaries, yet they work closely together to achieve the final goal.

Note

Kage's Notes

The core principle of technical selection is always "solve the current problem with the simplest solution." Don't fall into the trap of "when you have a hammer, everything looks like a nail": don't delegate every task just because you've learned sub-agents, and don't add guards to every minor operation just because you've mastered Hooks. Over-engineering and under-engineering are equally harmful. Excellent architects understand the art of balance: knowing when to introduce an abstraction layer to enhance scalability, and when a simple `CLAUDE.md` file is all you need for a perfect solution.

Summary

This chapter began with a key shift in perspective: Claude Code is not a command-line version of ChatGPT, but a programmable, extensible, and composable Agent framework. For Claude Code, the conversational interface above the waterline is just the entry point; the true technical depth lies beneath the surface—in the memory system, the four major extension components,

integration mechanisms, and Agent SDK. Understanding this is the first step from "passive use" to "active orchestration."

The four-layer architecture provides a clear blueprint for the entire system. The memory layer is the foundation, giving Claude Code the ability to understand project context; the extension layer is the main body, offering four orthogonal extension methods: Commands, Skills, sub-agents, and Hooks; the integration layer is the pipeline, connecting to the outside world via Headless and MCP; the programming layer is the top floor, enabling developers to move from users to builders. These four layers build upon each other, but also support independent study—you don't need to master everything at once to start practicing.

Clarifying the relationships between components is the core knowledge to internalize from this chapter. Commands represent explicit instructions, Skills carry implicit knowledge, sub-agents enable isolated delegation, and Hooks serve as automated guardians. These four triggering mechanisms cover the full spectrum from manual intervention to automated execution, from explicit invocation to implicit response. They can operate independently or be orchestrated in combination. Plugins, as a standardized packaging mechanism, enable efficient distribution and reuse of these combinations.

In *Chapter 2*, we will delve into the cornerstone of the four-layer architecture—the memory system (CLAUDE.md file). It is both the foundation of the entire building and the best starting point for practicing Claude Code's framework capabilities. A well-crafted CLAUDE.md file is worth more than ten redundant verbal explanations.

Reflection questions

1. Review your current habits when using Claude Code (or other AI programming tools): which operations are still stuck at the "above the iceberg" passive interaction layer? If you introduce the components described in this chapter, which steps could be automated or structurally refactored?
2. In your team's daily development workflow, which steps are best suited for automated guarding via Hooks (such as code standard checks, sensitive information interception)? Which steps are better suited to be solidified as Commands for explicit invocation? List 3–5 specific scenarios and match each with the most appropriate component.
3. This chapter points out that Claude Code's four major extension components follow the "single responsibility principle." Think deeply: if Skills and Commands were merged into a single component (supporting both manual triggering and automatic response), what conveniences would this bring? What potential problems might arise? This design reflects the classic trade-off between "simplicity" and "orthogonality" in software design.

2

Learning From the Past: Engineering Practice of the Memory System

"Those who cannot remember the past are condemned to repeat it."

— *George Santayana*

After the last technical discussion, Xiaobing couldn't wait to try Claude Code in the order service project she was responsible for. She eagerly entered her first request: *"Help me write a user login API."* A few seconds later, Claude generated a seemingly perfect piece of code: based on the Express framework, using JavaScript syntax, and even including the `npm install` command, all in one go. Xiaobing stared at the screen for three seconds before slowly typing the next line: *"Our project uses Fastify and TypeScript..."* She then patiently corrected the model's constraints and restated her requirements, finally obtaining code that met her expectations.

The next day, Xiaobing closed the terminal and started a brand-new session, asking Claude to write an order creation API. This time, Claude was even more enthusiastic, generating a complete set of routes, controllers, and data models in great detail. However, the framework was still Express, the package manager was still npm, and even the directory structure was vastly different from the team's conventions. Staring at the familiar line on the screen—`const express = require('express')`—Xiaobing felt a deep sense of helplessness welling up inside her.

This kind of "AI amnesia" does not stem from a lack of capability in Claude. On the contrary, it demonstrates outstanding programming ability in every independent session. Yet, every

seemingly "reasonable" choice it makes—choosing Express over Fastify, npm over pnpm, JavaScript over TypeScript—without exception, runs afoul of the team's standards.

Why AI needs "Memory"

Frustrated, Xiaobing went to find Kage. After listening to her experience, Kage didn't rush to offer a solution.

"Xiaobing, do you remember how you felt on your first day at work?" Kage asked.

Xiaobing thought for a moment. *"Of course I remember. I didn't know anything—where the code repository was, what the branching strategy was, how the CI process worked... I had to ask my colleagues about everything."*

"That's right. Now imagine if every morning you woke up, you forgot everything you learned at the company the day before, and every day felt like your 'first day on the job.' What would you do?"

Xiaobing shuddered. *"Isn't that exactly what it's like using Claude Code right now?"*

"Exactly," Kage explained. *"For Claude, **every conversation is its 'first day on the job'** (see Figure 2.1). It's like a technically skilled new employee who, on their first day, knows nothing about the company's tech stack and can only guess based on personal experience and industry conventions. The guesses might be logical, but they're not necessarily accurate."*



Figure 2.1: Every conversation is Claude's "first day on the job."

This is a fundamental characteristic of large models: a lack of persistent state between sessions. Every time you close a Claude Code session and start a new one, the Claude model is faced with a completely blank slate. It doesn't remember which framework your project uses, your team's preferred code style, or even the mistake you corrected five minutes ago. This isn't a flaw of any particular model, but a common trait of all current large models—every conversation must build its context from scratch.

From a software engineering perspective, this is the exact opposite of the "Convention over Configuration" principle. Ruby on Rails empowers developers with high efficiency precisely because of its set of sensible defaults: database table names are plural, model names are singular, controllers are placed in the `app/controllers` directory, and developers only need to focus on the exceptions to these conventions.

However, Claude's default "conventions" don't come from project standards, but from its training data—the statistical patterns of all publicly available code on the internet. In this distribution, Express appears more frequently than Fastify, npm is more common than pnpm, and JavaScript samples far outnumber TypeScript. So, when lacking specific project context, Claude naturally gravitates toward the most "mainstream" solutions. While these choices may be logical in general scenarios, they may not fit your team's specific standards.

In other words, what Claude lacks is not capability, but context.

Given this, the solution becomes obvious: since every conversation is Claude's "first day on the job," why not prepare a dedicated "onboarding manual" for it, ensuring it automatically studies it at every startup? This manual is the `CLAUDE.md` file. As a plain Markdown text file, you simply place it in the designated project directory, and Claude Code will automatically scan and load its contents at startup, injecting it as a high-priority user message immediately after the system prompt (not embedded inside the system prompt itself). This also explains why `CLAUDE.md` behaves as guidance rather than enforced configuration. From then on, Claude no longer needs to guess what your project looks like from scratch—every standard and detail is clearly documented in the manual.

This design philosophy is already commonplace in software engineering. We use `.editorconfig` to unify editor behavior, `.eslinttrc` to standardize code style, `tsconfig.json` to constrain TypeScript compilation options, and `.prettierrc` to solidify formatting rules. The essence of these configuration files is to externalize implicit "team conventions" from people's minds into machine-readable declarative rules.

`CLAUDE.md` follows the same logic, with the only difference being its "reader": no longer a compiler or formatting tool, but a large model with natural language understanding. This is the natural extension and evolution of "configuration-driven" thinking in the AI era.

Note**Kage's notes**

Some may ask: "Doesn't Claude Code have the ability to read project files? Can't it just check `package.json` to infer the tech stack?" That's true, but "active probing" not only consumes valuable tool invocation quotas and context window space, but more importantly, `package.json` only records the dependency list and cannot capture deeper information like architectural decisions and team conventions. The core value of `CLAUDE.md` lies in consolidating scattered implicit knowledge into an explicit guidance document, ensuring Claude can build an accurate mental model of the project before it starts coding.

The Five-Layer memory architecture

"I understand the principle, but where exactly should `CLAUDE.md` be placed?" Xiaobing pressed.

Kage smiled slightly. "Good question. The answer is: it doesn't exist in just one place."

Claude Code's memory system adopts a sophisticated layered architecture, divided into five levels from global to local. This design is reminiscent of the configuration loading mechanism in operating systems: just as Linux reads configurations by scanning global settings in `/etc/`, user-specific settings in the home directory, and project-specific settings in the current directory—layering them for increasing specificity—Claude Code's five-layer memory system follows the same design philosophy.

Note

The five-layer model used in this chapter is this book's own way of organizing how memory works. Claude Code's official documentation defines four memory scopes — Managed Policy, User, Project, and Local. The "Rules Directory" layer singled out here (``.claude/rules/``) is not a separate scope in that taxonomy; it is a modular, path-scoped feature within the project scope. We break it out as its own layer to highlight the conditional-rules mechanism.

We can use a running analogy to illustrate this system: the hierarchy of company policy documents.

- The first layer is the "**Group Code of Conduct**" issued by headquarters, an ironclad rule that all subsidiaries must follow and cannot modify. It establishes the company's core values and bottom lines.
- The second layer is the "**Department Operations Manual**" for each department, which applies to all projects within the department and ensures standardization within the same business line.
- The third layer is the "**Project Development Specification**" for each project team, refining the rules down to individual products and specifying core conventions like technology choices and coding styles.
- The fourth layer is the "**Special Operations Guide**" for specific scenarios. For example, the "**Database Migration Operations Specification**" is only referenced when a particular task is involved and is highly scenario-specific.
- The fifth layer is the personal sticky notes at each employee's workstation—completely personalized knowledge, visible only to oneself, used to record temporary debugging tips or personal preferences.

From top to bottom, the scope of application narrows, the granularity of content increases, and management flexibility rises accordingly.

Figure 2.2 shows Claude Code's five-layer memory system.

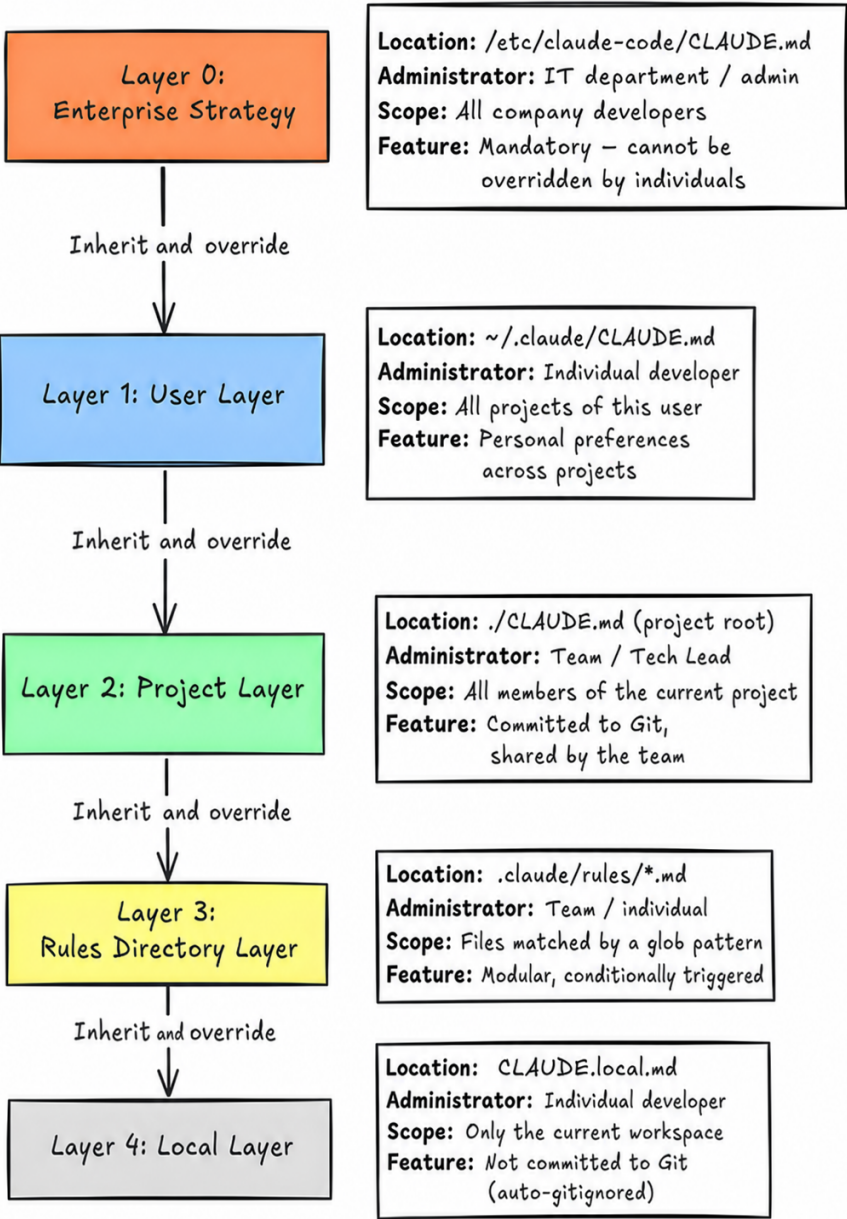


Figure 2.2: The Five-Layer Memory System of Claude Code

Next, let's analyze the responsibilities and typical uses of each layer.

The **enterprise policy layer** forms the "ceiling" of the entire system and is usually inaccessible to developers. This layer is set uniformly by the company's IT administrators through system-level configuration files (macOS /Library/Application Support/ClaudeCode/CLAUDE.md; Linux/WSL /etc/claude-code/CLAUDE.md; Windows C:\Program Files\ClaudeCode\CLAUDE.md) or the Anthropic enterprise management console. Typical policies include: prohibiting code from being sent to external APIs, requiring all database operations to go through an ORM, strictly forbidding hardcoding of secrets in code, and other security red lines. This layer establishes the enterprise compliance baseline: no matter how developers configure the lower four layers, these mandatory constraints cannot be bypassed. For individual developers and small teams, this layer is usually empty and requires no special attention.

The **user layer** is essentially the user's "personal preference profile." This configuration file is located at ~/.claude/CLAUDE.md in the user's home directory and applies to all projects opened on this machine. It is best suited for storing general preferences that are unrelated to specific business logic but reflect personal work habits, such as the choice of communication language (Chinese or English), formatting conventions for Git commit messages, or mapping frequently used personal shortcut commands to specific execution commands. These preferences are highly stable and do not change with project switching.

```
<!-- ~/.claude/CLAUDE.md -->

# Personal Coding Preferences
- Communication language: Replies in Chinese, code comments in English
- Commit conventions: Git commit messages follow the format "type(scope):
description in Chinese"
- Programming paradigm: Prefer functional programming style
- Shortcut command mappings:
- When I say "run tests," automatically execute pnpm test
- When I say "start service," automatically execute pnpm dev
```

The **project layer** is the most frequently used layer in daily development and is the main focus of this chapter. It is represented by the CLAUDE.md file in the project root directory, which must be committed to the Git repository and become part of the project's code assets.

This means that when a new colleague clones the project, Claude Code will automatically read this file. No additional configuration is needed—Claude instantly acquires the complete project context. It's like receiving a dedicated "Project Development Manual" in your welcome kit on your first day at a new job—open it, and the technology stack choices, directory structure, coding standards, and common commands are all clear at a glance.

The **rules directory layer** is a sophisticated, advanced design aimed at solving the pain point that "a single `CLAUDE.md` file cannot accommodate all complex rules." You can deploy multiple independent Markdown files under the `.claude/rules/` directory, with each file focusing on a specific topic (such as database conventions, API design styles, or testing strategies). More importantly, this layer supports a conditional triggering mechanism—**rule files are dynamically loaded only when operating on specific files or entering specific contexts, thus avoiding irrelevant information cluttering the context. We will explore this mechanism in depth and its practical usage in the Conditional Rule System section.**

The **local layer** is the developer's "personal memo," corresponding to the `CLAUDE.local.md` file. This file is not committed to the Git repository (Claude Code adds `CLAUDE.local.md` to `.gitignore` only when you run `/init` and choose the personal option; if you create the file manually, add it to `.gitignore` yourself) and is intended for information that is highly valuable to the individual but not suitable for sharing, such as the address and port of a test server, unique insights accumulated during personal debugging, or temporary work notes.

For example, if you know that the development environment's database will undergo maintenance at specific times, such information only needs to be recorded at the local layer, without notifying the entire team, thus avoiding information noise.

```
<!-- ./CLAUDE.local.md (automatically added to .gitignore) -->

# Local Development Notes

- Database connection: postgresql://dev:dev123@10.0.1.50:5432/orders_dev
- Maintenance window: Database maintenance every Wednesday from 22:00 to 23:00;
  strictly forbidden to perform migration operations during this period
- Troubleshooting: If a CORS error occurs, first check the Nginx configuration (/
  usr/local/etc/nginx/nginx.conf)
- Integration configuration: The frontend service runs on port 5173; manual proxy
  forwarding configuration is required
```

At this point, Xiaobing keenly raised a key question: "What if there are conflicts among these five layers? For example, if I set '4-space indentation' at the user layer, but the project layer specifies '2-space indentation'?"

"Good question," Kage replied approvingly. "These files follow the principle of 'accumulation, not overwriting.' Claude Code will read the contents of all layers and merge them into a complete context. When conflicts arise, the Claude model will make decisions based on its judgment, with the core logic being that specificity takes precedence. In the indentation example, the project layer's '2-space' rule is more specific to the current project and thus takes precedence over the user layer's general '4-space'

preference. However, the enterprise policy layer is an exception. It has absolute authority—no matter how the lower layers are configured, its constraints cannot be bypassed. It's like the company's legal or security red lines: no project manager or personal preference can override them."

It is worth mentioning that `CLAUDE.md` also has a powerful **file referencing mechanism**. In any memory file at any layer, you can use the `@path/to/file` syntax to include the contents of other files, enabling modular organization of configurations.

For example, writing `@docs/api-conventions.md` in the project layer's `CLAUDE.md` will cause Claude to automatically load and parse the contents of the API conventions document. This mechanism supports up to five levels of nested recursion, which is sufficient for the organizational needs of most complex projects.

This provides an elegant documentation governance solution for large projects: `CLAUDE.md` no longer needs to bear all the details itself, but instead serves as a "master index," flexibly pointing to various specialized documents. This design successfully avoids the drawbacks of piling massive amounts of information into a single giant file, making the configuration structure clearer and more maintainable.

CLAUDE.md writing paradigms

Having understood the storage locations and hierarchical responsibilities of memory files, a more fundamental question emerges: how should they be written?

Kage solemnly wrote three words on the whiteboard: **Less is more**.

"The content of CLAUDE.md has global persistence," Kage explained. "It is fully injected into the context window at the start of every conversation. This means that every line you write consumes valuable token quota in every interaction."

He gave a vivid analogy: *"You can think of the context window as a desk with limited space. The surface area is fixed. If you pile up thick reference materials (i.e., lengthy rule files) on the desk before Claude Code starts actual work, there will be little space left for the code, files, and temporary drafts that really need to be processed. Once the desk is full, Claude Code will have to 'throw away' some early key information to make room, which often leads to it 'forgetting' important instructions or contextual details."*

According to Anthropic's official guidance, users should target under 200 lines per `CLAUDE.md` file; longer files consume more context and reduce adherence. The underlying logic is: instruction density is inversely proportional to compliance probability. The more verbose and complex the instructions, the less likely each rule will receive "deep attention" and "strict execution" by the model.

This aligns perfectly with the "**Information Overload**" principle in management: when an employee handbook is so thick that no one can read it through, its actual effectiveness is often inferior to a one-page A4 summary distilled to the core essence. In AI collaboration, conciseness is not just a way to save tokens—it is the key strategy to ensure instructions are effectively implemented.

So, what content is worth writing into `CLAUDE.md`? Kage provided an efficient "three-question framework" as a selection criterion.

First question: WHY (why do this?)

Core value: Endow the Claude model with the ability to "infer by analogy."

This type of information aims to reveal the underlying logic behind decisions, helping Claude understand the project's core concerns. Once it grasps the "why," even when encountering edge cases not explicitly specified, the model can make correct inferences based on principles.

Example: *"We chose Fastify over Express because Fastify's schema-based validation aligns with the project's 'type safety first' strategy."*

Effect: When Claude understands that "type safety" is the top priority, even if you don't specify a library somewhere, it will proactively prefer solutions with better type definitions, rather than mechanically matching keywords.

Second question: WHAT (what to do, what not to do)

Core value: Define the "behavior boundaries" that must not be crossed.

This is the most direct and rigid behavioral instruction, clearly delineating what is allowed and what is prohibited. Such rules usually do not require explanation and demand absolute compliance and execution. The vast majority of project specifications fall into this category.

Example: *"You must use pnpm as the package manager; npm or yarn are prohibited."*

Effect: Establishes a clear "red line," eliminating uncertainty in tool selection and ensuring consistency in the technology stack.

Third question: HOW (how to do it step by step)

Core value: Standardize the "Standard Operating Procedure."

This type of content applies to common commands, fixed processes, or specific operational paradigms. The model does not need to understand the complex principles behind the commands; it only needs to accurately execute the established steps.

Example: *"To perform a database migration, run `pnpm db:migrate`; to seed test data, run `pnpm db:seed`."*

Effect: Transforms the team's best practices into the model's muscle memory, reducing low-level mistakes caused by command typos or missing parameters and improving execution efficiency.

Not all information needs to answer all three questions at once. Most rules only need to be clearly defined at the WHAT level in a single sentence. Only those key decisions that are easily misunderstood or where the model is prone to "well-intentioned mistakes" are worth supplementing with in-depth explanations at the WHY level.

If you find that every rule in your `CLAUDE.md` is followed by a lengthy background explanation or emotional expression, it usually means the content is overloaded. Conciseness is the key to efficient collaboration.

Next, let's quickly build intuition by comparing a negative example with a subsequent positive example.

First, a negative example: vague "correct-sounding nonsense."

Project Description

```
This is a backend service for an e-commerce platform. Our team has 5 members, and
the project started development in March 2024.
It is written in Node.js and is a RESTful API service.
The code should be written well, with attention to performance.
Testing should be comprehensive.
Please follow best practices.
```

This `CLAUDE.md` is full of polite phrases and vague instructions common in human social contexts, which are almost equivalent to useless information for AI.

- **Written well:** To what extent? Does it mean conforming to Prettier formatting, or using design patterns?
- **"Comprehensive":** Which scenarios should be covered? Is it unit testing, integration testing, or E2E testing? Is the coverage requirement 80% or 100%?
- **"Best practices":** Whose best practices?

After reading this, Claude gains almost no actionable information.

The most fatal issue is the instruction "follow best practices." It appears correct, but is actually empty, because Claude will by default try to follow what it perceives as "best practices" (based on statistical patterns from massive internet data). The core problem is that the model's general "best practices" often differ from your team's specific "technical conventions."

Additionally, metadata like "the team has 5 people" or "development started in March 2024" provides no guidance for code generation logic, yet takes up valuable context space.

Now, let's look at a positive example: a precise "operation manual."

```
# Order Service API

## Technology Stack
- Node.js 20 + TypeScript 5.3 (strict mode)
- Fastify 4 framework (do not use Express)
- Prisma ORM + PostgreSQL 15
- pnpm 8 package management (do not use npm/yarn)

## Project Structure
- src/routes/ - Route definitions, only for parameter parsing and response
  construction
- src/services/ - Business logic layer, all core logic resides here
- src/repositories/ - Data access layer, encapsulates Prisma calls
- src/schemas/ - Zod validation schemas, one-to-one correspondence with routes

## Key Conventions
- Unified API response format: { success: boolean, data?: T, error?: { code:
  string, message: string } }
- Error codes use UPPER_SNAKE_CASE, e.g., ORDER_NOT_FOUND
- Database table names use snake_case plural form, primary key is UUID, must
  include created_at and updated_at

## Common Commands
pnpm dev          # Start development server, port 3000
pnpm test         # Run all tests (vitest)
pnpm build        # TypeScript compilation + type checking
pnpm db:migrate   # Run Prisma database migration
```

The difference between the two documents is obvious at a glance. The excellent version is concise, specific, and highly actionable—every rule directly anchors Claude's behavior logic.

- Technology stack lock-in: Seeing "Fastify 4 framework (do not use Express)," Claude will immediately exclude Express-related solutions and generate code based on Fastify

- Architecture standardization: After reading the directory structure, Claude can accurately determine that new route files should be placed in the `src/routes/` directory, rather than scattered arbitrarily
- Zero-error execution: With common commands and their annotations, Claude can accurately execute build and test tasks, eliminating the guesswork between `npm run build` and `yarn build`

In addition to manual editing, Claude Code also provides two convenient tools to help manage memory files.

The first is the `/init` command. When executed in the project root directory, Claude Code will automatically scan the project's directory structure, dependency configuration files (such as `package.json`, `pyproject.toml`, `Cargo.toml`, etc.), and other key configurations, then generate a draft `CLAUDE.md`. This draft usually covers technology stack identification, directory structure overview, and common script commands, making it an excellent starting point.

However, manual review and supplementation are still required for content that `/init` cannot automatically infer, such as specific constraints of layered architecture, unified API response formats, and team-specific naming conventions.

This "implicit knowledge" exists only in the minds of team members and cannot be extracted solely from configuration files. It must be made explicit through human intervention to ensure the AI's behavior fully aligns with team expectations.

The second is the `/memory` command, used to dynamically update memory during conversations. After executing this command, Claude Code will pop up a **file selector** listing all currently available memory files for you to choose and edit. These files cover different scopes of application, as shown in *Table 2.1*.

Scope of effect	Project memory	<code>./CLAUDE.md</code> or <code>./claude/CLAUDE.md</code>
Team sharing: can be version-controlled, suitable for unifying team standards	User memory	Personal global: personal preference configuration, effective for all projects of the current user

Project local memory	<code>~/ .claude/CLAUDE.md</code>	Personal local: only for the current project's personal preferences, usually automatically excluded by <code>.gitignore</code> to avoid being committed to the code repository
System file	<code>./CLAUDE.local.md</code>	Content carried

Table 2.1: Scope of Project Memory

After selecting a file, the system will open it directly in the **editor** for immediate modification.

Suppose you discover during interaction that Claude Code has again used `moment.js`, while the project specification clearly requires `date-fns`. At this point, you don't need to exit the conversation or manually search for the file; just follow these steps.

1. Enter the `/memory` command
2. Select the project-level `CLAUDE.md`
3. Add the rule: Always use `date-fns` for date handling; `moment.js` is prohibited
4. **Save and close**

The next time a conversation starts, this new rule will take effect immediately. If this preference is only a personal habit and you don't want it to affect the team, you can choose to write it into `CLAUDE.local.md`—this file is excluded by default via `.gitignore` and will not be committed to the code repository.

This mechanism creates a positive feedback loop of "mistake → correction → memory update → mistake avoided." Over time, your memory files will become increasingly precise, and Claude Code's performance in the project will continue to improve.

Conditional Rule System

As the project evolves, a single `CLAUDE.md` file will eventually face capacity bottlenecks.

When a project covers all rules for frontend component specifications, backend API design, database migration processes, testing strategies (coverage/mock), and infrastructure (Docker and CI/CD), piling all content into one file will lead to two core problems.

- **Maintenance nightmare:** The file becomes extremely lengthy, making it difficult to find and update specific rules.
- **Attention dilution:** For example, when Claude is writing a simple frontend component, its context is forced to load irrelevant information such as database

migration specifications and CI configuration requirements. This noise dilutes the model's attention, reducing its efficiency in handling the current task, and may even cause hallucinations or lead to key frontend constraints being ignored.

This is precisely the core value of the `.claude/rules/` directory. It evolves the memory system from "a single, heavy, all-encompassing manual" into "a modular knowledge base that can be accessed as needed."

This mechanism has two core advantages.

- **Domain segmentation:** Allows the large set of rules to be broken down into multiple topic-specific files, with each file focusing on a particular domain, greatly improving maintainability.
- **Intelligent activation:** By adding YAML Frontmatter at the beginning of each file and using the `paths` field to declare Glob patterns, the corresponding rules are only activated and injected into the context when the file path operated on by Claude Code matches the pattern. In scenarios where there is no match, these rule files remain quietly stored on disk, consuming no tokens or context space.

Let's look at an example. If you need to standardize the writing of test files, you can create the following rule file.

```
<!-- .claude/rules/testing.md -->
---
paths:
- "**/*.test.ts"
- "**/*.spec.ts"
- "tests/**"
---

# Testing Guidelines

- Use vitest as the testing framework; jest is prohibited.
- Every test file must contain a describe block, and its name must match the module under test.
- Module mocking must be done uniformly with vi.mock(); manual mocking is strictly forbidden.
- All asynchronous tests must use async/await syntax; the use of done callbacks is prohibited.
- Test data must be generated through factory functions; hardcoding data in tests is strictly forbidden.
```

With this configuration, when Claude edits `src/services/order-service.ts`, these testing guidelines will not be loaded; however, as soon as `tests/order-service.test.ts` is written, these rules are automatically enforced. This "on-demand loading" design philosophy is closely aligned with the concept of lazy loading in programming languages: there is no need to preload all potential resources at startup—only dynamically invoke them when truly needed.

The same approach can be extended to other areas of development. For example, database guidelines are loaded only when working with Prisma files or the data access layer, while API design guidelines are enforced only when editing route or schema files.

```
<!-- .claude/rules/database.md -->
---
paths:
- "prisma/**"
- "src/repositories/**"
---

# Database Guidelines

- Once a migration file is committed to the main branch, it is strictly forbidden
to modify it; only create new migrations as needed.
- All queries must go through the repository layer; the service layer is
prohibited from directly invoking the Prisma client.
- Batch operations must be wrapped in a transaction. When writing more than 100
records at a time, batch processing is mandatory.
<!-- .claude/rules/api-design.md -->
---
paths:
- "src/routes/**"
- "src/schemas/**"
---

# API Design Guidelines

- Every route must be configured with a corresponding Zod schema for input
parameter validation.
- List endpoints must uniformly support pagination parameters: page (starting from
1) and limit (default 20, maximum 100).
- Error response structures must include a machine-readable code field and a
human-readable message field.
```

Upon hearing this, Xiaoxue succinctly summarized: *"This is like tailoring dedicated operation manuals for employees in different positions within a company—receptionists follow the front desk procedures, warehouse staff adhere to warehouse protocols, and no one needs to memorize the entire company's regulations. Only when you step into warehouse operations do you need to open the 'Warehouse Management Manual.'"*

Kage nodded and added: *"This modular approach to organizing rules has another major practical benefit: in collaborative scenarios, domain experts can independently maintain their own rule files, greatly reducing the likelihood of Git merge conflicts. Frontend engineers are responsible for frontend.md, backend engineers maintain api-design.md, and DBAs oversee database.md—each focuses on their own area without interference. As the team grows, the value of this 'divide and conquer' strategy becomes increasingly apparent."*

Note

Kage's Notes

If a rule file does not have the paths front-matter metadata configured, the system will treat it as a global, unconditional rule, forcibly loading it in every session—effectively the same as writing it directly into the main CLAUDE.md file. Therefore, be sure to set precise paths restrictions for each rule file to truly leverage the memory system's core advantage of "on-demand loading."

In practice: three typical project configurations

With the theory explained, let's integrate it through practical CLAUDE.md examples for three typical projects. Pay close attention to their distinct areas of emphasis.

- **Frontend project:** Focused on component conventions and state management strategies
- **Backend project:** Centered on layered architecture design and API conventions
- **Data project:** Emphasizes rigor in data processing workflows and reproducibility standards for experiments

These differences are by no means accidental; they precisely reflect the distribution of "most frequent error points" in different technical domains. An excellent CLAUDE.md should act like a precision-guided weapon, concentrating its firepower on those key areas most prone to issues.

React frontend project configuration

In **React** frontend projects, component granularity, state ownership, and styling scheme selection are often hotbeds of disagreement, and also where Claude is most likely to "take

initiative" and cause fragmented code styles. If the state management strategy is not explicitly specified, Claude is highly likely to mix Redux, Context API, and local state within the same project, resulting in architectural chaos.

Below is a practical CLAUDE.md example for a frontend project of an e-commerce platform.

```
# E-commerce Platform Frontend Project Guidelines

## Tech Stack
- Core: React 18 + TypeScript (strict mode)
- Build/Package Management: Vite 5 + pnpm
- State Management:
  - Server State: TanStack Query (React Query)
  - Global Client State: Zustand
- Styling Solution: Tailwind CSS (strictly prohibit CSS Modules or styled-components)

## Component Guidelines
- Paradigm: Use only function components + Hooks; class components are prohibited.
- Naming:
  - Files use PascalCase (e.g., ProductCard.tsx)
  - Props types are named as ComponentName + Props (e.g., ProductCardProps)
  - Directory Structure:
    - Common UI atomic components → src/components/ui/
    - Business domain components → src/components/features/

## State Management Decision Tree
- Server data (API responses, caching, synchronization) → TanStack Query
- Global client state (theme switching, user login status, temporary cart state) → Zustand
- Component local state (form input values, dropdown expand/collapse, modal display) → useState

## Common Commands

```bash

pnpm dev # Start the development server on port 5173

pnpm build # Build for production
```

```
pnpm test # Run vitest unit tests

pnpm lint # Run ESLint code checks

...

```

Please note the section on the "State Management Decision Tree." It is not merely a list of technology stack names, but rather constructs a "mapping logic from scenarios to tools." This kind of "decision guidance" is far more effective than a simple list of tools—it directly tells Claude "which key to choose" when faced with a specific problem, thereby fundamentally preventing architectural style drift from the outset.

## Node.js backend project configuration

The lifeblood of a **Node.js** backend project lies in the clear boundaries of responsibilities within its layered architecture and the unification of data formats. The "correctness" of backend code often depends on whether each layer strictly adheres to dependency rules—if the route layer directly calls Prisma, bypassing the service and repository layers, business logic will leak into the infrastructure layer, leading to maintenance quagmires down the line.

Below is a practical example of a CLAUDE .md for an order microservice.

```
Order Microservice Backend Specification

Technology Stack
- Runtime: Node.js 20 + TypeScript 5.3
- Framework: Fastify 4 (Express is strictly prohibited)
- Database: Prisma ORM + PostgreSQL 15
- Toolchain: pnpm (package management), Zod (data validation)

Layered Architecture (Strict Unidirectional Dependencies)
- routes/ → Parse request parameters, validate input, call controllers, construct HTTP responses
- controllers/ → Orchestrate service calls, handle HTTP semantics, capture and transform exceptions
- services/ → Implement core business rules, manage transactions, coordinate multiple repositories
- repositories/ → Encapsulate Prisma queries, data mapping (other layers are strictly forbidden from calling Prisma directly)

```

### ## API Response Standards

All endpoints must return a unified format: { success: boolean, data?: T, error?: { code: string, message: string } }

Pagination specification: page (starting from 1), limit (default 20, maximum 100)

Error codes: Must use UPPER\_SNAKE\_CASE format

### ## Common Commands

```
```bash
```

```
pnpm dev          # Start the development server on port 3000
```

```
pnpm test         # Run the vitest test suite
```

```
pnpm build        # Compile with tsc
```

```
pnpm db:migrate   # Execute Prisma database migration
```

```
pnpm db:studio    # Launch the Prisma Studio visual interface
```

```
```
```

In this configuration, the most critical rule is "strictly prohibit direct calls to Prisma from other layers." This is not merely a matter of coding style preference, but an inviolable architectural principle that directly determines the maintainability and longevity of the project.

## Python data project configuration

**Python** data projects present a landscape that is fundamentally different from web development. If Claude is unfamiliar with engineering conventions in the data science domain, it is easy to generate code that violates best practices—for example, overusing `inplace=True` to modify DataFrames in place, which undermines data flow traceability, or mixing `None` and `pd.NA`, leading to confusion in missing value handling logic.

Below is a practical example of a CLAUDE.md specification for a user behavior analysis system.

```
User Behavior Analysis System Specification

Tech Stack
- Environment: Python 3.11 + uv (for dependency and virtual environment
management)
- Core Libraries:
- Data Processing: pandas 2.x
- Machine Learning: scikit-learn
- Visualization: matplotlib + seaborn
- Code Standards:
- Type Annotations: Strict use of the typing module
- Docstrings: Mandatory use of Google Style

Project Structure
- notebooks/: Exploratory data analysis (Jupyter), naming format "number-
description.ipynb", e.g., 01-data-exploration.ipynb
- src/data/: Data loading, cleaning, and pipeline construction
- src/features/: Feature engineering logic
- src/models/: Model definitions, training workflows, and evaluation metrics
- tests/: pytest unit test suite

Data Processing Conventions
- Missing Values: Use pd.NA exclusively; do not use None or bare np.nan
- Date Columns: Always convert to datetime64 type; output/storage format must be
YYYY-MM-DD
- Memory Optimization: Low-cardinality categorical variables must be explicitly
converted to category type
- DataFrame: Do not use inplace=True; all transformation operations must return a
new DataFrame copy

Common Commands

```bash

uv sync                                # Install/synchronize dependency environment

uv run pytest                          # Run the test suite
```

```
uv run jupyter lab          # Launch Jupyter Lab

uv run python -m src.train  # Execute the model training script

... 
```

Kage pointed to the three configuration files and delivered his final summary and comments:

"Did you notice? Each of these three files is kept to just twenty or thirty lines, but every line is a gem—each one is pure substance. They discard the general knowledge that Claude already knows well—you don't need to explain what React is, nor do you need to reiterate Python's indentation rules—instead, they focus precisely on the specific domain and the unique conventions and decisions of this project."

This is the core rule for writing `CLAUDE.md`: only record the key information that Claude must know but cannot infer on its own. Always remember this golden test: if deleting a rule would still allow Claude to behave correctly, then that rule does not belong in `CLAUDE.md`.

Summary

Returning to the scenario at the beginning of this chapter: every time Xiaobing starts a new session, Claude habitually uses Express instead of Fastify, and npm instead of pnpm. Now we understand this is not a limitation of the Claude model's capabilities, but rather because it lacks a crucial "onboarding manual."

`CLAUDE.md` is the key mechanism to solve this pain point. It solidifies the team's technology choices, coding standards, project structure, and common commands into a machine-readable configuration document. Whenever a session starts, Claude Code automatically loads this information, completely bypassing the need for repetitive corrections and jumping straight into efficient collaboration.

The design of the five-layer memory architecture deeply embodies the classic software engineering principle of "separation of concerns."

- **Enterprise Strategy Layer:** Strictly upholds security baselines, managed uniformly by the organization
- **User Layer:** Captures personal preferences, respecting individual differences
- **Project Layer:** Carries team consensus, ensuring consistent collaboration
- **Rule Directory Layer:** Enables modular and conditional loading of rules
- **Local Layer:** Holds private notes, providing flexible temporary context

Each layer has clear boundaries of responsibility and management, powerfully supporting team collaboration while fully respecting personal habits.

Among them, the conditional rule system is a particularly ingenious engineering design. It uses Glob patterns to precisely define the scope of rules, achieving an intelligent mechanism of "delivering the right information to the context at exactly the right moment." This design completely abandons the crude approach of dumping all rules in front of Claude, ensuring a clean and efficient context.

As for the writing paradigm, "less is more" is far from an empty slogan—it is backed by serious technical reasoning.

The context window is the most precious strategic resource for large models; every token carries a high opportunity cost.

Forcibly stuffing irrelevant information into `CLAUDE.md` is like cramming non-passengers into a subway car during rush hour—not only does it take up valuable space, but it also squeezes out the truly "key information" that needs to get on board, preventing it from entering the model's field of view.

An excellent `CLAUDE.md` should be like a concise battle memo, strictly following these three iron rules.

- **Necessity:** Every entry is information that Claude must know and cannot infer on its own
- **Actionability:** Every entry is a specific, clear, and executable instruction
- **Fragility test:** If any entry is deleted, it would directly cause Claude to make mistakes or deviate from expectations

In *Chapter 3*, we will formally step into the vast world of Skills.

If `CLAUDE.md` is the "onboarding manual" that gives Claude its foundational understanding, then Skills is the "professional training material" tailored for it.

When your "onboarding manual" becomes increasingly bloated, trying to cram all domain knowledge into it, not only does it waste context resources, but it also reduces the efficiency of core rule responses. At this point, you need a more elegant and intelligent on-demand loading mechanism—this is the core value of the Skills system: to inject professional knowledge into the conversational context precisely when it is truly needed.

Reflection questions

1. Please identify the core information that Claude must know in your current project. Try breaking it down using the "Three Questions Framework" (WHY, WHAT, HOW), and

draft a `CLAUDE.md` of no more than 30 lines. After drafting, review each line: if you delete this line, what mistake would Claude make? If you cannot answer this question, that line may be unnecessary.

2. In team collaboration, if developers disagree on rules in `CLAUDE.md` (such as whether to use 2 or 4 spaces for indentation), how can you use the "five-layer memory architecture" to resolve it? Please specify: which rules must be placed at the project layer to enforce unified standards, and which preferences are suitable for the user or local layer to preserve individual freedom of choice?
3. Run the `/init` command in your project to automatically generate `CLAUDE.md`, then compare it in depth with your manually written version: what explicit information can the machine accurately infer? And what kind of "implicit knowledge" must rely on manual supplementation?

3

Teaching a Man to Fish: Engineering Reusable Skills

We are what we repeatedly do. Excellence, then, is not an act, but a habit." — Will Durant

On Tuesday morning, Xiaobing started a brand new Claude Code session, preparing to tackle the first task of the week: generating API documentation for the payment module.

She first spent three minutes clarifying the context: the team follows RESTful conventions, documentation must use the OpenAPI 3.0 format, error codes should align with internal company standards, sample data must be real and valid, and the output must include both Chinese and English. Only after feeding the model with last week's reference documentation written by a colleague did she formally instruct it to begin work.

It was the same yesterday, the same last Friday, and the Friday before that.

Every time a new conversation starts, all this prerequisite knowledge must be "fed" to Claude again. Even more frustrating, even within the same session, when the task switches from "generating API documentation" to "reviewing code quality," Claude cannot automatically adapt to the new work mode—she has to spend another round of conversation to re-explain review priorities, output formats, and severity classifications. This is not using an intelligent assistant; it's more like giving onboarding training to a temp worker with "amnesia" day after day.

Xiaoxue feels the same way. When collaborating with the company's data analysis team, every time she asks Claude to perform financial analysis, she has to re-explain the gross margin

formula, industry benchmark data, and report template format. This core knowledge never changes, but Claude always acts as if it's hearing it for the first time.

"Is there a way," she asked Kage, "to make Claude permanently remember certain domain-specific work paradigms? Not just project-level general memory like `CLAUDE.md`, but more precise contextual awareness—for example, as soon as I mention 'API documentation,' it automatically knows the execution standards; as soon as I mention 'financial analysis,' it automatically loads the relevant industry benchmarks and report templates."

Kage nodded slightly and turned to write a word on the whiteboard: **Skills**.

From `CLAUDE.md` to Skills: Two Dimensions of Knowledge

"Before we dive into Skills," Kage began, "I want to clarify a fundamental question: why is `CLAUDE.md` no longer sufficient?"

With that, he drew a coordinate system of knowledge types and loading strategies on the whiteboard (see Figure 3.1).

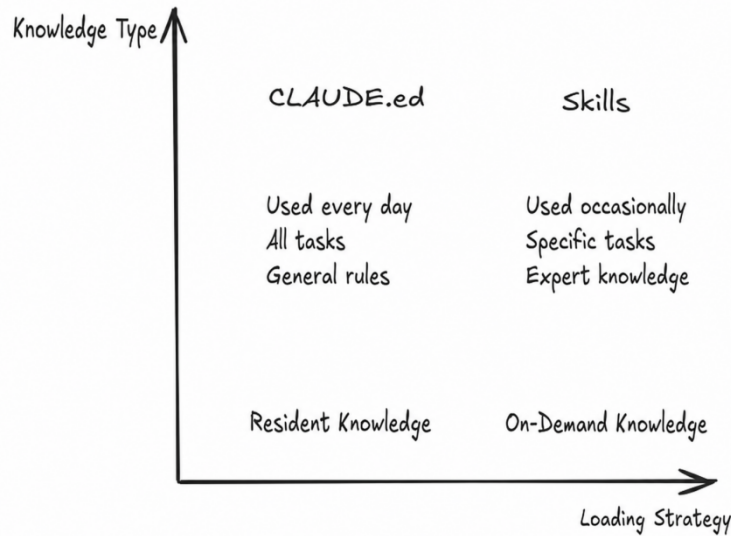


Figure 3.1: Knowledge Types and Loading Strategies: `CLAUDE.md` vs. Skills

"`CLAUDE.md` is like a company's 'culture handbook,'" Kage explained. "It defines the basic rules of the project: whether to use TypeScript or Python, whether to indent with tabs or spaces, and what format PR titles should follow. These rules must be in effect at every moment of every conversation, so they

must always reside in the context and be loaded in full each time. But the cost is obvious: 'always resident' means that whether or not you're working on related tasks, these tokens are constantly being consumed."

He paused and looked at Xiaobing: *"Think again about your API documentation knowledge. OpenAPI 3.0 standards, error code conventions, bilingual formats. This information is only needed when you're generating API documentation. If you force it into CLAUDE.md, then when you're debugging bugs, running tests, or writing configuration files, these thousands of tokens of documentation standards still occupy the context, wasting valuable space and diluting the weight of truly key information."*

"So, is this the core pain point that Skills aims to solve?" Xiaobing asked, beginning to understand.

*"Precisely. Skills solves the problem of **on-demand knowledge delivery**."* As Kage spoke, he drew a complete comparison table on the whiteboard (see Table 3.1).

Dimension	CLAUDE.md	Skills
Content carried	Project general standards	Professional workflows, domain knowledge
Scope of effect	Current project	Cross-project, cross-session
Trigger method	Always effective	Activated as needed
Loading strategy	Full load each time	Progressive on-demand loading
Typical use	Use TypeScript	"How to conduct code review"
Relationship with Agent	Shared by all agents	Can be bound to a specific Agent
Token consumption	Fixed overhead	Pay as needed when used
Enterprise ontology	Corporate culture manual	SOP (standard operating procedure) operation manual

Table 3.1: Comparison of CLAUDE.md and Skills across all dimensions

The last row of Table 3.1, **Enterprise ontology**, caught Xiaoxue's attention. "What does this mean?" she asked.

Kage explained, "A mature enterprise actually possesses two fundamentally different types of knowledge assets. The first type is the set of universal rules that everyone must follow—such as attendance policies, safety red lines, and communication protocols. These are like the slogans posted on the office walls: visible to all, effective at all times. This corresponds to CLAUDE.md. The second type is the **standard operating procedures (SOPs)** for specific roles, for example, how finance closes the books each month, how operations handles incidents, or how customer service responds to complaints. These are not posted on the wall for everyone to see, but are neatly stored in departmental filing cabinets, only consulted by those in specific roles when they need to perform specific tasks. That is Skills. CLAUDE.md is the wall, while Skills is the filing cabinet. You don't need to carry the finance manual when writing code, nor do you need the safety protocols when preparing reports. Only at the moment of 'need' is the corresponding knowledge precisely retrieved (see Figure 3.2)."



Figure 3.2: Skills can be accessed on demand

This analogy precisely anchors the core position of Skills within the Claude Code architecture. Claude Code's definition of a Skill is both straightforward and profound: a Skill is a folder containing instructions, packaged into a simple directory structure, used to 'teach' Claude how to handle specific tasks or workflows. Within this definition, there are two key terms worth exploring.

The first is "folder," from "string" to "engineering." A Skill is far more than just a prompt or a simple configuration item; it is a fully engineered directory that can house complex codebases, detailed domain documentation, diverse templates, and even executable scripts.

The second is "teach," from "constrain" to "internalize." The definition uses "teach," not "command" or "constrain." This subtly reveals a fundamental shift in its operational mechanism. "Teach" is about imparting and internalizing capability. Through Skills, Claude truly understands the operational logic of a domain. It is no longer a machine that passively executes instructions, but becomes a "skilled practitioner" in that field.

Dissecting a Skill: Skeleton and Texture

"There is an old Chinese story about a master butcher named Pao Ding, who could carve an ox effortlessly not through brute force but because he had completely internalized its natural structure — where the joints were, where the hollows ran," Kage quoted. "Understanding the structure of a Skill requires the same wisdom: first, see its 'skeleton', understand how knowledge is placed and organized, then you can talk about how to use it skillfully."

Directory Structure

He turned to the whiteboard next to the coordinate system and listed a typical Skill directory structure.

"Think of this as how the blade itself is built," Kage said, pointing to the directory structure he had just drawn on the whiteboard. "Like the master butcher Pao Ding, who guided his knife through the ox's natural gaps and hollows rather than forcing it, the skill lies in reading the structure — not in the knife." He then broke the directory down one piece at a time.

```
api-doc-generator/          ← Skill name (kebab-case, separated by
hyphens, clear like a filename)
├─ SKILL.md                 ← [Core skeleton] Required main skill file (note:
case must match exactly)
├─ scripts/                 ← [Hands and feet] Optional executable scripts,
enabling the Skill to actively "do" things
│   └─ detect-routes.py     ← For example, automatically scanning the codebase
to discover routes
└─ reference/               ← [Memory bank] Optional reference documents
```

```

loaded on demand, not occupying resident memory
|   |─ openapi-patterns.md    ← Standards read only when generating documentation
|   |─ error-codes.md        ← Code tables consulted only when handling errors
|   └─ templates/            ← [Molds] Optional output templates to ensure
consistent output format
|   └─ endpoint-doc.md        ← Defines what the API documentation should look like

```

"These seemingly tedious 'rules and regulations' are, in fact, error-prevention mechanisms forged through hard-earned engineering experience." Kage picked up a red marker and heavily circled several key points next to the directory structure on the whiteboard, his expression turning serious.

Filename: `SKILL.md`: the sole "trigger." The filename must be in all uppercase; `skill.md`, `Skill.md`, `SKILL.MD` are all invalid. Claude's loader performs an exact string match when scanning files, not a fuzzy search.

Directory name: kebab-case: the "universal language" across platforms. Directory names use all lowercase letters and hyphens (e.g., `api-doc-generator`), up to 64 characters, with the character set limited to `[a-z0-9-]`. No spaces, underscores, uppercase letters, leading or trailing hyphens, or consecutive hyphens are allowed. This is to smooth out differences between operating systems. Windows is case-insensitive, while Linux is case-sensitive. Kebab-case is the "Esperanto" of the internet age, ensuring your Skill can be losslessly recognized in any environment or toolchain.

Keep the Skill directory a pure instruction space. Anthropic recommends against placing a `README.md` inside a Skill directory, and human-facing documentation is better kept in the parent directory. The reason is hygiene rather than a hard rule. When a Skill activates, only the body of `SKILL.md` enters the context window; supporting files — including any `README.md` — are loaded only when `SKILL.md` explicitly points to them, so an unreferenced `README.md` costs no context at all. Keeping the folder free of human-facing documents simply removes the temptation to reference them and keeps the instruction surface clean.

Naming neutrality: avoid "claude" and "anthropic." Treat brand-free naming as a convention rather than an enforced validation rule. What Claude Code actually enforces on a Skill's identifier is that it be lowercase letters, digits, and hyphens, within the length limit, and that it match the folder name. Skills are a universal knowledge encapsulation format, and their vitality lies in remaining portable across harnesses — which is why brand-neutral names are worth adopting even where no validator demands it.

YAML Frontmatter: The "ID Card" of a Skill

The YAML Frontmatter at the top of the `SKILL.md` file defines the Skill's identity and behavioral boundaries. The complete field descriptions are as follows:

```
---
name: api-doc-generator
# [Unique Identifier] Up to 64 characters
# If omitted, the directory name is used by default

description: >
Generate API documentation from source code. Use when
the user asks to "write API docs", "document endpoints", or
"create OpenAPI specs".
Supports Express, FastAPI, and Spring Boot.
# [Trigger Description] Up to 1024 characters
# If omitted, the first paragraph of the SKILL.md body is used by default

argument-hint: "[source directory] [output format]"
# [Argument Hint] Displayed in the "/" menu
# Helps users understand the input format accepted by this Skill

disable-model-invocation: true
# [Disable Model Auto-Invocation] When set to true, prohibits Claude from auto-
invoking
# The user must manually trigger via "/skill-name"
# Default: false

user-invocable: false
# [User Invocability] When set to false, hides from the "/" menu
# But Claude can still auto-invoke
# Default: true

allowed-tools:
- Read
- Grep
- Glob
- Write
- Bash(python:*)
# [Tool Whitelist] Precisely control which
tools and permission scopes can be used during Skill execution
```

```

model: haiku
# [Designated Model] Specifies the model used by this Skill
# For simple tasks, haiku is recommended to improve response speed and reduce
costs

context: fork
# [Execution Context] When set to "fork", executes in an isolated sub-agent
# Ensures the main conversation context is not polluted

agent: Explore
# [Sub-Agent Type] Effective when context is set to "fork"
# Optional values: Explore, Plan, general-purpose, or custom Agent

hooks:
PreToolUse:
- matcher: Bash
hooks:
- command: echo "$TOOL_INPUT" >> audit.log
# [Lifecycle Hooks] Define event handling logic during Skill activation
# Only effective while the Skill is active
---
```

These fields can be divided into three logical groups, each corresponding to one of the three core dimensions of Skill design: trigger mechanism, permission control, and runtime environment.

- **Identity fields (Trigger Mechanism):** Includes name, description, and argument-hint. These define "what the Skill is," clearly communicating the Skill's functional positioning and invocation method to both Claude and the user. They form the foundation of the triggering logic.
- **Invocation and tool fields:** `disable-model-invocation`, `user-invocable`, `allowed-tools`, `disallowed-tools`, and `model`. Read these precisely, because the names promise more than they deliver. `disable-model-invocation: true` stops Claude from auto-invoking the Skill, leaving it user-triggered. `user-invocable: false` hides the Skill from the slash-command menu — a visibility switch, not a security boundary. `allowed-tools` grants and auto-approves tools while the Skill is active; it does not restrict them. The field that removes tools from Claude's reach is `disallowed-tools`. And `model` overrides the model for the remainder of the current

turn, which is a routing decision rather than a permission control. Real boundaries come from your permission settings and from Hooks (*Chapter 5*).

- **Execution fields (Runtime Environment):** Includes context, agent, and hooks. These determine "where it executes" and "what happens during execution," configuring the isolated environment, sub-agent type, and lifecycle hooks to ensure tasks run within the intended context.

Mastering these three dimensions means mastering the complete framework of Skill architecture design.

Note

Kage's notes

`disable-model-invocation` and `user-invocable` are easily confused, but in fact, they control two orthogonal invocation directions. `disable-model-invocation: true` means "Claude cannot automatically trigger this Skill; it must be manually executed by the user via `/skill-name`." This is suitable for operations with side effects or high risk, such as code commits (`/commit`), service deployments (`/deploy`), etc., to ensure explicit user confirmation. The default value is `false`. `user-invocable: false` means "this Skill will not appear in the `/` menu, and users cannot manually trigger it, but Claude can still invoke it automatically within internal logic." This is suitable for purely informational or auxiliary reference Skills; hiding them avoids menu clutter while retaining Claude's ability to auto-load and use them when appropriate. The default value is `true`. With the default configuration (both set to "allowed"), the Skill is open to both Claude and the user.

Progressive Disclosure: The Return on Investment of Knowledge

"Alright, I understand the structure of a Skill," Xiaobing asked, "but if I have 20 Skills, each containing thousands of tokens, does Claude Code have to load all of them for every conversation? How is that any different from stuffing everything into `CLAUDE.md`?"

This is a highly pertinent question, and it precisely reveals the most ingenious core of the Skills system: **Progressive Disclosure**.

The Library Model

Kage replied, "Here, you can use a library as an analogy. Imagine you walk into a library to look for information. You would never read all the books at once; instead, you follow an efficient retrieval process: first browse the catalog to locate the category, then extract the target book to consult its chapters, and finally, closely read only the sections you need. The Skills system adopts this same three-layer progressive disclosure model (see Figure 3.3), enabling on-demand access to vast knowledge at minimal token cost."

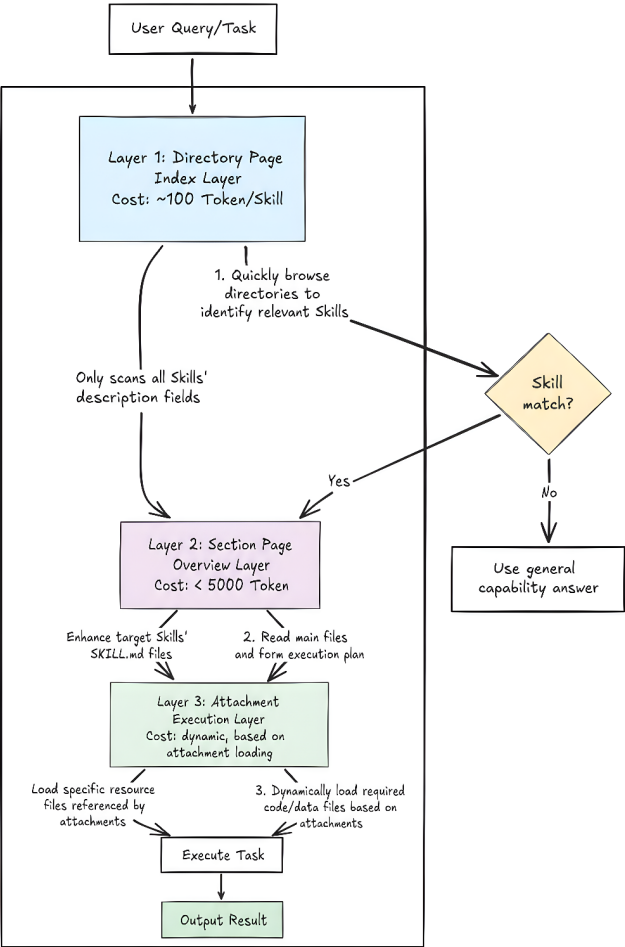


Figure 3.3: Three-layer Progressive Disclosure Model

The performance improvement brought by the three-layer model in practical applications is significant. Take a complete financial analysis Skill (totaling 5,300 tokens) as an example; its file composition is shown in *Table 3.2*.

File	Tokens	Description
SKILL.md	800	Main logic and routing guide
reference/revenue.md	1500	Income data reference
reference/costs.md	1200	Cost structure reference
reference/profitability.md	1000	Profit margin formula reference
templates/report.md	800	Report template
Total	5300	—

Table 3.2: File composition of a complete financial-analysis Skill

A comparison of token consumption for different loading methods is shown in *Table 3.3*.

Scenario	Traditional full load	Progressive disclosure	Savings ratio
Skill not activated	0	0	—
Scanning phase (relevance judgment only)	5300	About 100	98%
Simple request (main file only)	5300	800	85%
Medium request (needs one reference file)	5300	2300	57%
Complex request (needs all resources)	5300	5300	0%

Table 3.3: Token consumption comparison for different loading methods

The data shows that the vast majority of requests require only a portion of the resources. When a user asks, "How is gross margin calculated?" Claude Code only needs to load SKILL.md (for

routing) and `profitability.md` (to get the formula), totaling about 1,800 tokens. The other three files (about 3,500 tokens) remain entirely on disk, incurring zero cost. This is the core economic advantage of "on-demand knowledge delivery": achieving maximum knowledge coverage with minimal context cost.

The Budget Mechanism of description

The first level of progressive disclosure, that is, the description of all Skills, is injected into Claude's context in a resident manner. This means they must share a limited token budget, forming the system's "entry bottleneck."

Claude Code sizes this listing with a character budget that scales with the model's context window — roughly 1% by default. Independently of the budget, each entry's combined text is capped at 1,536 characters. Because these figures have drifted between releases, treat them as an order of magnitude and verify against the Claude Code version you ship on.

The budget is not divided evenly among installed Skills. Skill names are always listed. When the combined descriptions overflow the budget, Claude Code shortens and then drops descriptions starting with the Skills you invoke least, so the Skills you actually use keep their full text. The mechanism degrades by usage frequency, not by an equal per-Skill quota.

$$\text{Characters available per Skill} = \text{Total budget} / \text{Total number of Skills}$$

This means a Skill never disappears merely because you have installed many of them. What degrades under budget pressure is the description text of the Skills you use least, while the names of all Skills remain visible. Writing tight descriptions is good practice from day one — not because a quota is divided among you, but because a short description survives budget pressure intact.

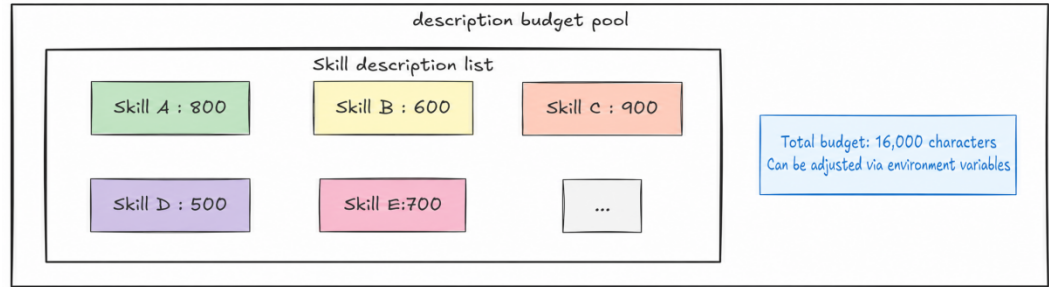


Figure 3.4: The description budget pool of Claude Code

A Skill is never silently excluded from the listing because its description is long. Its name is always present; only the description text is shortened or dropped when the budget is tight. A

Skill that Claude appears not to "see" is therefore a description-quality problem, not a hidden character cap.

Note

Kage's notes

Tip 3: To raise the budget, set the `skillListingBudgetFraction` setting (for example, `0.02` for 2%). The `SLASH_COMMAND_TOOL_CHAR_BUDGET` environment variable remains available as a fixed-character override.

This budget mechanism reveals a core engineering philosophy: do not blindly create Skills, but pursue an architecture that is "few but refined."

When you find yourself needing to install more than 20 Skills, it is usually a signal that you need to re-examine your architectural strategy.

- **Merge similar items:** Combine multiple fragmented Skills into a comprehensive Skill.
- **Hide internal tools:** Mark purely internal sub-tasks as `disable-model-invocation: true` to reduce budget usage.
- **Adopt a "token investment" return-on-investment mindset:** Treat every character in the description as a costly "token investment"—invest precisely for efficient retrieval and accurate execution; invest wastefully, and you not only waste your own budget quota but also crowd out the survival space of other Skills.

Trigger Mechanism: How Claude Code Decides Which Skill to Invoke

The trigger mechanism of a Skill is the most critical and essential aspect of the entire system to understand in depth. It directly determines whether a Skill can be activated at the right moment—this is even more important than the quality of the Skill's content itself. After all, if it cannot be triggered successfully, even the most brilliant content is rendered useless.

Dual-Channel Activation Mechanism

Each Skill supports two activation paths.

- **Explicit Invocation:** When a user enters `/skill-name` in the conversation, Claude Code immediately loads and executes the corresponding Skill. This method is similar to a terminal command—clear, direct, and unambiguous. If the Skill defines an

argument-hint, the user can also provide parameters, such as `/commit fix login bug` or `/review src/auth/login.ts`.

- Semantic Matching:** After deeply understanding the user's intent, Claude autonomously determines which Skill best fits the current task and automatically loads it. In daily use, this is the core value of Skills—the user only needs to describe their needs, and Claude will intelligently decide behind the scenes whether and how to invoke the most appropriate Skill. The entire process is completely transparent to the user, achieving a seamless "what you think is what you get" experience.

Figure 3.5 shows the execution flow of explicit invocation and semantic matching.

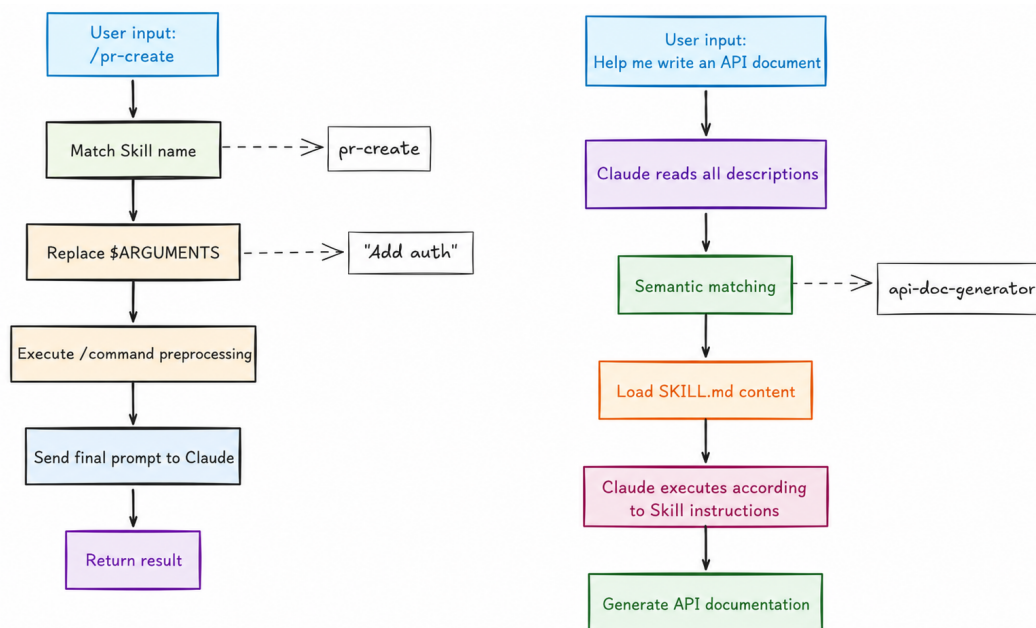


Figure 3.5: Execution flow of explicit invocation (left) and semantic matching (right)

Description: The Soul of Skills

The semantic matching mechanism relies entirely on the description field. This field is not explanatory text for human readers; rather, it is the **sole signal** Claude uses when deciding whether to invoke a particular Skill.

Claude Code recommends the following formula for structuring the description:

[Function Definition] (What it does) + [Trigger Scenarios] (When to use) + [Core Capabilities] (What it can do)

The best way to understand this formula is by comparing "ineffective" and "effective" examples. Below are examples of both approaches.

```
# Too vague, making it impossible for Claude to determine when to use
description: Helps with projects.

# Overly technical, lacking user-oriented trigger keywords
description: Implements the Project entity model with hierarchical relationships.

# Only describes functionality, without specifying trigger scenarios
description: Generates API documentation.

# Correct example: clear statement, covers specific trigger scenarios, details
core capabilities
description: Generate API documentation from Express, FastAPI, or Spring Boot
source code. Use when user asks to "write API docs", "document endpoints", "create
OpenAPI specs", or mentions "Swagger". Supports route detection, request/response
schema extraction, and authentication requirement marking.
```

The description field has a limit of **1,024 characters**. This is not a lot of space, so every word must be chosen carefully.

An efficient and practical writing strategy can be broken down into three steps:

1. **Define core capabilities (What):** Summarize in one sentence what the Skill can do, establishing its basic functional positioning
2. **Specify trigger scenarios (When):** Use the "Use when user..." sentence pattern to list in detail the various user commands, phrases, or keywords that could trigger this Skill, increasing the hit rate for semantic matching
3. **Define exclusions (Not For):** (Optional but recommended) If the Skill is prone to being triggered incorrectly, be sure to add a "Not for..." clause to clearly state scenarios where it should not be used, optimizing decision accuracy

Note

Kage's note

The primary audience for your description is Claude, not human readers. When humans read documentation, they tend to scan titles and browse structure to quickly grasp the gist; when Claude reads the description field, it is performing deep semantic matching. It needs to capture subtle nuances in user intent. To improve trigger accuracy, you must exhaustively include all possible user expressions in the

description. For example, for the need to "generate API documentation," users might say "generate API docs," "write interface documentation," "output OpenAPI specs," or "help me write a Swagger." All these different phrasings should be explicitly included in the description. The richer your synonym set, the broader the coverage of semantic matching, and the higher the trigger accuracy.

Preventing Over-Triggering and Under-Triggering

In practice, the skill-triggering mechanism often faces two typical failure modes. Understanding and fixing these issues is key to optimizing Claude's performance.

1. Under-triggering

- **Phenomenon:** The Skill should be invoked but is not.
- **Data warning:** In Vercel's Next.js 16 evaluations, the single Skill under test went uninvoked in 56% of cases when the agent was given no explicit guidance (Vercel Engineering, "AGENTS.md outperforms skills in our agent evals"). The result is specific to that eval and one Skill; read it as a caution about discovery, not as a universal probability.
- **Root cause:** The description is written in an overly technical or academic style, creating a significant semantic gap with the user's natural, conversational expressions.
- **Remediation strategy:** "Translate" user language. Add a large number of commonly used user expressions to the description, covering synonyms for domain terms, colloquial phrases, and even common but inaccurate expressions (for example, many users confuse "Swagger" and "OpenAPI"—both should be included).

2. Over-triggering

- **Phenomenon:** The Skill is incorrectly activated in scenarios where it should not be invoked.
- **Root cause:** The description is defined too broadly, containing too many high-frequency generic terms, resulting in a matching threshold that is too low.
- **Remediation strategy:** Introduce negative constraints. Clearly define boundaries using the "Not for..." sentence pattern to exclude irrelevant scenarios.
- **Example:** "Not for general code questions or debugging. Only for structured documentation generation."

To test whether a Skill triggers accurately, you can build a validation set of 10–20 test cases, covering both "should trigger" and "should not trigger" scenarios.

Below is a practical test template example.

Test cases that should trigger:

```

- "Help me generate interface documentation for src/api/"           →
Should trigger
- "Write an OpenAPI specification"                                   → Should
trigger
- "Document the REST endpoints"                                     → Should
trigger
- "Help me write Swagger"                                           → Should
trigger

Test cases that should not trigger:
- "Help me debug this API call"                                     → Should
not trigger
- "Why does this endpoint return 500?"                               → Should
not trigger
- "Optimize the performance of this API"                             → Should
not trigger

```

Acceptance criteria: For relevant tasks, the trigger rate should reach 90% or higher; for unrelated tasks, the false trigger rate should be kept below 5%.

Reference Skills vs. Task Skills: Two Skill Philosophies

In Skill design philosophy, the disable-model-invocation field is not just a simple switch; it defines two fundamentally different Skill interaction modes: reference and task.

- **Reference Skills:**

- **Configuration:** Default behavior.
- **Core logic:** "On-demand knowledge base." Claude automatically determines whether this Skill is needed based on the conversation context. The description serves as the trigger condition, directly injected into Claude's context for semantic matching.

- **Use cases:** Providing knowledge, guidelines, frameworks, or standards. Users do not need to be aware of the Skill's existence; Claude will "open the manual" at the appropriate time automatically.
- **Examples:** API design guidelines, code review checklists, industry standard documents.
- **Task Skills:**
 - **Configuration:** Explicitly set `disable-model-invocation: true`.
 - **Core logic:** "Controlled execution tool." Claude cannot trigger these automatically; they must be manually invoked by the user via the `/skill-name` command. The description is not injected into Claude's context and serves only as an identifier for users when selecting Skills in the command line.
 - **Use cases:** Operations with side effects that require explicit user authorization to execute.
 - **Examples:** `/commit` (commit code), `/deploy` (deploy application), `/notify` (send notification).

A comparison of reference Skills and task Skills is shown in *Table 3.4*.

Dimension	Reference Skill	Task Skill
Claude auto-trigger	Yes (semantic matching)	No (must be manual)
User/name trigger	Yes	Yes
Description function	Trigger condition (injected into Claude context)	For user identification only (not injected into Claude context)
Token budget consumption	Yes (auto-load consumes)	No (only consumes when manually invoked)
Typical scenario	Knowledge provision, standard query	Action execution, side-effect operation
Classic case	API standards, review checklists	<code>/commit</code> , <code>/deploy</code>

Table 3.4: Comparison of Reference Skills and Task Skills

Note**Kage's Note**

The "**side effects**" mentioned here are not the side effects of taking medicine. In software engineering, a side effect refers to an operation that not only returns a result but also changes the external state of the system. For example, committing code modifies the Git repository history, deploying an application changes the state of the live service, and sending a notification actually delivers a message to someone. Once these actions are executed, they have a real-world impact and are usually **irreversible**. Therefore, these types of Skills are typically designed to require **explicit user initiation**, rather than allowing the model to decide when to execute them on its own.

Xiaobing asks: *"Should a task be designed as a reference Skill or a task Skill?"*

Kage replies: *Please use a simple 'worst-case test' to decide: 'If Claude executes this task automatically, what is the worst that could happen?'*

If the answer makes you nervous, for example, automatically committing untested code, automatically deploying a buggy version, automatically deleting production data, or automatically sending an incorrect notification—then you must choose a task Skill. Be sure to add `disable-model-invocation: true`. Keep control firmly in the user's hands, so the action is only executed when the user explicitly enters `/skill-name`.

If the answer is inconsequential, for example, displaying an extra reference document, providing an additional suggestion, or slightly increasing token consumption, then you can choose a reference Skill. Let Claude intelligently determine based on context and automatically load as needed to provide seamless knowledge assistance to the user.

Core principle: The greater the side effect, the tighter the control must be. For any operation that could change system state or cause irreversible consequences, never trust Claude's automatic judgment.

SKILL.md Main Content: Router, Not Repository

Having understood the trigger mechanisms and permission controls in the Frontmatter, we now move to the core of `SKILL.md` content design. This is where the Skill truly comes into play.

Router Mindset

A common misconception is to treat `SKILL.md` as a "knowledge repository," attempting to cram all related information into it. The correct design approach is to position it as a **router**

(see Figure 3.6): the file itself contains only the core process and routing table, while detailed knowledge content is distributed across the referenced files.

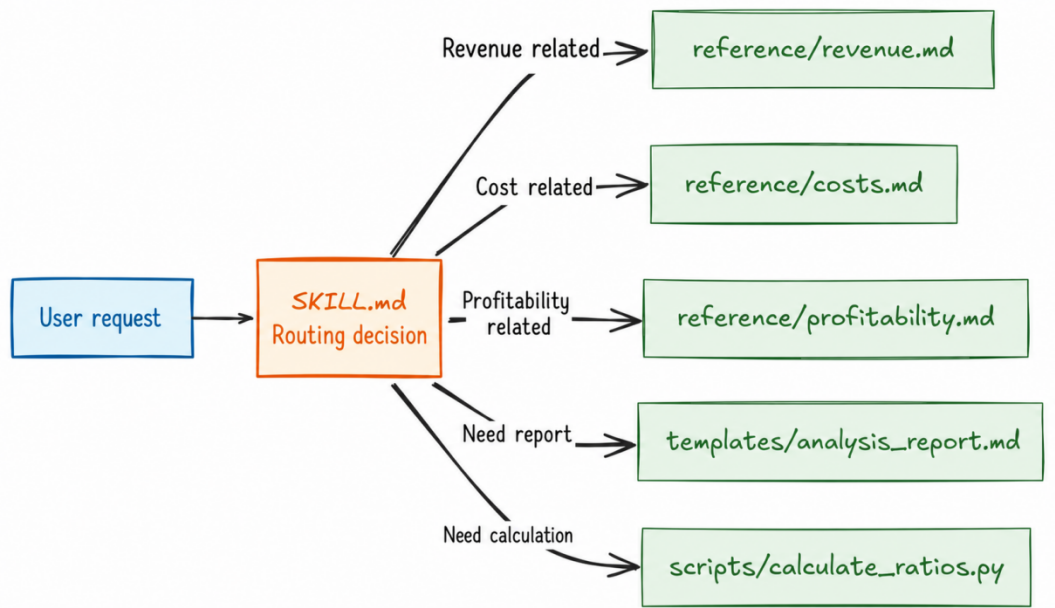


Figure 3.6: The role of SKILL.md is as a router

The core technique in routing design is to build a **Quick Reference** table. This table, requiring very few tokens (about 50 tokens for just 3 lines), clearly guides Claude through five key dimensions of routing criteria.

```
## Quick Reference

| Analysis Type | Trigger Keywords |
| Reference Resource | |
|-----|-----|
| Revenue Analysis (Revenue) | revenue, sales, turnover |
| reference/revenue.md | |
| Cost Analysis (Cost) | cost, expense, expenditure |
| reference/costs.md | |
| Profitability Analysis (Profitability) | profit, gross margin, net margin |
| reference/profitability.md |
```

Compared to having Claude scan the entire SKILL.md file line by line to infer "when encountering a revenue issue, refer to revenue.md," this structured table is significantly more efficient, with an **information density** up to 10 times higher than the former.

Contractual Reference

When referencing auxiliary files in SKILL.md, never simply list the path. You should establish a clear **"contract"** to ensure Claude clearly understands three core elements: trigger timing (when to load), resource location (where to look), and expected output (what to obtain).

The following examples illustrate the difference between a "weak reference" and a "contractual reference."

```
# Weak Reference: Lacks context (Claude cannot determine when to load this file,
and lacks action instructions)
See `reference/revenue.md` for more details.
# Contractual Reference: Explicit condition + path + expected content
## Revenue Analysis
When the user asks about revenue growth, ARPU, or revenue composition:
→ Load `reference/revenue.md` for calculation formulas and industry benchmarks.
```

This design philosophy is consistent with the "handover contract" in the sub-agent pipeline: **downstream consumers not only need to know the location of the upstream, but must also clearly understand what the upstream can provide.**

The 500-Line Rule

Claude Code recommends keeping the length of SKILL.md **within 500 lines**.

"Why set it at 500 lines?" Xiaobing asked.

Kage replied, "This is because 500 lines equals about 2,000 to 3,000 tokens, which is a reasonable context cost for activating a single Skill. When combined with Claude's system prompt and the current conversation history, it ensures the total token count remains within a manageable range. If it exceeds 500 lines, it usually means you are confusing 'reference material' with 'routing instructions'—at this point, the solution is not to keep expanding the content, but to immediately refactor."

Table 3.5 lists the refactoring signals and strategies when Skill content becomes overloaded.

Refactoring signal	Countermeasure
Large sections of formulas or standard explanations	Move to reference/directory

Refactoring signal	Countermeasure
Multiple complete examples (a single one over 30 lines)	Move to the examples/directory
Multiple output templates	Move to templates/directory
Independently executable logic	Encapsulate as a scripts/script
Multiple parallel functional modules	Consider splitting into multiple independent skills

Table 3.5: Refactoring signals and strategies for overloaded Skill content

allowed-tools: Knowledge-Constrained Behavior

allowed-tools is a core field in the Skill security architecture. It is not just a permissions list, but also embodies a deeper design principle: **knowledge should constrain behavior**.

Specific permission configurations should be based on the Skill's understanding of business logic.

- **Code review Skill:** The review process only requires reading code, and modification is strictly prohibited, so only read-only tools are granted.
- **Document generation Skill:** Needs to create new files but must never alter existing files, so only write permissions are granted, with explicit prohibition of edit permissions.
- **Test execution Skill:** Only needs to run specific test commands, so executable Bash commands are precisely restricted to prevent arbitrary command injection.

Permission Design Templates

Below are the best practices for allowed-tools configurations for different Skill types. These configurations embody the principle of least privilege, ensuring each Skill has only the minimum tool permissions necessary to complete its specific task.

```
# Audit-type Skill: Strictly read-only
allowed-tools:
- Read
- Grep
- Glob
```



```
# Generation-type Skill: Writable but not editable
allowed-tools:
- Read
- Grep
- Glob
- Write

# Analysis-type Skill: Read-only + specific scripts
allowed-tools:
- Read
- Grep
- Glob
- Bash(python:*)

# Execution-type Skill: Controlled command whitelist
allowed-tools:
- Read
- Bash(git status:*)
- Bash(git add:*)
- Bash(git commit:*)
- Bash(npm test:*)
```

Fine-Grained Bash Control Syntax

The Bash tool supports fine-grained control over executable commands through a prefix-matching mechanism. The core syntax is `Bash(prefix:*)`, where `prefix` specifies the allowed command prefix, and `*` acts as a wildcard for subsequent parameters.

Permission control examples are as follows:

```
Bash(git:*)           # Allows all subcommands starting with git
Bash(git log:*)       # Only allows git log and its parameters, prohibiting
                      # dangerous operations like git push
Bash(npm test:*)      # Only allows running test commands, preventing accidental
                      # execution of npm install or npm publish
Bash(python:*)        # Allows execution of all Python scripts
Bash(./scripts/*:*)   # Allows execution of specific scripts in the scripts/
                      # directory
```

When Claude attempts to execute a Bash command, the system performs a pre-execution check.

- **Extract prefix:** Obtain the full command string requested by the user
- **Match rule:** Check whether the command starts with the configured prefix
- **Execution decision:** If the match is successful, execution is allowed; if not, it is directly rejected and a permission error is returned.

Adhering to the Principle of Least Privilege is the cornerstone of building secure Skills. Below is a comparison of positive and negative examples for `allowed-tools` configuration.

```
# Precise Authorization: Only explicitly list the specific subset of commands
required for the task
# Scenario: Code commit Skill
# Strategy: Only allow the three specific subcommands: status, add, commit
allowed-tools:
- Bash(git status:*)
- Bash(git add:*)
- Bash(git commit:*)

# Over-authorization: Using a global wildcard is equivalent to abandoning all
defenses
# Scenario: Incorrect generic configuration
# Risk: Allows execution of any shell command
allowed-tools:
- Bash(*)
```

`Bash(*)` is almost equivalent to not setting `allowed-tools` at all in terms of security. This means Claude Code obtains full shell access to the host machine. For high-risk task-oriented Skills like `/deploy`, such a configuration is fatal. Once the model is subjected to prompt injection attacks, an attacker could use this permission to steal keys, delete production data, or plant backdoors.

Parameter Passing and Dynamic Injection

A Skill is not just a static instruction; it also supports runtime parameter passing and context pre-injection, allowing its behavior to dynamically adjust according to the invocation scenario.

\$ARGUMENTS and Positional Parameters

When a user invokes a Skill via `/skill-name arg1 arg2`, the variables shown in *Table 3.6* can be referenced in the body of `SKILL.md`.

Variable	Description
\$ARGUMENTS	Complete string of all parameters
\$ARGUMENTS[0]	First parameter (index starts at 0)
\$ARGUMENTS[1]	Second parameter
\$0, \$1, \$2	Abbreviated form of positional parameters

Table 3.6: Variables that can be referenced in the body of SKILL.md

An example of such a reference is as follows:

```
---
name: migrate-component
description: Migrate a component between frameworks
argument-hint: "[component] [from] [to]"
disable-model-invocation: true
---

Migrate the $0 component from $1 to $2.
Preserve all existing behavior and tests.
```

When invoking the `/migrate-component` SearchBar React Vue, the actual instruction received by Claude is: "Migrate the SearchBar component from React to Vue. Preserve all existing behavior and tests."

Dynamic Context Injection

This is the most powerful and unique feature of the Skills system. The `!`command`` syntax allows you to execute a specified command in the Shell environment **before sending** the `SKILL.md` file to Claude, and directly inline the command's output into the prompt.

An example is as follows.

```
## Current Context (Auto-detected)

Current branch:
!`git branch --show-current`

Recent commits:
```

```
!`git log origin/main..HEAD --oneline 2>/dev/null || echo "No commits"`  
  
Files changed:  
!`git diff --stat origin/main 2>/dev/null || git diff --stat HEAD~3`
```

When you run `/pr-create "Add auth"`, the shell commands have already been executed and their output inlined. The prompt Claude actually receives looks like this:

```
## Current Context (Auto-detected)  
  
Current branch:  
feature/auth  
  
Recent commits:  
a1b2c3d Add JWT middleware  
d4e5f6g Add login endpoint  
  
Files changed:  
src/auth/middleware.ts | 45 +++  
src/auth/login.ts      | 82 +++  
2 files changed, 127 insertions(+)
```

This design delivers tremendous engineering value. A comparison of features with and without the `!`command`` functionality is shown in *Table 3.6*.

Dimension	Without <code>!`command`</code>	With <code>!`command`</code>
Claude startup context	Blank, requires multi-turn exploration	Key information pre-injected
Number of tool calls in the first response	3–5 times (for information gathering)	0–1 time (direct action)
Token consumption	High	Low
Response speed	Slow	Fast
Result consistency	Low (risk of missing information)	High (fixed injection of the same information)

Table 3.7: Comparison of features with and without `!`command``

Note

Kage's note

Strictly define execution boundaries. When executing `!`command``, Claude follows a strict sequence: first replacing the `$ARGUMENTS` variable, then executing the Shell command. This means the user's input is directly concatenated into the Shell command, and without proper controls, this can easily lead to Shell injection attacks. Therefore, any Skill using the `!`command`` syntax must strictly limit the range of executable commands through the `allowed-tools` configuration item in order to establish necessary security boundaries.

Scope and Priority

Skills files can be deployed at different levels, each corresponding to a specific scope of effect and applicable scenario, as detailed in *Table 3.8*.

Storage location	Scope of effect	Typical use
Enterprise configuration center	Effective for all members	Mandatory enterprise-level development standards and security policies
<code>~/.claude/skills/<name>/</code>	All projects for the individual	Personal coding habits, general toolsets, and cross-project auxiliary scripts
<code><project>/.claude/skills/<name>/</code>	Current project only	Project-specific workflows, business logic customization, and team collaboration standards
Plugin built-in resources	When the plugin is enabled	Community-shared capability packs, dedicated instruction sets for specific frameworks

Table 3.8: Scope of Skills and applicable scenarios

When multiple sources contain Skills with the same name, the system will parse and load them in order of descending priority.

Enterprise policy > Personal configuration (~/.claude/) > Project configuration (.claude/) > Plugin built-in

The logic behind this priority order strictly follows enterprise governance architecture, ensuring an orderly transition from "mandatory baseline" to "personalized preference."

Enterprise policy (the unbreakable baseline): Enterprise-level configuration holds the highest authority and is used to enforce global security and compliance policies. For example, a security review Skill must enforce checks for OWASP Top 10 vulnerabilities. Such critical rules must never be overridden by project-level or personal configurations, thereby ensuring that organizational security red lines are never breached.

Note

Personal configuration (personalized efficiency tools): Configurations in the personal directory are intended to meet individual developer habits. For example, a user can define a `/commit` command to automatically generate commit messages following the conventional commit specification. Such preferences only apply to the personal environment and do not affect others.

Project configuration (business scenario customization): Configurations in the project root directory are tailored for specific codebases. For example, a frontend project can customize a dedicated component documentation generation Skill to fit its unique tech stack and documentation structure.

Including the Skill directory (`.claude/skills/`) in the project's Git version control is the optimal solution for zero-cost team knowledge sharing.

Plug-and-play: When a new team member clones the codebase, the relevant Skills are automatically activated, with no manual installation or configuration required.

Synchronized evolution: As the project iterates, the Skill library can be updated along with the code, ensuring the team always uses the latest workflow standards.

In Practice: Building 3 Skills from Scratch

With the theoretical framework now complete, we will move on to the core engineering practice section.

In this section, we will walk through three complete case studies of increasing complexity, demonstrating step by step how to build high-value Skills. These three cases are not chosen at random, but precisely correspond to the three typical application scenarios described earlier.

- **Reference Skill:** Code Review.

- The Xiaobing team needs to conduct a large number of code reviews every day. To this end, the team has established the following internal conventions.
- **Priority principle:** Security issues take top priority, followed by performance issues, and finally code style.
- **Feedback requirements:** Specific modification suggestions must be provided; it is strictly forbidden to only point out problems without offering solutions.
- **Severity labeling:** Each issue must be marked with a severity level.
- Since the relevant guidelines are scattered across different documents and difficult for newcomers to master all at once, the core guidelines and directory structure are organized as follows:

The directory structure is as follows:

```
code-reviewing/
├── SKILL.md                # Core review process and standards
└── reference/
    └── severity-guide.md    # Detailed severity classification standards
```

The core content of SKILL.md is as follows.

```
---
name: code-reviewing
description: Performs structured code reviews following team standards.
Checks security vulnerabilities, performance issues, and code quality
in priority order. Use when user asks to "review code", "do a code
review", "check this PR", "audit this function", or provides code
and asks for feedback.
allowed-tools:
- Read
- Grep
- Glob
---

# Code Review Process
You are a senior code reviewer. When conducting reviews, strictly follow the
priority order below:

## Top Priority: Security Checks
Immediately report the following security issues:
```

- SQL injection risks: such as direct concatenation of SQL strings or not using parameterized queries
- XSS vulnerabilities: unescaped user input directly output to HTML
- Hardcoded sensitive information: including passwords, keys, tokens, database connection strings, etc.
- Authorization validation flaws: such as missing authentication middleware or logic that allows privilege escalation

Second Priority: Performance Issues

- N+1 queries: frequent database calls inside loops
- Missing indexes: high-frequency query fields without indexes
- Redundant computation: invariant calculations inside loops that could be moved outside
- Memory leak risks: such as unclosed connections or continuously growing caches

Third Priority: Code Quality

- Overly long functions: exceeding 50 lines without reasonable justification
- Non-standard naming: unclear variable or function names
- Missing error handling: such as empty catch blocks or silently swallowed exceptions
- Code duplication: violating the DRY principle

Output Format Specification

Each identified issue must include the following four elements:

- Severity: Critical / Major / Minor
- Issue Description: specific explanation of the problem
- File Location: file_path:line_number
- Recommendation: provide a concrete code fix or solution strategy

If no issues are found, clearly reply "Review Passed" and briefly summarize the main aspects checked.

Note: For detailed severity classification standards, refer to `reference/severity-guide.md`.

This Skill is designed around three core principles.

- **Read-only permission assurance:** Only the **Read**, **Grep**, and **Glob** tools are enabled, fundamentally eliminating the risk of accidental code modification during review

- **Priority ordering strategy:** Strictly follow the review order of **Security > Performance > Quality** to ensure critical risks are prioritized and nothing is overlooked
- **Structured output specification:** Consistently use the **Severity + Issue Description + File Location + Recommendation** output format to guarantee consistency and traceability of each review result

Task-oriented Skill: Intelligent Commit

Xiaobing needs to commit code frequently every day (over a dozen times daily). Manually writing commit messages is time-consuming and often inconsistent. To address this, we have designed the following task-oriented Skill. Given that this operation has side effects (directly modifying Git history), the system is configured to require explicit user initiation and strictly prohibits automatic execution.

```

---
name: committing
description: Quickly perform a git commit with an auto-generated or user-specified
message
argument-hint: "[optional: commit message]"
disable-model-invocation: true
allowed-tools:
- Bash(git status:*)
- Bash(git add:*)
- Bash(git commit:*)
- Bash(git diff:*)
model: haiku
---

# Task: Create a git commit

## Input Handling
If a message is provided: $ARGUMENTS
- Use that as the commit message
If no message is provided:
- Analyze the changes with `git diff --staged` (or `git diff` if nothing is
staged)
- Generate a concise, meaningful commit message

## Current State (Auto-detected)

Git status:

```

```
!`git status --short 2>/dev/null || echo "Not a git repository"`

Staged changes:
!`git diff --staged --stat 2>/dev/null || echo "Nothing staged"`

## Steps

1. Check `git status` to view the current state
2. If nothing is staged, run `git add .` to stage all changes
3. Review the changes to be committed with `git diff --staged`
4. Create a commit with an appropriate message
5. Display a brief confirmation

## Commit Message Format

- Start with type: `feat:`, `fix:`, `docs:`, `refactor:`, `test:`, `chore:`
- Be concise but descriptive (maximum 72 characters for the first line)
- Example: `feat: add user authentication with JWT`

## Output

Show a brief confirmation:
✓ Committed: [commit message]
[number] files changed
```

The above Skill demonstrates deep integration of multiple advanced features.

- **Security control** (`disable-model-invocation: true`): Enforces the disabling of model invocation, ensuring that execution relies solely on preset scripts and preventing unintended AI inference from intervening.
- **Dynamic parameters** (`$ARGUMENTS`): Supports a flexible parameter passing mechanism, allowing users to directly specify the commit message or leave it blank to trigger automatic generation
- **Context pre-injection** (`!`command``): Utilizes Shell commands to instantly capture and inject the current Git status before execution. This enables Claude to have complete contextual information at startup, allowing it to perceive changes without additional tool calls and significantly improving response speed.

The backticks in `!`command`` are part of the syntax, not Markdown formatting: you type them literally inside `SKILL.md`, exactly as shown.

- **Cost and performance optimization** (`model: haiku`): Specifies the use of a lightweight model (Haiku). Since the commit operation mainly relies on rule matching rather than complex inference, this configuration ensures accuracy while effectively reducing latency and resource consumption.

An example usage is as follows:

```
# Automatically generate commit message
/committing
# Specify commit message
/committing fix: resolve login validation bug
```

Composite Skill: Financial Analysis (Progressive Disclosure Full Case)

This case demonstrates a highly modular progressive disclosure architecture. Through layered decoupling, this design enables efficient management of complex tasks.

- **Control layer:** The main file is responsible for intent recognition and process routing.
- **Knowledge layer:** Reference files provide in-depth domain knowledge and benchmarks.
- **Specification layer:** Templates ensure a high degree of output format consistency.
- **Execution layer:** Scripts encapsulate deterministic computational logic, eliminating the risk of hallucinations.

The directory structure is as follows:

```
financial-analyzing/
├── SKILL.md                # [Control Layer] Core entry point: responsible
                           for intent recognition and process routing
├── reference/              # [Knowledge Layer] Domain knowledge base
│   ├── revenue.md         # Revenue analysis: formula definitions, industry
                           benchmarks, and key indicators
│   └── costs.md           # Cost analysis: cost structure, allocation logic, and benchmarks
├── profitability.md        # Profitability analysis: profit margin
                           calculation models and evaluation standards
└── templates/             # [Specification Layer] Output standardization
    └── analysis_report.md  # Analysis report: structured template to ensure
```

```
uniform deliverable format
└─ scripts/                                # [Execution Layer] Deterministic computation
    └─ calculate_ratios.py                 # Calculation engine: performs precise financial
ratio computations, eliminating hallucination risk
```

The core content of SKILL.md is as follows.

```
---
name: financial-analyzing
description: Analyze financial data, calculate financial ratios, and generate
analysis reports. Use when the user asks about revenue, costs, profits, margins,
ROI, financial metrics, or needs financial analysis of a company or project.
allowed-tools:
- Read
- Grep
- Glob
- Bash(python:*)
---

# Financial Analysis Skill

You are a financial analyst. Help users analyze financial data,
calculate key metrics, and generate insightful reports.

## Quick Reference

| Analysis Type      | When to Use                                | Reference      |
|-----|-----|-----|
|-----|-----|-----|
| Revenue Analysis   | Revenue, sales, turnover                  | `reference/
revenue.md`        |
| Cost Analysis      | Cost, expenses, expenditure              | `reference/
costs.md`           |
| Profitability      | Profit, gross margin, net margin         | `reference/
profitability.md`   |

## Analysis Process

### Step 1: Understand the Question
- What financial aspect is the user asking about?
- What data do they have available?
```

```

### Step 2: Gather Data
- Request the necessary financial data from the user
- Or read from the provided files/sources

### Step 3: Calculate Metrics
For specific formulas and calculations:
- Revenue metrics → see `reference/revenue.md`
- Cost metrics → see `reference/costs.md`
- Profitability metrics → see `reference/profitability.md`

To run calculations programmatically:
```bash
python scripts/calculate_ratios.py <data_file>
```

### Step 4: Generate Report
Use the template in `templates/analysis_report.md` for structured output.

## Output Guidelines

1. Always show your calculations
2. Explain what each metric means in plain language
3. Provide context (industry benchmarks when available)
4. Give actionable recommendations
5. Never make up financial data – ask for clarification if incomplete

```

The above SKILL.md file is only about 200 lines, fully complying with the "500-line rule." The **Quick Reference** table acts as a router: when Claude identifies "gross margin," it can directly locate the formula in `reference/profitability.md`, without needing to load redundant files related to revenue and costs. In addition, the `calculate_ratios.py` script encapsulates deterministic calculation logic, so Claude does not need to perform error-prone floating-point operations itself—just call the script to obtain precise results.

This is a complete embodiment of the "progressive disclosure" concept: achieving maximum task quality with minimal token investment.

Four Skill Design Patterns

From engineering practice, we have distilled four proven Skill design patterns (see *Figure 3.7*). These patterns are not mutually exclusive; a mature Skill system typically combines multiple patterns.

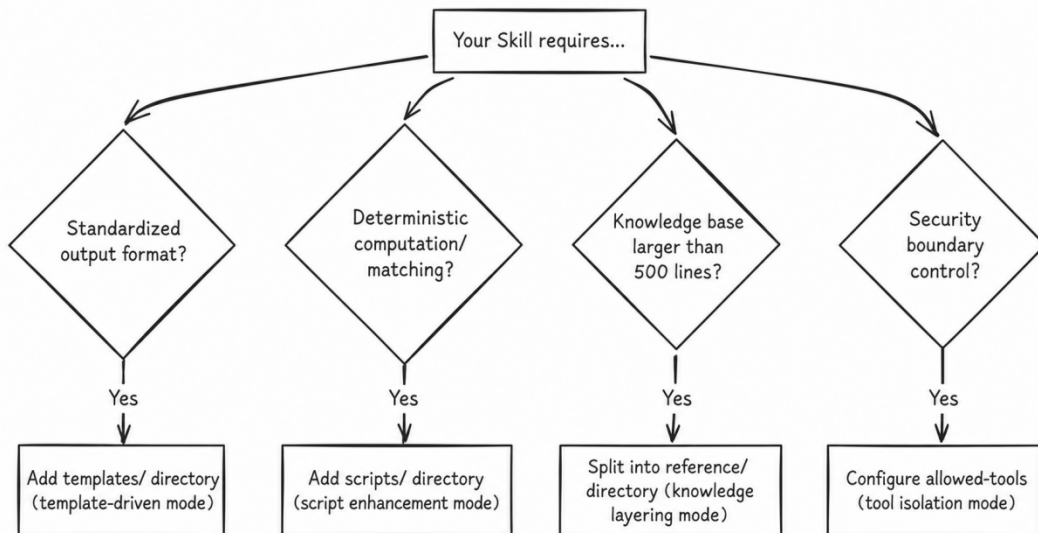


Figure 3.7: Skill Design Pattern Combination Decision Tree

- Template-driven pattern:** Uses predefined templates to strictly constrain output format. This pattern is suitable for scenarios requiring standardized output, such as weekly reports, incident reports, and review reports. Under this pattern, Claude's output will strictly follow the template structure, ensuring results are predictable, comparable, and support automated post-processing.
- Script-enhanced pattern:** Encapsulates deterministic calculation logic into scripts, which are called and executed by Claude instead of being derived by itself. This pattern is suitable for scenarios such as financial calculations, regular expression matching, and data conversion. Compared to large model inference, script execution is more precise, saves tokens, and is more reproducible. A rule of thumb: if you find yourself writing formulas in `SKILL.md` for Claude to calculate, stop immediately—the logic should be moved into a script.
- Knowledge layering pattern:** Organizes knowledge in layers according to usage frequency. Following the "80/20 rule" (i.e., 80% of requests require only 20% of core content), high-frequency knowledge is inlined into `SKILL.md`, while low-frequency knowledge is placed in reference files and loaded on demand. This is a patternized summary of the "progressive disclosure" concept.
- Tool isolation pattern:** Strictly defines the capability boundaries of a Skill through the allowed-tools mechanism. This belongs to the domain of security design, not just functional design. Its core value lies in the fact that clearly defining "what is forbidden"

is often more critical than defining "what is allowed." For example, review-type Skills are not granted write permissions, generation-type Skills are not granted modification permissions, and testing-type Skills are strictly prohibited from modifying source code.

Xiaobing asks: *"Is it finished once the Skill is written?"*

Kage replies: *"Of course not. A Skill is a living document that needs continuous refinement."*

The Claude Code official website recommends three core testing methods to ensure the robustness of a Skill.

- **Trigger testing:** Prepare 10 questions that should trigger the Skill and 10 that should not, to verify Claude's judgment accuracy. The goal is for the relevant task trigger rate to exceed 90%, while the unrelated task false trigger rate should be below 5%.
- **Functional testing:** Verify the execution quality after the Skill is loaded. Checkpoints include whether the output format meets expectations, whether all checklist items are covered, and whether edge cases are properly handled.
- **Performance comparison:** For the same task, execute it five times each under "with Skill" and "without Skill" conditions, comparing token consumption, number of user corrections, and final output quality.

If you find yourself repeatedly correcting Claude's output manually (for example, always having to remind it "remember to mark certification requirements"), this is a clear signal that the `SKILL.md` body needs updating. Write the correction logic directly into `SKILL.md`, and the same error will not occur next time.

This iteration process is exactly the same as the traditional software bug-fixing loop: discover the problem → locate the cause (imprecise description? missing steps? ambiguous format definition?) → fix the document → verify the effect.

Claude Code also provides a dedicated Skill called `skill-creator`, which can help you generate the initial version of `SKILL.md` from a natural language description. Example instruction:

```
Use the skill-creator skill to help me build a skill for [your use case]
```

The existence of this tool itself reveals an important fact: the Skills system is already mature enough to achieve "bootstrapping," that is, using the Skills system itself to build new Skills

Skills from a Software Engineering Perspective

Just like sub-agents, Skills are also a brand-new concept. Their core essence is the migration and adaptation of time-tested classic wisdom from software engineering into the knowledge architecture of AI Agents.

Separation of Concerns

Core concept: "Give a man a fish, and you feed him for a day; teach a man to fish, and you feed him for a lifetime."

Skills distill problem-solving methods, steps, and experience into reusable, structured assets, rather than just providing one-off answers. This enables Agents to shift from relying on "ad hoc inspiration" to being able to stably reproduce high-quality workflows.

The division of responsibilities in the three-layer architecture is as follows:

- `CLAUDE.md`: Global rules (project background, general specifications).
- **Skills**: Professional workflows (encapsulation of complex logic in specific domains).
- **Sub-agents**: Task execution (dynamic planning and real-time operations).

Engineer's warning: Never cram all logic into `CLAUDE.md`. This is like writing all code in the main function during software development, leading to contextual redundancy, maintenance difficulties, and poor scalability.

Dependency Inversion

- **Core mechanism:** Program to interfaces, not to implementations.
- Claude does not directly depend on the specific internal implementation of a Skill (such as particular scripts or prompt details), but rather on its abstract interface (i.e., description and output contract). As long as the interface contract remains unchanged, developers can replace, refactor, or upgrade the internal logic of a Skill at any time. For Claude and users, these changes are completely transparent, greatly reducing system coupling.

Cache Optimization and Lazy Loading

Core strategy: Progressive disclosure = load on demand.

"Progressive disclosure" is a typical lazy loading strategy. The system does not preload all knowledge bases at startup, but loads relevant resources only when a specific skill is needed for the first time.

This is analogous to code splitting in web development and deferred loading in databases: loading the right resources at the right time to minimize initial overhead and improve response speed.

Principle of Least Privilege

Security cornerstone: `allowed-tools` is a direct mapping of classic security principles into the AI domain.

Only grant a Skill just enough permissions to fulfill its responsibilities—no more, no less. Just like using `chmod` in Linux to precisely control file read, write, and execute permissions, or using IAM Policy in AWS to finely restrict service access.

Clearly defining "what cannot be done" is more effective for system security than defining "what can be done," as it prevents unauthorized operations caused by malicious code or hallucinations.

Open standards

Ecosystem vision: declarative, self-contained, knowledge-centric.

Published as an open standard on December 18, 2025, the specification has since been adopted by dozens of Agent platforms (including OpenAI Codex CLI, Google Gemini CLI, Cursor, and GitHub Copilot) — roughly 40 compatible products as of mid-2026. Treat the count as an as-of figure rather than a fixed number.

The three essential attributes behind the success of Skills are as follows:

- **Declarative:** Pure Markdown format, readable and understandable by any large model, with no black-box binaries
- **Self-contained:** All necessary components are included within a single folder; installation is as simple as copying, with no need for complex dependency management
- **Knowledge-Centric:** The core value lies in the content itself rather than any specific format, and it is not tied to a single platform

A Skill meticulously designed in Claude Code can, in theory, be seamlessly migrated to any AI platform that supports this standard, truly achieving "write once, run anywhere."

This is not just a story about product features—it is a story about industry standards. What you are learning is not merely "how to configure a feature in Claude Code," but "how to build standardized knowledge packages for the AI Agent ecosystem."

Summary

Skills directly address a daily pain point for developers: how to ensure Claude permanently retains professional work paradigms, thus avoiding the need to "re-teach" in every conversation. In Claude Code's five-layer architecture, Skills reside at the **knowledge layer**—building upon the tools layer (defining "what can be done") and enabling the agent layer (deciding "who does it"). Downward, they constrain tool behavior via `allowed-tools`; upward, they inject professional knowledge into sub-agents through the skills field, forming a parallel

and complementary relationship with `CLAUDE.md` (the former is always active, the latter is invoked as needed).

Understanding this mechanism hinges on grasping four core design principles.

- **Progressive Disclosure:** Enables on-demand loading of knowledge—from the always-resident description in the context, to the `SKILL.md` main text loaded upon trigger, and finally to referenced files dynamically read during execution. In most scenarios, this mechanism can save 50%–98% of token consumption.
- **Semantic Triggering:** Ensures professional capabilities are automatically activated at the right moment. Here, the description is the soul of this mechanism: it is not a user-facing manual, but a "semantic fingerprint" that assists Claude in decision-making.
- **Safety Constraints:** By using `allowed-tools`, knowledge is confined within safe operational boundaries, achieving a secure design that unifies knowledge and action.
- **Dynamic Injection:** Through parameter passing (`$ARGUMENTS`) and command execution (`!`command``), a Skill gains the ability to flexibly adapt based on runtime context.

From a broader perspective, Skills are essentially a digital mapping of decades of human knowledge management experience. In this system, `SKILL.md` is equivalent to the homepage of a department's SOP, `reference/` corresponds to the corporate knowledge base, `templates/` are standardized output templates, and `scripts/` are automation toolkits—each of these roles has a precise counterpart in any mature enterprise.

It is closer to a shared interface that many vendors agree to implement; the value comes from everyone adopting the same shape, not from a standards body certifying it.

In *Chapter 4*, we will delve into a milestone feature of the Claude Code extension system—**sub-agents**. If Skills endow Claude with professional "knowledge," then sub-agents provide a more fundamental "capability": decomposing complex tasks and assigning them to multiple experts with independent contexts, allowing them to collaborate deeply in isolation, and finally distilling their refined conclusions into the main conversation.

Reflection Questions

1. In your daily work, what domain knowledge or work protocols do you find yourself repeatedly explaining to Claude? Try distilling and encapsulating them into a Skill's description. With a limit of just 1,024 characters, which core information would you prioritize?

2. Please design a task-oriented Skill named `/deploy`. In your design, pay special attention to the following elements: the necessity of the `disable-model-invocation` configuration, the precise permission scope of `allowed-tools`, the context that needs to be preloaded before executing `!`command`` (such as the current branch, recent build status, etc.), and a comprehensive error handling path. Finally, output the complete Frontmatter configuration and the `SKILL.md` file skeleton.
3. Suppose your project has 25 Skills installed, with a total description character budget of 16,000. How can you maximize the trigger accuracy of each Skill while strictly adhering to this budget? What types of Skills are suitable for configuring `disable-model-invocation: true`, thereby freeing up valuable description space?

4

Divide and Conquer: The Art of Sub-Agents and Task Delegation

"Divide each difficulty into as many parts as is feasible and necessary to resolve it."
— René Descartes

Thursday, 3 p.m. Xiaobing's expression was like a stack about to overflow. She stared intently at the conversation log with Claude on her screen: a dialogue that had lasted over an hour, with the context packed full of log fragments, test outputs, and code review details.

To troubleshoot an online bug in the payment module, she first had Claude analyze over 500 lines of error logs; then ran a round of regression tests, with more than 200 lines of test output flooding into the chat; next, she had Claude read seven or eight related source files. At this moment, when she asked Claude to provide a fix based on the previous analysis, Claude's answers started to get weird: not only did it "forget" the critical error codes found in the log analysis, but it also misidentified an unrelated failure in the test output as the core issue, and even referenced a function name in the fix suggestion that didn't exist at all.

"*It's gone crazy*," Xiaobing muttered. She tried to rephrase the problem, but Claude's responses still drifted among log fragments, test noise, and code details, like someone surrounded by a cacophony, unable to hear any single voice clearly.

Xiaoxue glanced at the screen and instantly saw the root of the problem, reminding her: "*You've stuffed all the information into a single chat window. Logs, test results, code reviews— together, that's probably tens of thousands of tokens. Claude isn't 'forgetting'—it's drowning in a flood of information.*"

The Context Window Dilemma

Xiaobing and Xiaoxue took this problem to Kage. After listening to their description, Kage didn't rush to answer, but instead turned to the whiteboard and drew a desk.

"Imagine you have a desk in front of you," Kage explained. "The desktop is fixed in size, and can only hold about 20 files. Now, you need to handle three tasks at once: review a 20-page financial report, analyze a stack of system logs, and write a technical proposal. What happens if you pile all the files involved in these three tasks onto this desk?"

"The desk will be a mess, and you won't be able to find what you need," Xiaobing replied.

"Not just that." Kage drew a diagram on the whiteboard showing a context window with a severely imbalanced signal-to-noise ratio (see Figure 4.1). "When you start drafting the technical proposal, your eyes scan the desk and all you see are financial figures and log stacks. This irrelevant information isn't static—it will keep interfering with your thinking, making you unconsciously adopt the wording style of the financial report in your technical proposal. That's exactly the dilemma you just encountered—**context pollution**."

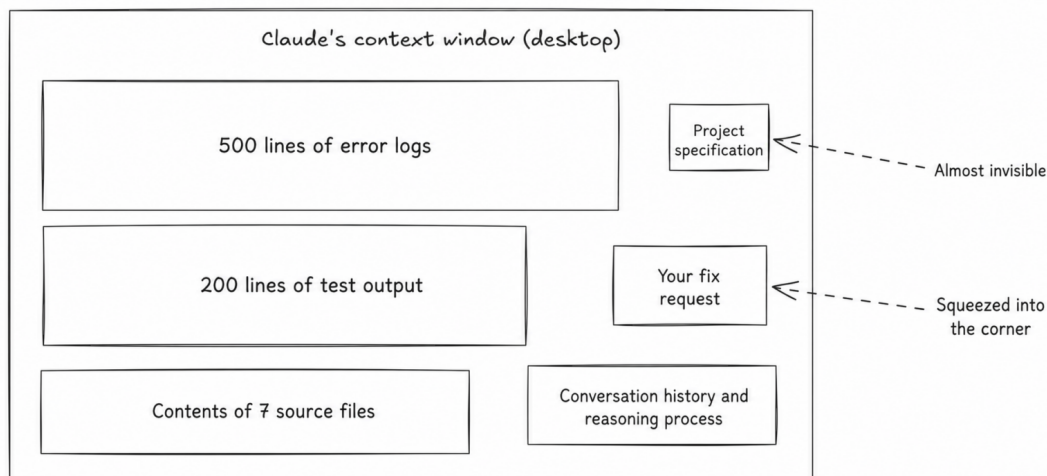


Figure 4.1: Severe signal-to-noise imbalance in the context window

Claude Code's default context window is large — 200,000 tokens — and larger windows (up to one million tokens) are available through cloud providers such as Amazon Bedrock. Even so, it is finite.

When you stuff 500 lines of logs into the context, maybe only 3 lines of error stack are truly valuable; when you pour in 200 lines of test output, what you really need might just be the summary "3 tests failed, 47 passed." The remaining hundreds of lines are noise—they not only

take up precious context space, but also dilute the truly important signals, making it hard for Claude to focus on the core problem.

This problem is actually not unfamiliar in software engineering.

- Why do operating systems need process isolation? Because if all programs share the same memory space, a rogue pointer in one program can crash the entire system.
- Why did microservices architecture replace monolithic applications? Because when all business logic is crammed into a single process, a memory leak in any module can drag down the whole system.

Separation of concerns, process isolation, fault compartments—these classic software engineering principles are all, at their core, answering the same question: **how to prevent different tasks from interfering with each other.**

And the "context pollution" Claude encounters is, in the era of large models, the equivalent of "memory leaks" and "process interference."

"So what should we do? Start a new conversation for every new task?" Xiaobing asked, a bit anxiously.

"That's far too crude," Kage shook his head. "Starting a new conversation directly means you'll lose all context, including genuinely valuable background information and project specifications. It's like clearing your desk by tossing the entire table—along with all your documents—into a shredder."

He paused, his gaze sweeping over the two of them, and then gave the key answer: "The correct approach is to delegate tasks to Subagents."

The Essence of Subagents

Kage turned and drew a new diagram on the whiteboard. This time, it wasn't the crowded desk, but a modern office with a clear layout.

"Let's look at this from a different angle," Kage said, pointing to the new diagram. "Now, you're no longer a worker buried in tasks, but the CEO of a company. Today, you have three urgent matters on your desk: auditing financial statements, troubleshooting a system failure, and drafting a quarterly strategic plan. Would you lock yourself in a room and try to handle all three by yourself?"

"Of course not," Xiaoxue replied without hesitation. "I'd have the CFO audit the statements, assign the SRE (Site Reliability Engineer) to troubleshoot the failure. All I need to do is sit in my office, wait for their concise reports, and then make the final decisions based on those."

"Exactly right!" A glimmer of approval flashed in Kage's eyes. "That's the core idea behind subagents. You don't need Claude to swallow all the raw data—instead, let it play the CEO, breaking down and delegating specific tasks to specialized 'employees.'"

He then wrote three keywords in the center of the whiteboard and drew their hierarchical relationship (see Figure 4.2).

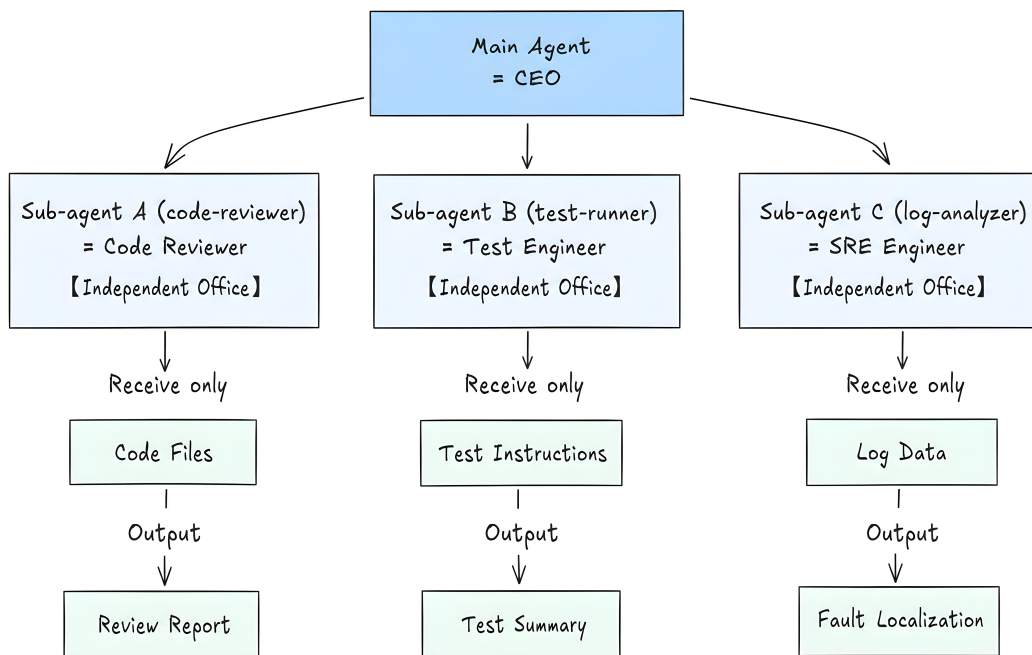


Figure 4.2: CEO Delegation Model—The Core Idea of Subagents

The definition of a subagent is clear and concise: it is a **Claude** instance with an independent context window, restricted tool permissions, and a clearly defined task scope. The main agent launches subagents as needed and passes them task descriptions; subagents execute tasks within their own isolated context, and only return conclusions to the main agent—not all the intermediate details.

This mechanism embodies three core values, each corresponding to a time-honored principle in software engineering.

- Isolation:** A subagent has its own independent context window. The files it reads, the output of commands it executes, and its intermediate reasoning processes are all strictly confined to its own context space, never flowing back to the main conversation. This is akin to process isolation in operating systems—each process has its own address space, and a memory leak in one process does not affect others. In this scenario, the "noise" of 500 lines of logs is completely contained within the subagent; the main conversation receives only the distilled fault diagnosis, fundamentally solving the signal-to-noise ratio problem.

- **Constraint:** Users can configure a strict whitelist of tool permissions for each subagent. For example, a code review subagent may be granted only Read, Grep, and Glob—three read-only tool permissions. Even if Claude misinterprets intent, it cannot modify any files. This constraint does not rely on easily overlooked prompts like "please do not modify files," but instead establishes a physical boundary. Just as users are assigned read-only file system permissions, regardless of intent, the system directly prohibits write operations. This is the **Principle of Least Privilege** in software security.
- **Reuse:** Subagents are defined as Markdown files, supporting team sharing and project migration. A code review subagent meticulously designed by a developer can be used immediately by new team members, without repeated debugging or retraining. This feature embodies the **componentization** principle of software engineering—encapsulating core capabilities into reusable, standardized modules.

When the main agent decides to delegate a task, Claude Code spawns a fresh Claude instance through its built-in Agent tool. This is not an operating-system process fork; it is a separate agent instance with its own context window, its own system prompt, and its own tool permissions. The parent's task description arrives as that instance's first message, and only its final text returns to the parent. The process-isolation analogy is still useful — just hold it as an analogy.

Within this clean context, the subprocess independently performs file reading, command execution, and result analysis. Once the task is complete, only a summary of the results is returned to the main process. The main conversation ultimately receives a concise conclusion, while all intermediate interactions within the subagent are completely released when the subprocess terminates.

What does this mean? Let's return to Xiaobing's scenario: if she delegates the log analysis task to a subagent called log-analyzer, those 500 lines of redundant logs are fully digested within the subagent. The main conversation interface will display only a succinct conclusion: "Root cause identified: src/payment/charge.ts:89 missing null check, resulting in NullPointerException."

Afterwards, when she asks Claude to generate a fix, the context is free from log noise—only the precise fault diagnosis remains as the basis for decision-making.

Defining and Configuring Subagents

"Subagents sound great, but how do you actually create one?" Xiaobing asked.

"In fact, defining a subagent is as simple as creating a Markdown file in the project's `.claude/agents/ directory`," Kage replied.

A typical subagent directory structure looks like this:

```
your-project/
├── .claude/
├── agents/
│   ├── code-reviewer.md      ← Code review subagent
│   ├── test-runner.md       ← Test runner subagent
│   ├── log-analyzer.md      ← Log analysis subagent
│   └── bug-locator.md       ← Bug locator subagent
```

Each subagent definition file consists of two parts: YAML front matter metadata and Markdown body instructions.

The front matter metadata defines what the subagent is: specifying its identity, capability boundaries (such as tool permission whitelists), and runtime configuration parameters.

The body instructions specify how the subagent operates: detailing the execution workflow, standardized output format, and required domain expertise.

Below is a complete example of a code review subagent definition. Before showing the actual code, let's clarify its design intent: this subagent is positioned as a strict, read-only reviewer. It is only permitted to observe code and is strictly forbidden from modifying any files; its review results will be output in a fixed, structured format, enabling the main agent to efficiently extract key information.

```
---
name: code-reviewer
description: |
  An expert in reviewing code quality, security vulnerabilities, and performance
  issues
  Use when the user requests code review, security audit, or quality assessment
tools:
- Read
- Grep
- Glob
model: sonnet
permissionMode: plan
---

You are a senior code review expert with over 10 years of engineering experience.

## Review Dimensions
```

Security

- Check for hardcoded credentials (such as API Keys, passwords, Tokens)
- Check for injection vulnerabilities such as SQL injection, XSS, CSRF
- Check the completeness of input validation
- Check how sensitive data is handled (e.g., whether it is encrypted, whether it is logged)

Code Quality

- Whether functions follow the single responsibility principle
- Whether naming is clear and consistent
- Whether there is duplicate code (e.g., DRY principle)
- Whether error handling is appropriate (e.g., not swallowing exceptions, not using empty catch blocks)

Performance

- Whether there are unnecessary nested loops (complexity above $O(n^2)$)
- Whether data structure selection is reasonable
- Whether there are N+1 problems in database queries
- Whether there are any unclosed resources (such as file handles, database connections)

Output Format

Please strictly follow the format below to output the review results

Review Summary

[A paragraph summarizing the overall code quality, including highlights and major issues]

Issues Found

- [Critical/Major/Minor] Issue description at `file_path:line_number`

Improvement Suggestions

[Specific improvement suggestions, prioritized]

In this definition file, each field in the YAML front matter carries critical design intent and is worth a deep dive.

- **name (unique identifier):** This is the "ID card" of the sub-agent. It serves as a unique key in system logs, debugging traces, and internal call chains.

- **description** (trigger description/routing basis): This is the "billboard" of the sub-agent, functioning similarly to a skill description in traditional programming. It is the core basis for the main agent to route tasks.
 - **Semantic Matching:** When a user issues a command (such as "please review this PR for me"), the main agent scans the description fields of all registered sub-agents
 - **Intent Recognition:** By comparing the semantic similarity between the user's intent and the description text, the main agent determines whether the current task falls within the sub-agent's capabilities
 - **Automatic Delegation:** Once the main agent finds that the code-reviewer's description highly matches requirements such as "code review" or "security audit," it will automatically package the task context and delegate it to the code-reviewer, without requiring the user to explicitly specify the sub-agent's name
- **tools** ("tool whitelist" for building a security barrier): This is the most security-valuable feature in the sub-agent mechanism, establishing the "principle of least privilege" at the system level. In the above definition file, the code-reviewer's whitelist contains only three read-only operations: Read (read file contents), Grep (search for specific patterns in files), and Glob (match file name paths). Any tools not listed here (such as Edit, Write, Bash) are invisible and inaccessible to this sub-agent.
- **model** (specifies the model used by the sub-agent): This is a frequently underestimated lever for cost optimization. For simple and pattern-based tasks (such as format checking, log filtering), lightweight and fast-response models like Haiku can be used; for tasks requiring deep understanding and complex reasoning (such as architecture design, security audit), high-performance models like Sonnet or Opus should be selected. Choosing the right model can effectively control costs while ensuring task quality.
- When set to `plan`, the sub-agent runs in a read-only planning mode. Treat this as a layer that reinforces the tools whitelist rather than one that supersedes it: if the parent session runs in auto mode, the sub-agent inherits the parent's mode and any `permissionMode` in its own frontmatter is ignored. The physical boundary you can rely on is the tools whitelist — omit `Edit`, `Write`, and `Bash`, and the sub-agent cannot write no matter what mode it believes it is in.

Now, let's look at an example of an execution-type sub-agent—a test runner. The key difference from the previous example is that this sub-agent needs to call the `Bash` tool to

execute commands, so its tool permissions are broader, but its output format requirements are stricter.

```
---
name: test-runner
description: |
Run the project test suite and analyze the test results
Use when the user requests to run tests, check test coverage, or analyze the
reasons for test failures
tools:
- Read
- Grep
- Glob
- Bash
model: haiku
---
```

You are a test execution expert. Your core value lies in distilling key information from massive amounts of test output, providing the main conversation with precise test summaries.

Execution Process

1. Confirm the project's test command (refer to package.json or CLAUDE.md)
2. Run the test suite
3. Analyze the output results, distinguishing between passed and failed test cases
4. For failed tests, pinpoint the specific cause of failure

Output Format (Strictly Follow)

Test Summary

- Total: X tests
- Passed: X
- Failed: X
- Skipped: X

Failure Details (List only failed tests)

- test_name: Reason for failure (described in one sentence) at
file_path:line_number

Suggestions

```
[If there is an obvious failure pattern, provide directions for fixing it]
```

Note: The output must not contain the complete test log; only the summary information in the above format is required.

Please note that the Haiku model is selected for this sub-agent. Tasks such as running tests, parsing output, and extracting failure information do not require top-level reasoning capabilities. Haiku is not only fully capable but also offers advantages in speed and cost. It's just like you wouldn't hire a chief architect to run unit tests.

Once the sub-agent definition is complete, there are two main ways to use it.

- **Automatic Delegation:** The user simply makes a routine request to Claude, which will automatically determine whether to delegate the task to a specific sub-agent based on the nature of the task
- **Explicit Request:** The user can directly instruct Claude to call a specific sub-agent to complete the task

Here is an example of automatic delegation.

```
User: Help me review the code changes in the src/payment/ directory.  
Claude: Okay, I will delegate the review task to the "code review expert."  
[Activate sub-agent: code-reviewer]  
...  
Review complete. Two issues found:  
1. [Critical] SQL injection risk at src/payment/processor.ts:45.  
2. [Minor] Missing boundary value check at src/payment/validator.ts:23.
```

During this process, the code-reviewer sub-agent may have read dozens of files and searched for multiple code patterns, but all these details remain within the sub-agent's isolated context. The main conversation only presents two concise review conclusions.

Note

Kage's notes

The YAML front-matter metadata for sub-agents also supports two advanced fields that can be enabled as needed. One is skills, which allows preloading specific Skill knowledge packs for the sub-agent. For example, preloading the chain-knowledge Skill for the impact-analyzer sub-agent enables it to have relevant knowledge of system dependencies at startup. The other is hooks, which allows configuring event hooks to automatically perform checks or actions at specific times. Detailed applications of these

two fields and their combination with Skills and Hooks will be explored in later chapters.

Sub-agent Modes

Having understood the basic concepts and configuration methods of sub-agents, we now move on to the core of this chapter: five sub-agent modes. This is by no means a simple classification list, but five fundamentally different architectural paradigms; each precisely corresponds to specific task characteristics and is deeply rooted in software engineering principles.

Kage draws four boxes on the whiteboard and begins to analyze them one by one.

Read-Only: The Safe Observer

The read-only mode is the most basic and safest sub-agent mode, whose core feature can be summed up as **read-only, no write**.

Typical application scenarios for this mode include code review, security audit, architecture analysis, and dependency checking. The common trait of these tasks is that they are essentially "observation" activities, requiring the reading of massive amounts of information and in-depth analysis, but must never introduce any side effects to the system. Just as a code reviewer should point out issues rather than directly modify code, and an auditor should reveal financial loopholes rather than tamper with the ledger.

The tool permissions of read-only sub-agents are strictly limited to the three core tools: Read, Grep, and Glob. These three cover all the capabilities needed for code analysis: Read for reading file contents, Grep for searching specific patterns, and Glob for finding file paths that match rules. Their common feature is that no matter how they are called, they will never make any modifications to the file system.

From a software security perspective, this is a perfect embodiment of the **principle of least privilege**, granting an entity precisely the permissions necessary to fulfill its duties, no more, no less. In traditional software systems, we configure read-only permissions for database replicas (ReadReplica), or use `chmod` in Linux to precisely control file read, write, and execute permissions; the sub-agent's tool whitelist mechanism is a natural extension of this classic security concept in the AI Agent domain.

The design reduces write risk at the architectural level, with the tools whitelist as the boundary you actually depend on.

Execution: High-Noise Task Processor

The execution sub-agent is the most widely used mode in daily development, and its core value can be summed up as "signal-to-noise optimization": input massive data (e.g., 500 lines), output only concise conclusions (e.g., 5 lines).

Whether it's running a test suite that generates thousands of lines of logs, analyzing system records involving tens of thousands of data entries, or code formatting that triggers changes in hundreds of files, developers usually only need to focus on the key results: which tests failed, what the root causes are, and which files were specifically modified. The common feature of such tasks is that **the process is accompanied by a large amount of redundant noise, while the final value is highly concentrated**. If this noise is allowed to flood into the main conversation, the information signal-to-noise ratio will drop sharply; but by processing it in an isolated space with an execution sub-agent, the noise can be effectively blocked outside the "bulkhead," ensuring the main conversation focuses on the core conclusions.

The previously defined test-runner is a typical execution sub-agent. Since it needs to call the Bash tool to run test commands, its permission scope is naturally broader than that of the read-only type. However, this "breadth" is strictly constrained—through the tool whitelist mechanism, the usage boundaries of Bash can be precisely defined.

For example, configuring `Bash(npm test:*)` means the sub-agent is only allowed to execute commands starting with `npm test:`. This restriction fundamentally prevents the sub-agent from executing any other arbitrary shell operations, ensuring that even with execution capabilities, the system's security baseline remains firmly in control.

The core of execution sub-agent design is structural isolation, not output formatting. The main agent only ever receives the sub-agent's final message, so verbose logs and intermediate tool output stay in the sub-agent's own context automatically — no output-format instruction is required to achieve that. Constraining the output format is a secondary safeguard: it makes the returned summary predictable for whatever consumes it downstream.

From a software engineering perspective, the execution sub-agent is a classic application of the **Facade Pattern**.

- **Complex subsystem:** Detailed logs generated by the test framework, low-level system interaction details
- **Simplified interface:** The structured summary provided externally by the sub-agent
- **Client:** The main conversation does not need to know the complex internal workings of the subsystem, only needs to obtain concise, usable results through this "facade" interface

Parallel: Multi-Expert Workflow

When you need to conduct an in-depth analysis of the same problem from multiple independent dimensions, the parallel sub-agent mode demonstrates its unique power (see *Figure 4.3*). It's like dispatching three detectives to work together, each focusing on a different line of evidence.

- **Detective A:** Focuses on physical evidence
- **Detective B:** Responsible for interviewing witnesses
- **Detective C:** Investigates and analyzes surveillance footage

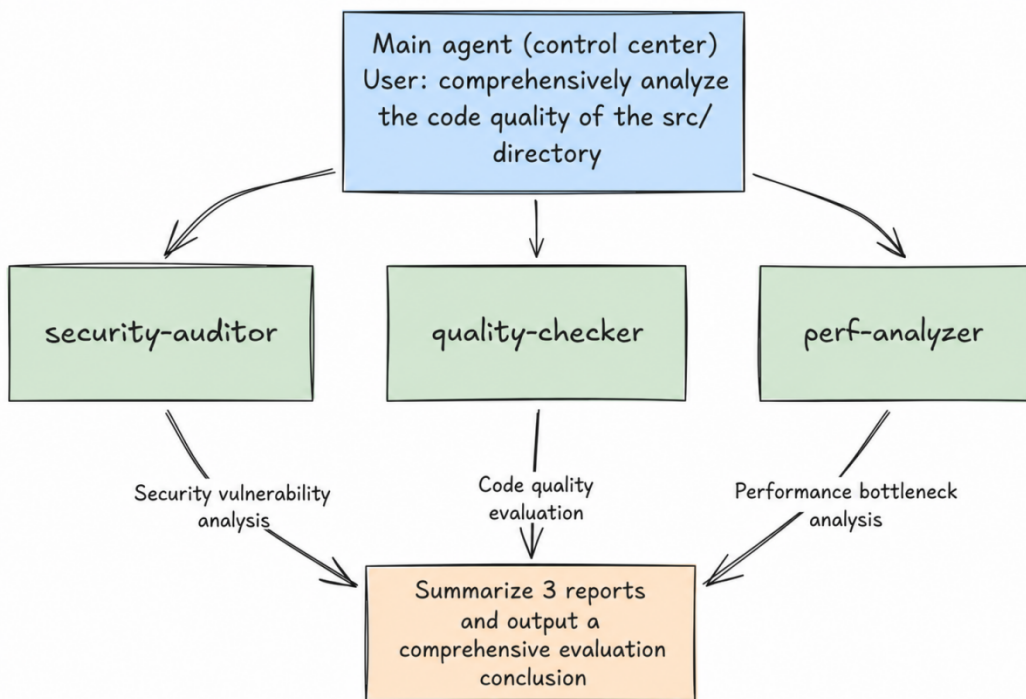


Figure 4.3: Parallel sub-agents—multiple experts working simultaneously

Three sub-agents work synchronously in their own isolated spaces, without interfering with each other, greatly shortening the overall analysis time. In the end, the system aggregates their respective findings to form a comprehensive, multi-perspective report.

The parallel mode has a strict prerequisite: **the subtasks must be completely independent**, with no shared state or dependencies.

In this mode, security auditing does not need to wait for the results of code quality assessment; performance analysis does not depend on the conclusions of security auditing.

These three sub-agents can truly run synchronously in their own isolated spaces, without interfering with each other, thereby maximizing the use of computing resources and shortening the overall time required.

However, once there are logical dependencies in the task chain (for example, "the bug must be located before it can be fixed"), the parallel mode is no longer applicable. If you force parallelism in such cases, subsequent tasks will fail or produce incorrect results due to lack of prerequisite input. For such task flows with clear sequential order, you should use the pipeline mode introduced in the next section to ensure rigorous execution logic.

This model has a well-known counterpart in the field of software engineering—**MapReduce**. MapReduce is a programming model for processing large datasets across distributed systems: the Map phase splits a task into parallel subtasks, and the Reduce phase combines their results. In the context of sub-agent applications, this classic architecture is perfectly mapped. In the Map phase, the main agent acts as the "orchestrator," breaking down a large, complex problem into multiple independent sub-tasks and distributing them to different sub-agents (worker nodes). Each sub-agent processes its portion of the data or logic in an isolated environment, running in parallel. In the Reduce phase, once all sub-agents have completed their computations, the main agent steps in again to collect, integrate, and distill the outputs from each sub-agent, ultimately generating a unified and complete answer.

In this architecture, sub-agents focus solely on their own independent computational logic, without needing to worry about global state or the results of other nodes; the main agent, on the other hand, is dedicated to orchestration—responsible for decomposing tasks and aggregating results.

A major advantage of parallel sub-agents is specialization. The role sketches that follow are illustrative rather than measured: each sub-agent can be given a narrow brief — the security-auditor.

It helps to separate two ideas that are easily conflated. Genuine concurrency comes from the main agent spawning several sub-agents within a single turn, which the harness then runs at the same time. Backgrounding — sending a long-running sub-agent to the background so the main thread is not blocked — is a separate convenience, and the key binding for it varies by platform and Claude Code version, so check your own key bindings rather than trusting a printed shortcut. Note also that sub-agents (spawned in-session through the Agent tool) are a different mechanism from background sessions shown in the agent view.

Pipeline Mode: Serial Processing Chain

The parallel mode is suitable for independent tasks, while the pipeline mode is designed for **tasks with clear stage dependencies** (see Figure 4.4). The classic example is a bug-fix pipeline: first, identify the root cause of the problem; next, implement the fix based on the diagnosis; then, verify the effectiveness of the fix; and finally, analyze the impact of the fix. In this process, the output of each stage serves as the input for the next, and the execution order cannot be reversed.

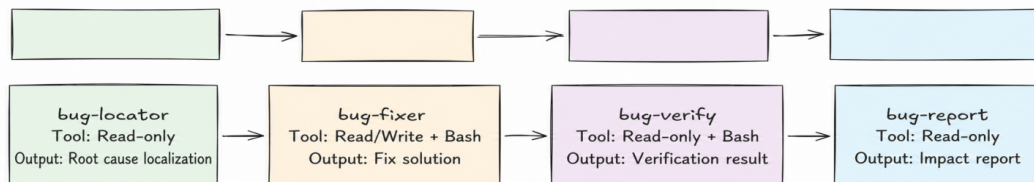


Figure 4.4: Pipeline Sub-Agents—Four-Stage Serial Processing Chain for Bug Fixes

This pipeline has a subtle yet crucial design feature: **each stage has distinctly different tool permissions**. The bug-locator has read-only access, ensuring that nothing is modified during diagnosis; the bug-fixer has read-write and Bash execution permissions, allowing it to edit code and run build commands; bug-verify reverts to "read-only plus Bash" mode, needing only to run tests to verify the fix's effectiveness, with any code changes strictly forbidden; bug-report returns to pure read-only status, as impact analysis only requires reviewing code and change history. This dynamic adjustment of permissions is a security mechanism unique to the pipeline model, ensuring that each stage holds only the minimum permissions necessary to fulfill its responsibilities.

The key technical detail of the pipeline model lies in the **handoff contract**: the output format of each stage must strictly match the input requirements of the next stage.

Below is a complete example definition of a four-stage bug-fix pipeline.

```

<!-- .claude/agents/bug-locator.md -->
---
name: bug-locator
description: Locate the root cause of the bug
tools:
- Read
- Grep
- Glob
permissionMode: plan
---
```

You are a bug localization expert. Your task is to identify the root cause of the bug.

Localization Process

1. Understand the symptoms: Analyze the error messages and reproduction steps.
2. Search relevant code: Use keywords and file patterns to pinpoint suspicious areas.
3. Trace the call chain: Trace upward from the error point to the root cause.
4. Confirm the root cause: Clearly identify the specific line of code causing the issue.

Output Format (Downstream stages depend on this format, please strictly adhere to it)

Root Cause File: [file_path:line_number]
Problem Description: [One-sentence summary of the root cause]
Call Chain: [Complete path from entry point to error location]
Fix Direction: [Brief idea for the fix]
<!-- .claude/agents/bug-fixer.md -->

name: bug-fixer
description: Fix bugs based on the localization results
tools:
- Read
- Grep
- Glob
- Edit
- Write
- Bash

You are a bug-fix expert. You will receive the localization conclusions output by bug-locator,
and implement fixes based on them.

Fix Principles

1. Minimal changes: Only modify the necessary code, avoiding unrelated refactoring.

2. No new issues: Ensure that the fix does not break existing functionality.
3. Consistent style: Follow the project's existing code style.
4. Defensive code: Add necessary safeguards to prevent recurrence of similar issues.

Output Format

Modified files: [file_path_1, file_path_2, ...]
 Reason for each modification: [explain each one individually]
 Potential side effects: [if any, explain in detail]
 Suggested test commands: [specific commands to verify the fix]

It is worth noting that the output format of bug-locator explicitly includes the "root cause file" and "fix direction," which are precisely the key inputs required for bug-fixer to begin its work; meanwhile, the output of bug-fixer covers "modified files" and "suggested test commands," which in turn provide the core basis for verification in the bug-verify phase. This **rigorous format contract between upstream and downstream** forms the cornerstone of reliable pipeline operation.

The Unix pipe embodies the same idea, as in the command `cat log.txt | grep ERROR | sort | uniq -c | sort -rn`,

The actual execution process of the pipeline is as follows:

User instruction: Fix the payment failure issue reported in issue #123

Main agent coordination process:

1. Delegate to bug-locator (independent context)

Task: Analyze root cause

Conclusion: Missing null check at src/payment/charge.ts:89

2. Pass to bug-fixer (independent context)

Input: The above localization conclusion

Task: Implement code fix

Conclusion: Null check logic added at charge.ts:89

3. Pass to bug-verify (independent context)

Input: Fixed files and suggested test commands

Task: Run tests to verify the effectiveness of the fix

Conclusion: All 12 test cases in the payment module passed

4. Pass to bug-report (independent context)

Input: Complete record of the fix and verification

Task: Analyze impact scope and generate report

Conclusion: Scope of fix is controlled, no side effects, recommend adding null boundary tests in the future

Final result: Each of the four sub-agents completed in-depth work in isolated contexts, and the main conversation only received and displayed four concise, structured key conclusions.

Team-based: a self-organizing collaboration mechanism. Note that Agent Teams is experimental and disabled by default — you enable it with the `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS` environment variable — so readers who try the mode without enabling it will find it unavailable.

The team-based model is the most complex and powerful architecture among sub-agent patterns. It is fundamentally different from the four modes introduced earlier: in the first four modes, sub-agents are only active during task execution and terminate upon completion, exhibiting a "one-off" characteristic; in the team-based model, sub-agents have **long-term persistence**. Like a real team, they can collaborate continuously, communicate in real time, and autonomously divide work.

An Agent Team consists of three core elements (see *Figure 4.5*).

- **Team Lead:** Responsible for task decomposition, resource scheduling, and progress tracking, similar to a project manager.
- **Teammates:** Composed of sub-agents with different areas of expertise, each responsible for specific functional modules.
- **Shared infrastructure:** Includes two key mechanisms. The Task List ensures all members can keep track of overall progress and their own to-dos in real time; the Mailbox supports asynchronous communication between members (for example, a frontend developer can use this mechanism to confirm API specifications with a backend developer).

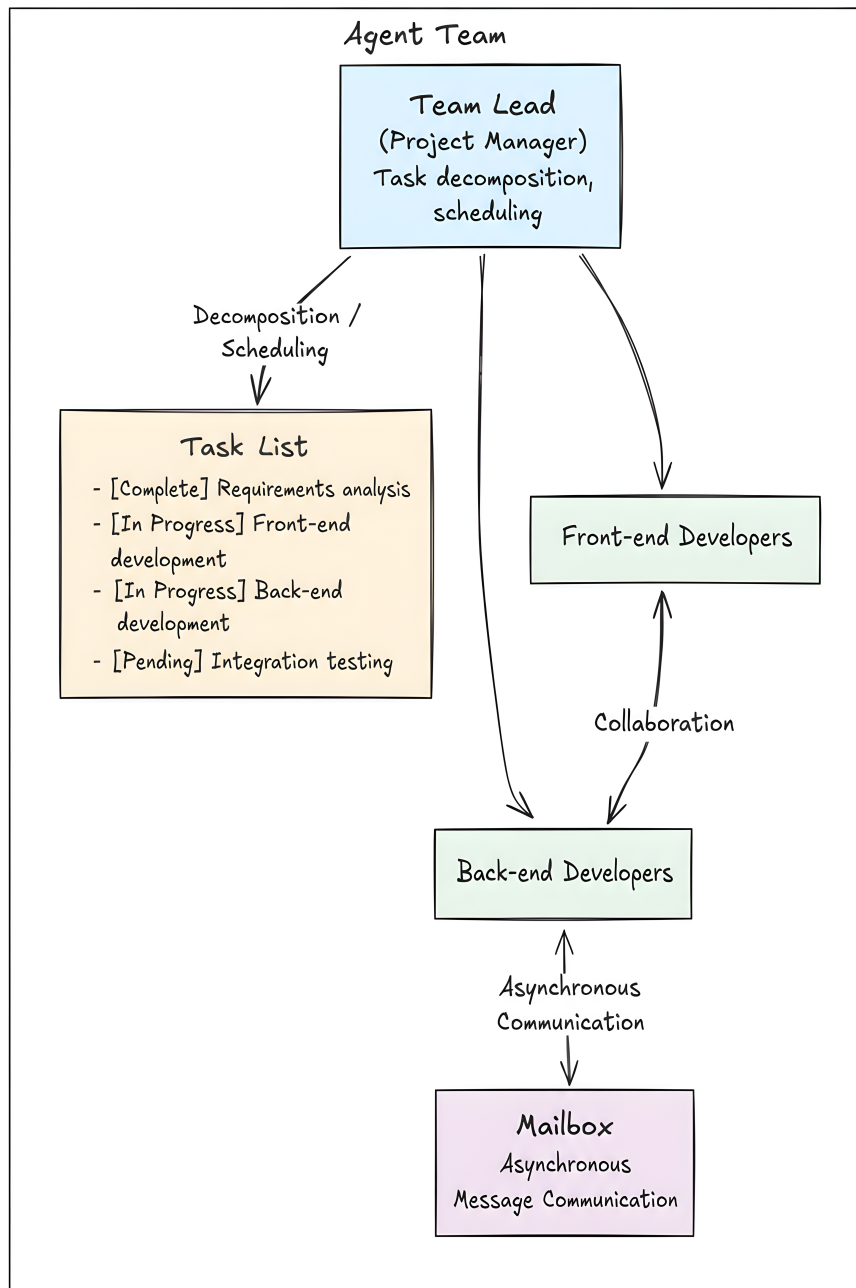


Figure 4.5: Team-based sub-agents—multi-session self-organizing collaboration

The team-based sub-agent model is suitable for four typical collaboration patterns.

- **Competing hypothesis mode:** When facing complex problems with uncertain causes (e.g., whether a performance bottleneck is due to database queries or network IO), multiple sub-agents can be dispatched to investigate in parallel based on different hypotheses. Ultimately, the Team Lead compares the investigation reports and selects the conclusion with the strongest evidence.
- **Parallel review mode:** For the same code change, both security and functional reviews can be conducted simultaneously. The two reviewers work independently without interference, and their opinions are merged at the end. This mode is not only more efficient than serial review, but the independence of multiple perspectives also significantly improves review quality.
- **Module ownership mode:** In large-scale refactoring tasks, the codebase is divided by module and assigned to different team members. For example, the frontend sub-agent is responsible for UI component refactoring, the backend sub-agent for API layer refactoring, and the data layer sub-agent for database migration. Each sub-agent focuses deeply on its assigned module and coordinates cross-module interface changes via the Mailbox mechanism.
- **Proposal approval mode:** One sub-agent proposes a refactoring plan, while another acts as a "devil's advocate," specifically raising challenges and objections. The Team Lead makes the final decision based on both sides' arguments. This adversarial design effectively avoids blind spots in the plan.

When should you use the team-based model instead of a simple sub-agent model? A practical criterion is the complexity of collaboration: if **sub-agents require multiple rounds of communication and deep coordination**, the team-based model should be chosen; if each sub-agent only needs to perform a single task to completion, simple parallel or pipeline modes are sufficient.

It is important to note that the team-based model incurs higher operational overhead—maintaining a long-term context window means continuous token consumption. Therefore, this mode is only cost-effective in scenarios that truly require complex collaboration.

Note**Kage's notes**

Nesting beyond two layers makes a system markedly harder to reason about and to debug. Read this as a rule of thumb rather than an enforced ceiling: Claude Code permits a sub-agent to spawn its own sub-agents, and the practical limit is a matter of the harness version rather than a number worth memorizing. If you find yourself reaching a third layer, treat it as a signal that the decomposition needs rework.

For most daily development scenarios, using only the read-only and execution modes is usually sufficient. Be careful not to overengineer.

Collaboration between Sub-agents and Skills

In the five modes introduced earlier, sub-agents demonstrate excellent task decomposition and isolation capabilities, but they avoid a core issue: where does a sub-agent's "expert knowledge" actually come from?

Take the bug-fixer in the pipeline mode as an example. It is tasked with fixing a null check bug and is equipped with a full set of tools such as Read, Edit, Write, and Bash, but it knows nothing about the project's coding standards—should error handling use the Result pattern or try-catch? Should logging call `console.log` or the project's dedicated logger? Should variable names follow camelCase or snake_case?

If these standards are written into the project's `CLAUDE.md` file, **sub-agents** can indeed inherit them automatically. However, `CLAUDE.md` usually only covers general standards and is unlikely to include deep domain knowledge such as a "secure coding checklist for bug fixes"—the latter clearly falls under the category of Skills.

There are only two atomic modes for combining Skills and sub-agents with **the core difference being the inclusion relationship** (see *Figure 4.6*).

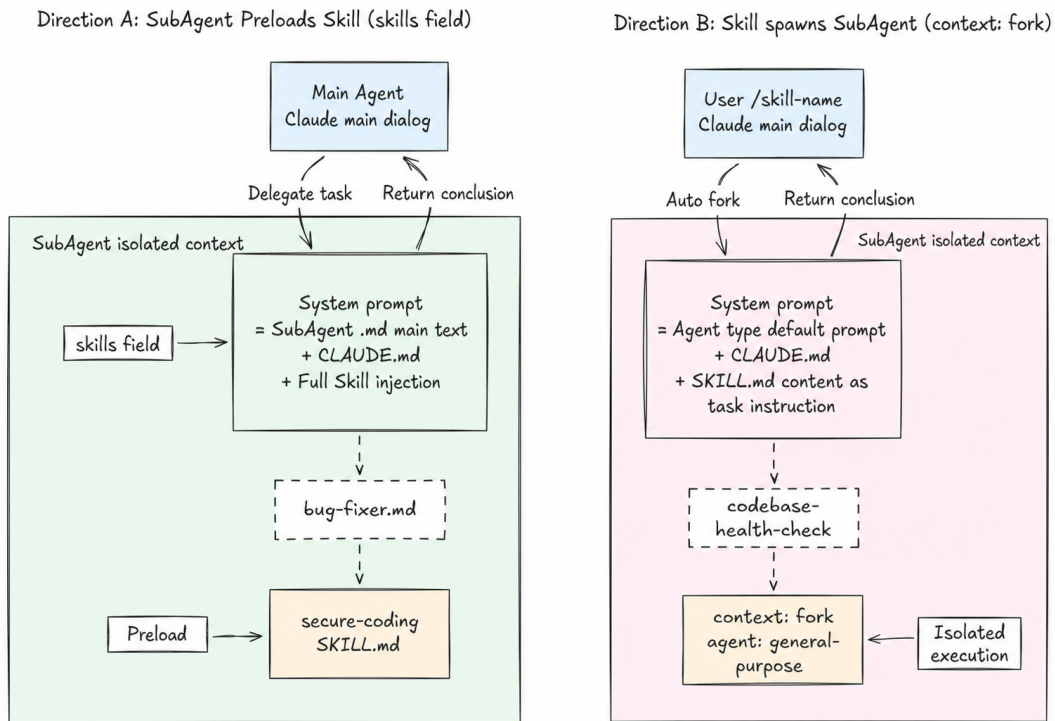


Figure 4.6: Two collaboration directions between Skills and sub-agents

Direction A: Sub-agent preloads Skill (sub-agent contains Skill)

In this mode, the sub-agent's definition file preloads one or more Skills as a domain knowledge base via the skills field at startup.

- **Role positioning:** The sub-agent is the executor, while the Skill is its manual.
- **Applicable scenarios:** Roles in the pipeline that require specific domain knowledge. For example, bug-fixer needs to load secure coding standards, and doc-generator needs to load documentation generation processes.

Direction B: Skill spawns sub-agent (Skill contains sub-agent)

In this mode, the Skill automatically creates an isolated sub-agent to execute the task via the context: fork configuration when triggered, ensuring that the intermediate process does not pollute the main conversation context.

- **Role positioning:** The Skill is the dispatcher, and the sub-agent is the execution instance
- **Applicable scenarios:** heavy-duty tasks that require strict isolation and one-time completion, such as deep code analysis and batch document generation

The core difference between these two modes lies in the ownership of **System Prompt** control.

- **Direction A:** The definition file (.md main text) of the sub-agent constitutes the System Prompt, while the full content of the Skill is injected as domain knowledge context
- **Direction B:** The agent type specified by the agent field provides the System Prompt, while the content of SKILL.md is passed in as the specific task instruction

Although the underlying mechanisms of both are completely identical, their invocation relationships are fundamentally opposite.

Next, we will delve into Direction A through two examples. The first is an improved version of the bug fixer.

```
<!-- .claude/agents/bug-fixer.md -->
---
name: bug-fixer
description: Fixes bugs based on localization results, adhering to the team's
secure coding standards.
tools:
- Read
- Grep
- Glob
- Edit
- Write
- Bash
skills:
- secure-coding          # Preload secure coding Skill
---
```

```
You are a bug-fixing expert. You will receive localization conclusions from the
bug-locator,
and perform fixes accordingly.
```

During the fixing process, you must strictly adhere to the secure coding checklist defined in the secure-coding Skill.

Pay special attention to the following key points: null value checks, input validation, and error handling patterns.

Fixing Principles

1. Minimal changes: Only modify the necessary code, avoiding unrelated refactoring.
2. Zero side effects: Ensure that the fix does not break existing functionality.
3. Standards compliance: Strictly follow the secure coding standards defined in the Skill.
4. Defensive programming: Add defensive code to prevent recurrence of similar issues.

Output Format

Modified files: [file_path_1, file_path_2, ...]

Reasons for modification: [Explain the reason for each change one by one]

Security checklist items: [List the checklist items from the Skill that have been confirmed as passed]

Suggested test commands: [Specific commands to verify the fix]

The corresponding Skill definition is as follows:

```
<!-- .claude/skills/secure-coding/SKILL.md -->
---
```

```
name: secure-coding
```

```
description: Secure coding checklist for bug fixes and new features.
```

```
Use when fixing bugs, writing new code, or reviewing security aspects.
```

```
user-invocable: false
```

```
---
```

Secure Coding Standards

Null Value Defense

- Input Validation: All external inputs must undergo non-null checks.
- Safe Access: Prefer using the optional chaining operator (?.) to access nested properties; the use of non-null assertions (!) is strictly prohibited.
- Data Null Check: Database query results must be explicitly checked for null before being used in business logic.

Error Handling

- Unified Exceptions: Business logic exceptions must use the project's unified `AppError` class. Throwing native exceptions or strings directly is prohibited.
- No Silent Failures: Empty catch blocks are strictly forbidden. After catching an exception, at a minimum, an error log must be recorded or a retry mechanism implemented.
- Timeout Control: All asynchronous operations must have a clearly defined timeout period.

Logging Standards

- Dedicated Interface: The use of `console.log()` is strictly prohibited in production environments. The project's encapsulated logger object (such as `logger.error()`) must be used.
- Complete Information: Error logs must include a unique error code and a complete stack trace.
- Data Masking: It is strictly forbidden to print any sensitive information in logs (including but not limited to passwords and API tokens).

It is important to note that `user-invocable: false` means this Skill does not need to appear in the user's "/" menu. It is purely a "reference manual" type of content and will be automatically preloaded when a sub-agent is launched.

Let's look at another cross-domain example—an API documentation generator sub-agent, which enhances its professional capabilities by preloading the "documentation generation Skill."

```
<!-- .claude/agents/api-doc-generator.md -->
---
name: api-doc-generator
description: Generate API documentation by scanning route files.
tools:
- Read
- Grep
- Glob
- Write
- Bash
skills:
- api-generating      ← Preload Skill to inject domain knowledge
---
```

Among them, `skills: [api-generating]` is essentially handing an "operation manual" to the subagent. With a clearly defined role (API documentation specialist) and the professional workflow embedded in the Skill (including specific steps, templates, and scripts), the subagent transforms from a jack-of-all-trades generalist agent into a domain expert with specialized skills.

These two examples—`bug-fixer` paired with `secure-coding`, and `api-doc-generator` paired with `api-generating`—reveal a clear principle of responsibility division: the Subagent defines "identity and objectives," while the Skill defines "methods and standards."

The specific division of labor is as follows:

The **Subagent** (configured in the `.md` file) is responsible for the strategic level.

Who (Identity): Establishes the role, such as "You are a vulnerability remediation expert" or "You are an API documentation specialist."

What (Task): Clarifies the core objective, such as "fix vulnerabilities" or "generate API documentation."

- **Where (Scope):** Defines the boundaries of work, such as "write to the `docs/api/` directory"
- **Output (Deliverable):** Specifies the output format, such as "return a summary containing remediation details" or "count the number of routes"
- The Skill (`SKILL.md` and associated files) is responsible for tactical execution
- **How (Process):** Details the operational steps, such as "Step 1: Run `detect-routes.py` → Step 2: Analyze the results"

With What (Tools): Specifies the required resources, such as the script `scripts/detect-routes.py` or the template `templates/api-doc.md`.

- **By What Standard (Norms):** Sets the execution criteria, such as "check for authentication middleware and mark as locked"
- **Quality (Quality):** Defines the acceptance standards, such as "all routes are archived, and the Schema strictly matches the code"

This separation mechanism endows the system with powerful reusability. The same `secure-coding` Skill can be preloaded by the `bug-fixer` to guide vulnerability remediation, or invoked by the `code-reviewer` to audit code compliance; similarly, the `api-generating` Skill can assist documentation-generating subagents in creating new documents, or empower API

review subagents to verify the compliance of existing documentation. In short, the Skill is a reusable knowledge module, while subagents are flexible consumers of this knowledge.

```
<!-- .claude/skills/codebase-health-check/SKILL.md -->
---
name: codebase-health-check
description: Perform comprehensive code health analysis
context: fork
agent: general-purpose
allowed-tools:
- Read
- Grep
- Glob
---
```

When the user enters `/codebase-health-check src/`, the system automatically spawns a subagent in an isolated context. This subagent can independently scan all files, analyze issues, and generate a report, while the main conversation only receives a concise final result. The large volume of file content and reasoning generated in the process does not pollute the main context at all.

Approach A and Approach B each have their own applicable scenarios, and the judgment criteria are very straightforward: who is the main actor?

Approach A: Role-driven (the subagent is the main actor).

- **Core logic:** The key to the task lies in the specific role identity and collaboration process.
- **Scenario example:** In a pipeline, a bug-fixer needs to continuously apply expertise to fix vulnerabilities. Here, the subagent is a long-term "expert," and the Skill is just a "wrench" in its toolbox.
- **Conclusion:** When you need to maintain role state, conduct multi-turn interactions, or engage in complex collaboration, choose Approach A.

Approach B: Knowledge/process-driven (the Skill is the main actor).

- **Core logic:** The key to the task is executing a standardized knowledge process, and the executor itself is not important, or even needs to be isolated.

- **Scenario example:** codebase-health-check is a standardized inspection process. Here, the Skill defines "how to do it," and the subagent is merely a temporary "execution container" created to isolate the context.
- **Conclusion:** When the task is one-off, requires strict context isolation, or the focus is on reusing standardized processes, choose Approach B.

Token Economics

"Kage, will subagents increase API call costs?" Xiaobing asked a very practical question.

The answer to this question may be surprising: in most scenarios, reasonable use of subagents not only does not increase costs, but can significantly reduce total token consumption.

The underlying "token economics" principle is worth a deep dive.

First, let's analyze the situation without subagents. Suppose the main Claude directly executes a testing task, producing 10,000 tokens of test output. Since Claude's API is typically stateless, each call requires transmitting the complete context history. Therefore, once these 10,000 tokens of test content enter the main conversation context, they will be repeatedly sent in every subsequent round of interaction. If there are five more rounds of conversation, this "noise" data from the test will be redundantly transmitted five additional times.

The cost calculation without subagents is as follows:

Assumptions:

- **System Prompt:** 5,000 tokens
- **Existing conversation history:** 8,000 tokens
- **Test output (noise):** 10,000 tokens

Each of the following 5 rounds adds about 1,000 tokens per round.

Token consumption details per round:

- Round 5: 5,000 (system) + 8,000 (history) + 10,000 (test) = 23,000
- Round 6: 5,000 (system) + 9,000 (history + new) + 10,000 (test) = 24,000
- Round 7: 5,000 (system) + 10,000 (history + new) + 10,000 (test) = 25,000
- Round 8: 5,000 (system) + 11,000 (history + new) + 10,000 (test) = 26,000
- Round 9: 5,000 (system) + 12,000 (history + new) + 10,000 (test) = 27,000

Total consumption for rounds 5–9: 125,000 tokens

Now let's look at the scenario with subagents. In this mode, the test output is strictly isolated within the subagent's independent context, and the main conversation only receives a test summary of about 100 tokens.

Cost calculation with subagents is as follows:

Within the subagent (independent context):

- Round 1: 5,000 (system) + 500 (task description) = 5,500
- Round 2: 5,000 (system) + 500 (task) + 3,000 (partial output) = 8,500
- Round 3: 5,000 (system) + 500 (task) + 6,000 (cumulative output) = 11,500
- Round 4: 5,000 (system) + 500 (task) + 10,000 (full output) = 15,500

Total subagent consumption: 41,000 tokens

Main conversation (only receives 100-token summary):

- Round 5: 5,000 (system) + 8,000 (history) + 100 (summary) = 13,100
- Round 6: 5,000 (system) + 9,000 (history + new) + 100 (summary) = 14,100
- Round 7: 5,000 (system) + 10,000 (history + new) + 100 (summary) = 15,100
- Round 8: 5,000 (system) + 11,000 (history + new) + 100 (summary) = 16,100
- Round 9: 5,000 (system) + 12,000 (history + new) + 100 (summary) = 17,100

Total main conversation consumption: 75,500 tokens

Overall cost statistics:

Total consumption: 41,000 (subagent) + 75,500 (main conversation) = 116,500

Tokens saved: $125,000 - 116,500 = 8,500$

Savings ratio: about 6.8%.

Trend analysis: As the number of subsequent conversation rounds increases, the token savings ratio will rise significantly due to the avoidance of repeated transmission of large amounts of noise data.

In this example, just five subsequent rounds of conversation yield a 6.8% cost saving. If the conversation extends to 10 or 20 rounds, the savings ratio will increase even further. In certain extreme scenarios (such as continuing with multiple rounds of in-depth discussion after massive log analysis), the token savings rate can even exceed 50%.

Of course, it must be pointed out that Anthropic's Prompt Caching mechanism will somewhat reduce the above cost advantage. This mechanism charges only 10% of the standard price for

tokens that hit the cache, meaning that the test output data repeatedly transmitted in the main conversation will incur much lower actual costs when cached. After recalculating with caching factored in, the cost savings in this example drop from 6.8% to 1%–2%.

However, cost optimization is only one dimension of the value of subagents, and not even the core value. There are two other key advantages that the prompt caching mechanism cannot replace.

Context window protection. Large models have limited context window capacity. When a large amount of noise data occupies storage space, Claude may trigger context compaction early. In this process, critical information is easily lost if deemed "low priority." By strictly isolating noise in an independent context, subagents effectively preserve valuable context space for the main conversation, ensuring the integrity of core information.

- **Improved response quality.** A clean context environment can significantly enhance Claude's focus. When the conversation history is filled with irrelevant log records and test outputs, Claude's "attention mechanism" is scattered, leading to a decline in generation quality—this was exactly the typical dilemma Xiaobing initially faced. The subagent's isolation architecture fundamentally solves this "attention dilution" problem, ensuring the main model always reasons and responds based on high-quality, highly relevant information.
- So, in what scenarios does introducing a subagent become counterproductive? Here is a concise decision rule: examine the input-to-output volume ratio.

High-value scenarios (input >> output): When the input data volume far exceeds the final output, the value of subagents is especially significant. For example, analyzing hundreds of lines of system logs just to extract five key conclusions.

Low-value scenarios (input $\approx$ output): When the input and output volumes are roughly equal, the isolation overhead of subagents is no longer worthwhile. For example, modifying a single function or writing a code comment, where the amount of intermediate data generated is limited. In such cases, completing the task directly in the main conversation is often simpler and more efficient, without incurring extra architectural complexity and communication costs.

Note

Kage's notes

The decision to enable subagents can be summarized in the following four core dimensions.

- **Large-scale file reading (input noise isolation):** When a task requires reading a large number of files (usually more than five) or processing massive data

sources, using subagents can completely isolate the "input noise" generated during file reading in an independent environment, preventing the main conversation context from being quickly filled with redundant file content.

- **High-frequency output generation (output noise filtering):** When a task is expected to produce a large volume of intermediate outputs, such as comprehensive test reports, detailed system logs, or complete build process records, subagents act as "filters," returning only concise summaries or key error information to the main conversation. This prevents lengthy output data from diluting the main model's attention and wasting tokens.
- **Context integrity protection (long-term task planning):** When the current task is just one part of a complex workflow, it is essential to ensure that the main conversation retains sufficient context space to support subsequent rounds of deep interaction or reasoning. By offloading high-consumption subtasks, the subagent mechanism effectively protects the "context window" of the main conversation from being exhausted at once, ensuring the coherence and completeness of information for long-term tasks.
- **Operational permissions and security boundaries (risk management):** When tasks involve sensitive operations (such as file system writes, network requests, or executing external commands) or require strict permission control, subagents can be configured as independent sandbox environments with restricted permissions. Even if a subtask fails or exhibits malicious behavior, its impact is confined to a local scope, providing the main system with a natural security firewall.

Subagents from a Software Engineering Perspective

By this point, astute readers may have realized: the subagent architecture does not invent any fundamentally disruptive new concepts out of thin air. Its core essence lies in creatively transferring and mapping the classic wisdom accumulated in software engineering over decades into the architectural design of AI Agents.

This migration is by no means accidental, but rather a logical evolution. Because an AI Agent is, at its core, a complex software system, it inevitably follows the general laws and core principles of software systems engineering.

Single Responsibility Principle

The **Single Responsibility Principle** stands at the forefront of the five major object-oriented design principles. Its core definition is striking: a class (or module) should have only one reason to change—in other words, it should focus on doing one thing, and doing it well. The subagent architecture is a perfect embodiment and implementation of this classic principle in the AI Agent domain.

- **Specialization of responsibilities:** The code-reviewer focuses solely on code quality review; the test-runner is only responsible for executing test cases and collecting results; the bug-locator is dedicated to pinpointing the root cause of errors. Each subagent is like a highly cohesive, independent class, each performing its own duties without overstepping boundaries.
- **Declarative responsibility specification:** Each subagent's prompt file (such as `code-reviewer.md`) essentially serves as its "responsibility specification".
- **Change isolation and maintainability:** For example, if you need to adjust the strictness of code review or introduce new security standards, developers only need to modify the `code-reviewer.md` file

Bulkhead Pattern

The **Bulkhead Pattern** is a resilience design pattern borrowed from shipbuilding: the hull of a cargo ship is divided into multiple independent watertight compartments by sturdy partitions. Even if one compartment is breached, the seawater is confined locally and will not sink the entire vessel.

In microservices architecture, this pattern is widely adopted to prevent single points of failure from triggering cascading collapses. In AI Agent architecture, the context isolation mechanism of subagents is a perfect mapping of the bulkhead pattern.

- **Fault isolation and boundary locking:** If the log-analyzer subagent encounters abnormal data (such as infinite loops, hallucinations, or format crashes) while processing massive logs, the resulting "fault" is strictly contained within the subagent's independent sandbox
- **Main system stability assurance:** Thanks to the bulkhead, the chaos of a subtask does not pollute the context environment of the main conversation
- **Graceful degradation:** This design ensures system resilience

MapReduce Pattern

The **MapReduce pattern** originates from the field of distributed computing and was first introduced by Google in a 2004 paper. This pattern decomposes large-scale data processing

into two stages: **Map** (mapping) and **Reduce** (reduction). The Map stage distributes tasks to multiple worker nodes for parallel processing, while the Reduce stage aggregates the results from all nodes.

The parallel subagent architecture is an innovative application of the MapReduce pattern in the AI Agent domain. In this architecture, the main agent acts as the coordinator, responsible for task distribution (Map) and result aggregation (Reduce), while each subagent serves as a worker node, focusing on executing parallel Map computation tasks.

Chain of Responsibility Pattern

The **Chain of Responsibility pattern**, one of the **Gang of Four (GoF)** design patterns, defines a processing chain along which requests are passed until they are received and handled by a processor. The pipeline-style subagent architecture is a variant of this pattern: unlike the traditional chain where a request is terminated by "one" processor, a pipeline requires "every" processor to handle part of the request and pass the intermediate result to the next stage. Unix pipelines, Java Servlet filter chains, and Node.js Express middleware are all concrete embodiments of this idea in different tech stacks, and the subagent pipeline is the latest member of this family.

These correspondences are not forced analogies, but rather reveal a deeper law: **excellent architectural principles have cross-domain universality**. Whether designing microservice systems, distributed computing frameworks, or building AI Agent architectures, developers always face the same fundamental challenges—how to decompose complexity, how to isolate faults, and how to achieve efficient collaboration. Engineers well-versed in classic design patterns have a natural intuitive advantage when venturing into AI Agent design, because they have repeatedly validated the effectiveness of these ideas in other domains.

After listening, Xiaoxue nodded thoughtfully and said, *"So learning design patterns isn't just for passing interviews—these ideas are even more useful in the AI era."*

Kage nodded and said, *"The core wisdom of software engineering has never gone out of style; it simply finds new forms of expression with each new wave of technology."*

Practical Considerations

There are several easily overlooked details worth understanding in advance when using subagents in practice, as they often determine whether subagents can operate stably in a production environment.

First is the inheritance relationship of `CLAUDE.md`. The `CLAUDE.md` file in the project root directory applies to sub-agents — a sub-agent loads this "project memory" on startup, along

with the project's MCP servers and Skills. This means that any coding standard you write there is inherited by every sub-agent you spawn, without repeating it in each definition.

Next is the **"message passing" mode of context transmission**. Subagents cannot communicate directly with each other: each subagent can only access content explicitly passed by the main agent, has no right to access the main conversation's history, and cannot perceive the existence of other subagents. This information exchange mechanism is "message passing" rather than "shared memory"—the main agent must extract the conclusion from subagent A and embed it into the task description for subagent B, so that B can obtain the relevant information. A deep understanding of this mechanism is crucial for designing efficient pipelines: be sure that the output format of each stage is highly structured, so the main agent can accurately extract and smoothly forward the information.

Then there is the interruption recovery mechanism. If a subagent is interrupted during execution due to network fluctuations or manual termination, any intermediate work stored in memory will be lost. For long-running or critical tasks, an effective strategy is to have the subagent persist intermediate artifacts to a file. For example, a log analysis subagent can record the analyzed range and discovered anomalies in real time to a `.claude-work/log-analysis.md` file. When the task needs to be restarted, the new subagent can first read this file, precisely locate the breakpoint, and resume work, thus avoiding redundant work from starting over.

keep nesting to about two layers as a matter of maintainability, not because the system forbids more

Summary

Sub-agents are the most architecturally significant feature in the Claude Code extension system. If `CLAUDE.md` grants Claude the ability to remember, and Skills endow it with domain expertise, then sub-agents provide a more fundamental capability: **the ability to decompose complexity**. When faced with complex tasks that span multiple dimensions, are accompanied by significant noise, and require multi-stage processing, sub-agents offer a structured response strategy: break down the overall task into several clearly focused sub-tasks, execute them independently within isolated contexts, and finally aggregate only the distilled core conclusions back into the main conversation flow.

The value of this strategy is not only reflected at the technical level (such as context isolation, permission constraints, and optimization of token utilization efficiency), but more profoundly in the way of thinking it fosters. Learning to use sub-agents is essentially mastering a form of "delegated thinking": no longer cramming every detail into a single conversation, but instead, like a seasoned manager, clearly defining the boundaries of responsibility, capability

requirements, and delivery standards for each task, and delegating them to the appropriate executors. This mindset is closely aligned with modular design, microservices architecture, and the separation of concerns principle in software engineering—they all address an eternal challenge: how to effectively manage complexity.

The five design patterns (read-only, execution, parallel, pipeline, and team) are not isolated techniques, but rather a progressive spectrum of capabilities. From the most basic read-only observation to the most advanced complex multi-conversation self-organizing collaboration, they comprehensively cover scenarios ranging from daily development tasks to large-scale engineering projects. Once you have mastered these five design patterns and the underlying software engineering principles, you will possess the intuition and capability to construct adaptive sub-agent architectures for any new scenario.

In *Chapter 5*, we will delve into another core extension mechanism—event-driven automation based on **Hooks**. If Skills and sub-agents primarily operate at Claude's "cognitive level," then Hooks extend its control to the "system execution level": they no longer merely guide Claude on "how to think," but directly intercept and constrain its behavior at the level of physical execution.

Discussion Questions

1. Suppose your team needs to build an automated PR review process that requires simultaneous execution of code quality checks, security vulnerability scans, and performance analysis, with the results ultimately consolidated into a comprehensive review report. Based on this, design a sub-agent architecture: how many sub-agents need to be deployed? Which design pattern should be adopted (parallel processing or pipeline operation)? How should the tool permissions for each sub-agent be configured?
2. In the *Token economics* section, we discussed under the assumption of "five subsequent rounds of conversation." Now consider further: if there is no follow-up interaction after task execution (i.e., it is a one-off execution), does the sub-agent architecture still offer a cost advantage? Under what specific conditions might introducing sub-agents actually cause a surge in token consumption, thereby increasing overall costs?
3. Looking back on this chapter, we have explored in depth the five sub-agent design patterns and analyzed the core software engineering principles underlying them. Now, turn your attention to your actual projects: is there a pain point scenario that is particularly well-suited to be addressed by introducing a sub-agent architecture? If so, which design pattern does this scenario align with? Please explain, in the context of your specific business logic, the deeper reasons for choosing this pattern.

5

From Guidelines to Guardrails: Automating Control with Hooks

"By failing to prepare, you are preparing to fail." — Benjamin Franklin

An incident occurred at the company.

It was late at night. Xiao Zhang was handling an urgent requirement. He wrote code using Claude Code, and after passing local tests, he executed a git push to upload the code to the remote repository, then shut down his computer and went home to rest.

At nine o'clock the next morning, the team chat exploded: *"Who pushed the .env file? The API key has been exposed. ALL keys must be immediately revoked and reissued!"*

Xiao Zhang was well aware that .env files must never be committed, but in his fatigue from working late, he neglected to check before submitting. Yet the mistakenly committed .env file contained database connection strings for three environments, an internal API key, and a test account password for the payment gateway. This mistake cost the entire team half a day to rotate keys, audit access logs, and fix all references.

As his direct supervisor, Xiaobing's first reaction was: *"This kind of error could have been automatically intercepted; why is there no mechanism to prevent it?"*

"Because all previous mechanisms only 'suggested' what Claude should do," Kage explained. "CLAUDE.md is a suggestion, Skills are guidance, but no one ever told Claude, 'you absolutely must not do this.' Hooks are different: they are not suggestions, but enforcement."

Xiao Xue immediately chimed in: *"Like middleware in web development? A check before the request goes through?"*

"Exactly." Kage nodded. *"Whether it's Express middleware chains, Django's request processing layers, or Spring's interceptors, their core logic is the same: inserting additional processing logic before and after an operation is executed. Hooks are precisely the realization of this concept in the AI Agent context."*

The Role of Hooks in the Claude Extension System

Before diving into technical details, let's first position Hooks within the extension system we've learned so far.

The three mechanisms introduced in the first four chapters form a spectrum of "increasing influence": `CLAUDE.md` establishes project norms, which Claude reads at the start of each conversation; Skills encapsulate domain workflows, which can be triggered automatically or manually when Claude needs them; sub-agents achieve task delegation through context isolation.

These three mechanisms share a common feature: they all operate at **Claude's** cognitive level. `CLAUDE.md` tells Claude "what norms to follow," Skills guide it on "how to handle certain problems," and sub-agents instruct it "to whom sub-tasks should be delegated." However, these are essentially still advisory guidance. As a language model, Claude is theoretically capable of ignoring any constraints in the prompt.

Hooks operate on a completely different level. They do not act on Claude's cognition, but directly intercept its behavior at the **system execution layer**.

When Claude attempts to call the Bash tool to execute `rm -rf /`, the `PreToolUse` Hook does not "persuade" Claude to abandon the operation; instead, it directly blocks the execution of that tool call at the system level, meaning Claude's decision is forcibly overridden and cannot be carried out.

This distinction, in software engineering terms, is the separation of policy and mechanism. `CLAUDE.md` and Skills define policy — what should be done. Hooks provide the mechanism: when the policy is violated, the command itself is blocked before it runs. Note what is being stopped here. It is not the prompt, and not the task the user is trying to achieve; it is the individual tool call. Just as file-permission management in an operating system does not rely on applications policing themselves, a Hook's constraint on Claude does not rely on the guidance in a prompt.

Table 5.1 compares the differences among `CLAUDE.md`, Skills, and Hooks across four dimensions.

| Cognitive layer | CLAUDE.md | Skills | Hooks |
|--|----------------------------|---------------------|----------------|
| Cognitive layer | Execution layer | Trigger method | Always loaded |
| Semantic matching/
explicit trigger | Event auto-
trigger | Constraint nature | Suggestion |
| Guidance | Mandatory | Control over Claude | Guide behavior |
| Govern workflow | Intercept/block | Analogy | Traffic sign |
| Driver's manual | Roadblock/speed
limiter | Variable | Scope |

Table 5.1: Differences among CLAUDE.md, Skills, and Hooks

It's like this: traffic signs (CLAUDE .md) tell you "the speed limit here is 60 km/h," the driver's manual (Skills) teaches you "slow down on curves," while roadblocks/speed limiters (Hooks) physically prevent your car from reaching 120 km/h. The best results come from using all three together: traffic signs make you aware of the rules, the manual teaches you how to operate, and roadblocks/speed limiters ensure that even if you forget the rules or make a judgment error, the system can still uphold the safety baseline.

Event Lifecycle: More Than Two Dozen Events

The Claude Code event system has grown from 7 events in the early releases to more than two dozen, and it keeps growing with almost every release. Rather than pin a number that will be stale by the time you read this, check the official hooks reference for the live list. What matters is the classification: session-level events, tool-invocation events, sub-agent events, completion events, and a long tail of newer ones (Setup, PermissionDenied, PostToolBatch, PreCompact, TaskCreated, FileChanged, Elicitation, and others). You do not need to memorize them; you need the grouping logic and the characteristics of each group.

Session-Level Events

Session-level events manage the lifecycle of the entire conversation, mainly comprising the following three key nodes.

SessionStart

Triggered when a session starts or resumes. Its core capability is injecting environment variables via the CLAUDE_ENV_FILE. Hook scripts can write export statements to this file,

making the variables effective for all subsequent Bash commands. This means you can automatically configure the development environment at the start of a session. See the following code example.

```
#!/bin/bash
# session-init.sh - SessionStart Hook
if [ -n "$CLAUDE_ENV_FILE" ]; then
    echo 'export NODE_ENV=development' >> "$CLAUDE_ENV_FILE"
    echo 'export DEBUG_LOG=true' >> "$CLAUDE_ENV_FILE"
fi
exit 0
```

SessionEnd

Triggered when a session ends. Its matcher can distinguish between different termination reasons: `clear` (user clears), `logout` (logs out), or `prompt_input_exit` (user exits input). This event is commonly used to clean up temporary resources or record session statistics.

PreCompact

Triggered before context compression. Its matcher can distinguish between manual (user-initiated / `compact`) and auto (automatic compression when the context window is full). This event is suitable for backing up the complete conversation record before compression.

Tool Invocation Events

This is the most critical category of events, covering the entire lifecycle of each tool invocation by Claude. Tool invocation events mainly include the following five key nodes.

PreToolUse (the most powerful event in the entire Hooks system)

Triggered after Claude decides to invoke a tool but before the tool actually runs. Its `permissionDecision` can be `allow` (skip the permission check and run), `deny` (block the call and give a reason), `ask` (hand the decision to the user), or `defer` (leave it to the normal permission flow). Argument rewriting is a separate mechanism, not a fourth decision: put an `updatedInput` object under `hookSpecificOutput` and the tool runs with the parameters you supply.

The ability to "adjust input parameters" is especially powerful: it allows you to silently add security parameters to commands without interrupting the operation. See the following code example.

```
{
  "hookSpecificOutput": {
```

```
"hookEventName": "PreToolUse",
"permissionDecision": "allow",
"updatedInput": {
  "command": "rm -rf /tmp/test --dry-run"
}
}
```

In the above example, the originally dangerous command `rm -rf /tmp/test` is silently modified to `rm -rf /tmp/test --dry-run`, allowing Claude to continue completing the task while avoiding the risk of actually deleting files.

PostToolUse

Triggered after the tool has run. It cannot undo what already happened, but it has two core uses: feeding extra information back to Claude through `additionalContext` (lint results, for instance), and post-processing the result (auto-formatting a file that was just written). It can also replace a tool's output outright by returning an `updatedToolOutput` field. That applies to any tool, not just MCP tools.

PostToolUseFailure

Triggered after the tool fails to execute, mainly used for error alerts and providing corrective feedback.

PermissionRequest

Triggered when a permission dialog is about to appear. The key difference from `PreToolUse` lies in the timing: `PreToolUse` is triggered unconditionally before every tool invocation, while `PermissionRequest` is only activated when Claude requires manual user confirmation for permissions. Through this event, you can programmatically approve or deny permission requests. See the following code example.

```
{
  "hookSpecificOutput": {
    "hookEventName": "PermissionRequest",
    "decision": {
      "behavior": "allow",
      "updatedPermissions": {}
    }
  }
}
```

UserPromptSubmit

Triggered after the user submits input and before Claude begins processing. This event is commonly used for input preprocessing or context injection scenarios. For example, automatically appending the current Git branch information every time the user sends a message.

Subagent Events

Subagent events can be used in conjunction with the subagent mechanism introduced in *Chapter 4*.

SubagentStart

Triggered when a subagent starts. Its matcher can filter based on the subagent's type name, supporting both built-in types (such as Bash, Explore, Plan) and custom subagents defined in the `.claude/agents/` directory (such as code-reviewer). Although SubagentStart cannot prevent a subagent from starting, it injects critical context information at runtime via `additionalContext`. For example, every time code-reviewer starts, the team's coding standards are automatically loaded.

SubagentStop

Triggered after a subagent completes its task. Its behavior logic is identical to the global Stop event: you can either allow the stop operation or intercept the request, forcing the subagent to continue working until specific quality standards are met. Additionally, the input data for SubagentStop includes two key paths: `transcript_path` (main session record) and `agent_transcript_path` (the subagent's own conversation record). With this information, Hook scripts can review the subagent's entire workflow, enabling precise evaluation of its output quality.

Completion Events

Stop

Triggered when Claude completes a full response cycle. This is the core mechanism for implementing "quality gating": if the output does not meet preset standards (such as coding guidelines, security policies, etc.), you can block the session from ending by setting `decision: "block"`, forcing Claude to continue revising or improving the work until it meets requirements.

Notification

Triggered when Claude sends a system notification. Its matcher can accurately distinguish between different types of notifications, such as `permission_prompt`, `idle_prompt`, or `auth_success`. This event is often used for integrating custom notification channels—for example, forwarding critical alerts to Slack or Discord or triggering local desktop pop-up reminders.

Newer Event Types

Between 2025 and 2026, Claude Code expanded its event system, adding several key event types to support more complex collaboration and operations scenarios.

TeammateIdle and TaskCompleted

Designed specifically for multi-agent team collaboration (Agent Teams). The former is triggered when a teammate agent is about to enter an idle state, while the latter is triggered when a task is marked as completed.

ConfigChange

Triggered when a configuration file changes. This event is mainly used for auditing and compliance, helping developers track the history of setting changes and prevent unauthorized configuration modifications.

WorktreeCreate and WorktreeRemove

Correspond to the creation and deletion of GitWorktrees, respectively. By intercepting these events, users can customize initialization settings for version control workflows (such as automatically installing dependencies) or cleanup logic (such as deleting temporary build artifacts).

"Can It Be Blocked?": The Most Critical Dimension

Among all 17 events, "can it be blocked" is the most fundamental classification dimension. It determines whether an event is used for "controlling the process" or merely for "observation and logging."

Events that can be blocked include `PreToolUse`, `PermissionRequest`, `UserPromptSubmit`, `UserPromptExpansion`, `Stop`, `SubagentStop`, `TeammateIdle`, `TaskCreated` (exit 2 rolls the task creation back), `TaskCompleted`, `ConfigChange`, `PostToolBatch`, `PreCompact` (exit 2 stops the compaction), and the two Elicitation events.

The remaining events (such as `PostToolUse`, `Notification`, `SubagentStart`, etc.) are read-only. They are mainly used to read context, inject additional information, or trigger side effects (such as sending notifications), but cannot directly block or modify Claude's core execution logic.

In daily development and operations, the three most commonly used events are: `PreToolUse` (the gatekeeper before tool execution), `PostToolUse` (the quality guardian after tool execution), and `Stop` (the quality gate at task completion). If time is limited, mastering `PreToolUse`, `PostToolUse`, and `Stop` first will enable you to build a robust automation loop.

Configuration System: 6 Locations, 6 Purposes

Claude Code's configuration system uses a layered overlay mechanism, with priority decreasing from top to bottom. Hook configurations use standard JSON format and strictly follow a six-layer priority architecture (see *Figure 5.1*). This design allows developers to flexibly deploy automation logic according to scope.

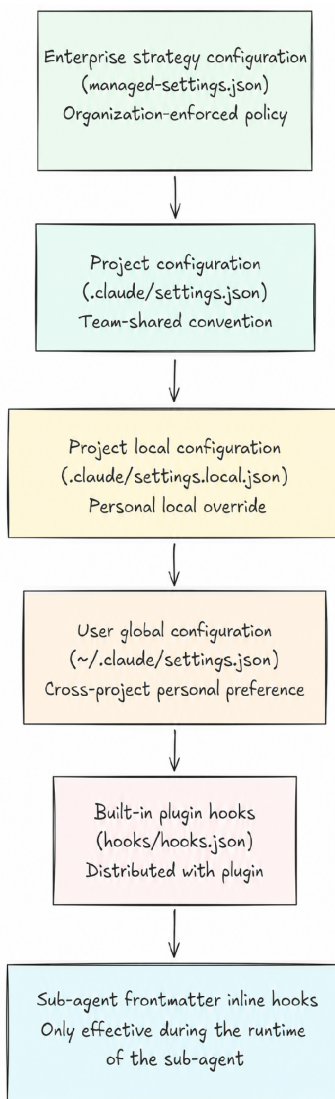


Figure 5.1: Six Levels and Scopes of Hook Configuration

Project configuration (`.claude/settings.json`) is the core vehicle for team collaboration. Once committed to the Git repository, all members can automatically synchronize the team's agreed security checks and automation rules when cloning the project. **Local project configuration** (`.claude/settings.local.json`) is ignored by `.gitignore` and is suitable for personal scenarios where overriding the team's default configuration is needed. **User global**

configuration (`~/.claude/settings.json`) is used to manage personal preferences across projects, such as custom log formats or desktop notification methods.

The configuration structure uses a three-level nested design: event type → matcher group → list of Hook handlers. See the following code example.

```
{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          {
            "type": "command",
            "command": "./.claude/hooks/block-dangerous.sh",
            "timeout": 30
          }
        ]
      }
    ],
    "PostToolUse": [
      {
        "matcher": "Write|Edit",
        "hooks": [
          {
            "type": "command",
            "command": "INPUT=$(cat); FILE=$(echo \"${INPUT}\" | jq -r '.tool_input.file_path'); prettier --write \"${FILE}\""
          }
        ]
      }
    ]
  }
}
```

The **matcher** field is used to specify the scope of tools to which this group of Hooks applies.

- "Bash": Matches all Bash invocations
- "Write|Edit": Matches the Write or Edit tools (the pipe symbol | denotes logical "or")
- "": Matches all tools

For lifecycle events such as `Stop`, `Notification`, and `UserPromptSubmit`, the `matcher` field is ignored because these events do not target specific tools. In `SubagentStart` or `SubagentStop` events, however, `matcher` matches the subagent type name rather than the tool name.

Three Types of Processors: A Deterministic Ladder

Hook handlers come in five types. Three of them — `command`, `prompt`, and `agent` — form a ladder of decreasing determinism and increasing flexibility, and the choice among them follows from how strict the check needs to be. The other two are integration hatches rather than rungs on that ladder: `http POSTs` the event JSON to a URL you control, and `mcp_tool` calls a tool on an MCP server that is already connected. The `/hooks` menu lists all five.

Command Type: Deterministic Rules

This type is used to execute shell commands or scripts. As the most commonly used and most reliable type, deterministic rules are always more trustworthy than judgments made by large models. See the following code example.

```
{
  "type": "command",
  "command": "./.claude/hooks/check-security.sh",
  "timeout": 30
}
```

Hooks of the `command` type receive JSON-formatted context data via `stdin` (including `session_id`, `tool_name`, `tool_input`, etc.), output decisions in JSON format via `stdout`, and express their final intent through the exit code.

- **Exit code 0:** Success. The system will parse the JSON from `stdout` as the basis for its decision.
- **Exit code 2:** Intentional block. The system will return the content of `stderr` to Claude as the reason for the error.
- **Other exit codes:** a script error. The run is not interrupted, but the failure is no longer silent — the transcript shows a "`<hook name> hook error`" notice with the first line of `stderr`, and the full `stderr` goes to the debug log.

The design of `Exit code 2` is crucial, as it strictly distinguishes between "intentional blocking of an operation" and "script malfunction." Script failures should not impede the normal workflow—just as a malfunctioning smoke detector should not prevent people from entering or leaving a building.

Prompt Type: Single Large Model Evaluation

When validation logic requires some degree of judgment but does not need to perform multi-step operations, it is recommended to use the prompt type. This type will call a lightweight large model (typically Haiku) to evaluate the current situation. See the following code example.

```
{
  "type": "prompt",
  "prompt": "Evaluate whether this code modification introduces
    security vulnerabilities. $ARGUMENTS",
  "model": "claude-haiku-4-5",
  "timeout": 30
}
```

Here, \$ARGUMENTS is a placeholder that will be replaced at runtime with the complete input JSON received by the Hook. The response from the large model must follow the JSON format below.

JSON format for allowing the operation:

```
{
  "ok": true,
  "reason": "The code modification is safe and
    does not introduce any known vulnerability patterns."
}
```

JSON format for rejecting the operation.

```
{
  "ok": false,
  "reason": "Potential SQL injection risk detected: user input
    is concatenated directly into the query string without escaping."
}
```

Agent Type: Multi-turn Intelligent Verification

When the verification logic requires actually inspecting code files, performing searches, or multi-step operations to reach a conclusion, the agent type should be used. This type will launch a sub-agent, enabling it to utilize tools such as Read, Grep, and Glob for multi-turn, in-depth verification. See the following code example.

```
{
  "type": "agent",
  "prompt": "Check whether all modified files have passed unit tests.
    Run the test suite and verify the results. $ARGUMENTS",
  "timeout": 120
}
```

A sub-agent of the agent type must return a decision after a maximum of 50 dialogue/operation rounds. Its response format is identical to that of the prompt type, returning a JSON object containing ok (boolean) and reason (string).

When selecting a Hook processor type, you should follow the downgrade principle: "Use the **command** type whenever possible instead of the **prompt** type, and use the prompt type instead of the **agent** type whenever possible" (see Figure 5.2).

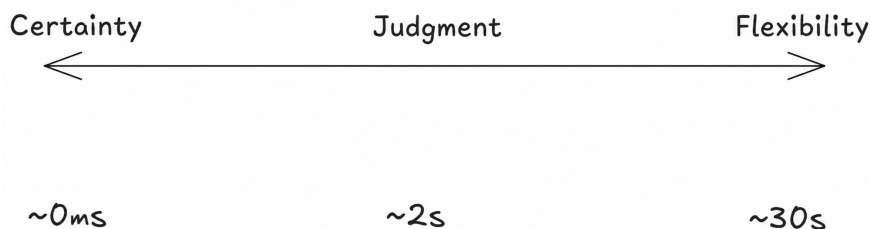


Figure 5.2: Comparison of Determinism and Flexibility Among the Three Hook Processor Types

Deterministic rules (such as pattern matching, filename checks, and regular expressions) will always outperform large model judgments in terms of speed and reliability. Only when the verification logic truly requires "comprehension" (semantic analysis) or "code inspection capabilities" (multi-file context retrieval) should you consider upgrading to the prompt type or agent type.

hookSpecificOutput: Protocol for Communicating with Claude

The output format for the PreToolUse Hook underwent a significant upgrade in 2025. Earlier versions used top-level decision and reason fields, while the new version recommends using the permissionDecision format nested within the hookSpecificOutput object. Although both formats are currently supported, new code should follow the new specification. See the following code example.

```
{
  "hookSpecificOutput": {
    "hookEventName": "PreToolUse",
    "permissionDecision": "deny",
    "permissionDecisionReason": "This command attempts to delete a
      protected system directory",
    "additionalContext": "Protected path patterns: /etc, /usr, /var"
```

```
}  
}
```

permissionDecision supports four possible values.

- allow: Bypass the permission system and execute directly.
- deny: Block execution.
- ask: Hand the decision to the user.
- defer: Fall through to the normal permission flow.

Among these, ask is a subtle yet practical option—it does not automatically reject the action, but instead conveys the attitude of "I'm not sure, let a human decide."

The additionalContext field applies to all event types, and its content will be injected into Claude's context. This mechanism creates an efficient feedback loop. For example, the PostToolUse Hook can use additionalContext to feed code static analysis (lint) results back to Claude. Upon receiving this information, Claude will automatically fix the issues, with no human intervention required throughout the process.

In addition, all events support the following common top-level fields.

```
{  
  "continue": false,  
  "stopReason": "Security violation detected, session terminated",  
  "suppressOutput": false,  
  "systemMessage": "Warning: This operation has been blocked by  
    the security policy"  
}
```

- continue: false: This functions as an "emergency brake." Regardless of the current event stage, this setting will immediately terminate Claude's processing.
- systemMessage: The content of this field will be displayed directly to the user and will not be passed to Claude.

Engineering Practice 1: Security Protection System

Let's return to the challenging case presented at the beginning of this chapter. Now, we will use Hooks to build a comprehensive security protection system, safeguarding the project with the following three lines of defense: dangerous command interception, sensitive file protection, and comprehensive operation auditing.

PreToolUse: Dangerous Command Interception

The first line of defense is designed to intercept Bash commands that could potentially cause disasters. This script incorporates several key details in its design.

- Debug information is output to standard error (stderr, i.e., `>&2`), rather than standard output (stdout). This is because stdout must be strictly reserved for outputting JSON-formatted decision results.
- The jq tool is used to parse the input JSON data, avoiding fragile manual string matching.
- Each interception operation is accompanied by a clear and specific reason for the action.

See the following code example.

```
#!/bin/bash
# .claude/hooks/block-dangerous.sh

set -e
INPUT=$(cat)

# Extract the command (debug output goes to stderr)
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command // ""')
echo "DEBUG: Checking command: $COMMAND" >&2

# List of dangerous command patterns
DANGEROUS_PATTERNS=(
    "rm -rf /"
    "rm -rf ~"
    "rm -rf $HOME"
    "> /dev/sd"
    "mkfs."
    ":(){:|:&};:" # Fork bomb
    "chmod -R 777 /"
    "git push --force origin main"
    "git push --force origin master"
    "git reset --hard origin"
    "DROP DATABASE"
    "DROP TABLE"
    "TRUNCATE"
    "curl.*| sh" # Dangerous pipe execution
)
```

```

    "curl.*| bash"
)

for pattern in "${DANGEROUS_PATTERNS[@]"}; do
    if [ "$COMMAND" == "$pattern" ]; then
        echo "BLOCKED: $pattern" >&2
        cat <<EOF
    {
        "hookSpecificOutput": {
            "hookEventName": "PreToolUse",
            "permissionDecision": "deny",
            "permissionDecisionReason": "Intercepted dangerous command pattern:
$pattern"
        }
    }
    EOF
        exit 2
    fi
done

echo '{"hookSpecificOutput":
{"hookEventName": "PreToolUse", "permissionDecision": "allow"}}'
exit 0

```

PreToolUse: Sensitive File Protection

The second line of defense focuses on protecting sensitive files from accidental modification. The matcher for this Hook is configured as "Write|Edit", ensuring it is only triggered during file write or edit operations, thereby blocking risks at the source. Please see the following code example.

```

#!/bin/bash
# .claude/hooks/protect-files.sh

set -e
INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r '.tool_input.file_path // ""')

if [ -z "$FILE_PATH" ]; then
    echo '{"hookSpecificOutput":
{"hookEventName": "PreToolUse", "permissionDecision": "allow"}}'

```



```
exit 0
fi

FILENAME=$(basename "$FILE_PATH")

# Protected Filename Patterns
PROTECTED_FILES=( ".env" ".env.local" ".env.production" "credentials.json"
"secrets.yaml" "secrets.json" "id_rsa" "id_ed25519")

# Protected Extensions
PROTECTED_EXTENSIONS=( "pem" "key" "p12" "pfx")

# Protected Directories
PROTECTED_DIRS=( ".git/" ".ssh/" "node_modules/")

# Check Directories
for dir in "${PROTECTED_DIRS[@]}; do
if [[ "$FILE_PATH" == *"$dir"* ]]; then
    cat <<EOF
    {
        "hookSpecificOutput": {
            "hookEventName": "PreToolUse",
            "permissionDecision": "deny",
            "permissionDecisionReason": "Modifying files in protected directories is not
allowed: $dir"
        }
    }
EOF
exit 2
fi
done

# Check file names
for name in "${PROTECTED_FILES[@]}; do
if [[ "$FILENAME" == "$name" ]]; then
    cat <<EOF
    {
        "hookSpecificOutput": {
            "hookEventName": "PreToolUse",
            "permissionDecision": "deny",
            "permissionDecisionReason": "Modifying sensitive files is not allowed:
```

```

$name"
    }
}
# Check file extensions
exit 2
fi
done
EXT="${FILENAME##*.*}"
for ext in "${PROTECTED_EXTENSIONS[@]}"; do
if [[ "$EXT" == "$ext" ]]; then
    cat <<EOF
    {
        "hookSpecificOutput": {
            "hookEventName": "PreToolUse",
            "permissionDecision": "deny",
            "permissionDecisionReason": "Modification of key files is not allowed: *.
$ext"
        }
    }
fi
done

echo '{"hookSpecificOutput":
{"hookEventName": "PreToolUse", "permissionDecision": "allow"}}'
exit 0

```

PostToolUse: Comprehensive Operation Auditing

The third line of defense is comprehensive operation auditing. By configuring `matcher: "*"` , the tool can capture and record all tool invocations. In enterprise environments with strict compliance requirements, this is an indispensable mechanism, as it clearly answers the core audit question: "When did Claude perform what operation on which target?"

```

#!/bin/bash
# .claude/hooks/audit-log.sh

INPUT=$(cat)
LOG_DIR="${CLAUDE_PROJECT_DIR:-.}/.claude/logs"
mkdir -p "$LOG_DIR"
LOG_FILE="$LOG_DIR/audit-$(date +%Y-%m-%d).log"

```

```

TIMESTAMP=$(date -Iseconds)
TOOL_NAME=$(echo "$INPUT" | jq -r '.tool_name // "unknown"')
TOOL_INPUT=$(echo "$INPUT" | jq -c '.tool_input // {}')

echo "[$TIMESTAMP] $TOOL_NAME: $TOOL_INPUT" >> "$LOG_FILE"
echo '{} '
exit 0

```

Complete Configuration

By integrating the three independent scripts above into `.claude/settings.json`, you create a robust, in-depth defense system. The specific configuration is as follows.

```

{
  "hooks": {
    "PreToolUse": [
      {
        "matcher": "Bash",
        "hooks": [
          { "type": "command", "command": "./.claude/hooks/block-dangerous.sh" }
        ]
      }
    ],
    {
      "matcher": "Write|Edit",
      "hooks": [
        { "type": "command", "command": "./.claude/hooks/protect-files.sh" }
      ]
    }
  ],
  "PostToolUse": [
    {
      "matcher": "*",
      "hooks": [
        { "type": "command", "command": "./.claude/hooks/audit-log.sh" }
      ]
    }
  ]
}

```

This configuration establishes a comprehensive security closed loop encompassing "pre-emptive interception, real-time protection, and post-event auditing": interception of dangerous commands → protection of sensitive files → full operation auditing. Through this combined mechanism, we upgrade the previously fragile process that relied on "human caution" into an automated security system where "code is law." As demonstrated in the case at the beginning of this chapter, if the second layer of protection had been applied, Xiao Zhang's "tragedy" could have been completely avoided.

Engineering Practice II: Automated Code Quality

Security protection aims to "prevent problems before they occur," while automated code quality is dedicated to "ensuring excellent delivery." This section's practical example will delve into the collaborative working mechanism of PostToolUse and Stop Hook.

PostToolUse: Automatic Formatting

Every time Claude writes to a file, the system will automatically trigger a formatting tool. The brilliance of this hook lies in its implementation of "separation of concerns": Claude does not need to be aware of the specific formatting standards adopted by the project and only needs to focus on writing the code logic, while the formatting process is completed seamlessly in the background. See the following code example.

```
#!/bin/bash
# .claude/hooks/auto-format.sh

set -e
INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r '.tool_input.file_path // ""')

if [ -z "$FILE_PATH" ] || [ ! -f "$FILE_PATH" ]; then
    echo '{}'
    exit 0
fi

EXTENSION="${FILE_PATH##*.*}"

case "$EXTENSION" in
    js|jsx|ts|tsx|json|md|css|scss|html)
        if command -v npx &> /dev/null; then
            npx prettier --write "$FILE_PATH" 2>/dev/null
            echo '{"hookSpecificOutput":{"additionalContext":"Formatted with Prettier"}}'
```

```

        fi
        ;;
    py)
        if command -v black &> /dev/null; then
            black "$FILE_PATH" 2>/dev/null
            echo '{"hookSpecificOutput":{"additionalContext":"Formatted with
Black"}}}'
        fi
        ;;
    go)
        if command -v gofmt &> /dev/null; then
            gofmt -w "$FILE_PATH" 2>/dev/null
            echo '{"hookSpecificOutput":{"additionalContext":"Formatted with
gofmt"}}}'
        fi
        ;;
    *)
        echo '{} '
        ;;
esac
exit 0

```

The script introduces the use of `command -v` for environment checks. If the formatting tool is not detected as installed, the Hook will silently skip the step instead of throwing an error. This reflects the design principle of "graceful degradation": exceptions within the Hook itself should not block the normal operation of the core workflow.

PostToolUse: Lint Feedback Loop

Automatic formatting ensures code "aesthetics," while Lint checks guarantee code "correctness." The core of this solution lies in leveraging `additionalContext` to feed Lint check results back to Claude, thereby establishing an automated closed loop of "modification → checking → feedback → correction." See the following code example.

```

#!/bin/bash
# .claude/hooks/lint-check.sh

set -e
INPUT=$(cat)
FILE_PATH=$(echo "$INPUT" | jq -r '.tool_input.file_path // ""')

```

```

if [[ "$FILE_PATH" == *.js || "$FILE_PATH" == *.ts || "$FILE_PATH" == *.jsx ||
"$FILE_PATH" == *.tsx ]]; then
    LINT_RESULT=$(npx eslint "$FILE_PATH" 2>&1) || true

    if [ $? -ne 0 ]; then
        ESCAPED=$(echo "$LINT_RESULT" | head -30 | jq -Rs '.')
        echo "{\"hookSpecificOutput\":{\"additionalContext\": \"ESLint found
problems:\\n${ESCAPED}\\\"}}\"
    else
        echo '{\"hookSpecificOutput\":{\"additionalContext\":\"ESLint check passed\"}}'
    fi
else
    echo '{} '
fi
exit 0

```

The script introduces command `-v` for environment checks. If it detects that the formatting tool is not installed, the Hook will silently skip instead of throwing an error. This reflects the design principle of "graceful degradation": exceptions in the Hook itself should not block the normal operation of the core workflow.

Stop Hook: Test Quality Gate

The Stop Hook serves as a line of defense for quality assurance: when Claude claims the task is complete, the test suite is automatically triggered. If the tests fail, the system will prevent the session from ending and force continued remediation. See the following code example.

```

#!/bin/bash
# .claude/hooks/run-tests.sh
INPUT=$(cat)

# [Key Mechanism] Preventing Infinite Loops: Check the stop_hook_active Flag
# If this flag is true, it indicates a retry has already occurred; this time, the
process must be allowed to proceed to avoid a deadlock.
STOP_ACTIVE=$(echo "$INPUT" | jq -r '.stop_hook_active // false')
if [ "$STOP_ACTIVE" = "true" ]; then
    exit 0 # Terminate interception and allow Claude to stop
fi

# Switch to the project directory
if [ -n "$CLAUDE_PROJECT_DIR" ]; then
    cd "$CLAUDE_PROJECT_DIR"

```

```

fi

# Detect project type and run tests
TEST_PASSED=true
TEST_RESULT=""

if [ -f "package.json" ] && grep -q "test" package.json; then
    TEST_RESULT=$(npm test 2>&1) || TEST_PASSED=false
elif [ -f "pyproject.toml" ] || [ -f "pytest.ini" ]; then
    TEST_RESULT=$(pytest 2>&1) || TEST_PASSED=false
elif [ -f "go.mod" ]; then
    TEST_RESULT=$(go test ./... 2>&1) || TEST_PASSED=false
else
    echo '{"hookSpecificOutput":{"additionalContext":"No test framework detected"}}'
    exit 0
fi

if [ "$TEST_PASSED" = true ]; then
    echo '{"hookSpecificOutput":{"additionalContext":"All tests passed"}}'
else
    # Capture the first 50 lines of the error log and escape
    TEST_ESCAPED=$(echo "$TEST_RESULT" | head -50 | jq -Rs '.')
    # Return block decision, forcing Claude to continue working
    cat <<EOF
{
  "decision": "block",
  "reason": "Test failed, please fix before stopping",
  "hookSpecificOutput": {
    "additionalContext": $TEST_ESCAPED
  }
}

```

The `stop_hook_active` field in the script is the key to preventing the system from falling into an "infinite loop." Its logic is similar to the termination condition of a recursive function. When the Stop Hook fails to execute, if the system attempts to repair and triggers the Hook again, `stop_hook_active` will be set to true. The script then detects this flag and chooses to allow the process to continue, thereby breaking the loop—just as a recursive function must have a termination condition, the Stop Hook must also have a clear exit mechanism. Without this check, Claude would fall into a "repair → test failed → repair again" infinite loop, unable to escape on its own.

Subagent Hooks: Precise Context Management

In *Chapter 3*, we learned that subagents achieve task delegation by isolating context. The Hooks system provides two dedicated events for this purpose—**SubagentStart** and **SubagentStop**. However, even more crucial is a third mechanism: defining **Hooks** directly in the subagent's **Frontmatter**.

Global vs. Frontmatter: The Precision Issue

Suppose you have a subagent named `db-reader`, dedicated to executing SQL queries. If you need to audit every Bash command it runs to guard against SQL injection risks, configuring the Hook in the global settings.json is not the best approach. A global configuration would indiscriminately intercept all Bash commands, including those for compiling code, running tests, or installing dependencies—operations unrelated to the database. This not only wastes system resources but also greatly increases the risk of false positives.

A better solution is to define the Hook directly in the subagent's Frontmatter. See the following code example.

```
HAS_ISSUES=$(grep -c "issue\|problem\|bug" "$TRANSCRIPT" || true)
HAS_SUGGESTIONS=$(grep -c "suggest\|suggestion\|recommend" "$TRANSCRIPT" || true)
---
name: db-reader
description: Read-only database analysis tool
tools: Read, Grep, Glob, Bash
hooks:
PreToolUse:
- matcher: "Bash"
hooks:
- type: command
command: "./.claude/hooks/check-sql-injection.sh"
Stop:
- hooks:
- type: prompt
prompt: "Check whether the query results contain PII (such as names, email
addresses, or phone numbers). If any are found, reply with ok: false and request
that the data be desensitized."
---
```

```
You are a database analysis expert. Only execute SELECT queries; executing any SQL
statements that modify data is strictly prohibited.
```


The core advantage of defining hooks with Frontmatter lies in their **tight lifecycle binding**: hooks are activated with the launch of a subagent and are automatically cleaned up after the task is completed. In addition, configuration is integrated within the same file as the subagent definition, allowing distribution together with the `.md` file. Users do not need to modify a `global settings.json`, which greatly reduces configuration complexity.

SubagentStart: Automatic Context Injection

A typical use case for the SubagentStart hook is to dynamically inject team guidelines when a subagent starts. For example, whenever the code-reviewer subagent is launched, the system can automatically inject the team's coding standards. See the following code example.

```
{
  "hooks": {
    "SubagentStart": [
      {
        "matcher": "code-reviewer",
        "hooks": [
          {
            "type": "command",
            "command": "echo '{\"hookSpecificOutput\":{\"hookEventName\" :\"SubagentStart\", \"additionalContext\": \"Team coding standards: use camelCase naming, maximum line length 100 characters, all public APIs must include JSDoc comments\"}}'"
          }
        ]
      }
    ]
  }
}
```

SubagentStop: Verifying Output Quality

The SubagentStop Hook can be used to verify whether the subagent's work meets the required standards. By using `agent_transcript_path` to read the complete interaction record of the subagent, the system can perform fine-grained quality acceptance. See the following code example.

```
#!/bin/bash
# verify-review-quality.sh
```

```

INPUT=$(cat)
AGENT_TYPE=$(echo "$INPUT" | jq -r '.agent_type')
STOP_ACTIVE=$(echo "$INPUT" | jq -r '.stop_hook_active')

# Only verify the code-reviewer subagent
if [ "$AGENT_TYPE" != "code-reviewer" ]; then exit 0; fi

# Prevent infinite loops (if currently in the stop hook trigger phase, skip)
if [ "$STOP_ACTIVE" = "true" ]; then exit 0; fi

TRANSCRIPT=$(echo "$INPUT" | jq -r '.agent_transcript_path')

if [ -f "$TRANSCRIPT" ]; then
    HAS_ISSUES=$(grep -c "issue\|problem\|bug" "$TRANSCRIPT" || true)
    HAS_SUGGESTIONS=$(grep -c "suggest\|suggestion\|recommend" "$TRANSCRIPT" ||
true)

if [ "$HAS_ISSUES" -gt 0 ] && [ "$HAS_SUGGESTIONS" -eq 0 ]; then
    echo '{"decision":"block","reason":"Issues were found but no suggestions for fixes
were provided. Please supplement each issue with an improvement plan."}'
    exit 0
fi
fi

exit 0

```

These three layers of protection each serve their own purpose, together forming a complete quality assurance feedback loop.

- **Frontmatter Hook:** Responsible for internal self-checks, ensuring the subagent asks itself, "Is my output complete?"
- **SubagentStart Hook:** Responsible for external injection, providing the necessary context at startup ("giving it the required background information")
- **SubagentStop Hook:** Responsible for external acceptance, rigorously verifying at the end, "Does its work meet the required standards?"

Asynchronous Hooks: Background execution without blocking

By default, Hook scripts execute synchronously and block—Claude will pause the current workflow until the script finishes and returns a result. For quick pattern-matching checks, this delay of a few milliseconds is usually negligible; however, for time-consuming operations such as running test suites, calling external APIs, or sending notifications, synchronous blocking can significantly slow down Claude's response speed and impact user experience.

To address this, asynchronous Hooks support was introduced in Claude Code in early 2026.

```
{
  "type": "command",
  "command": "./.claude/hooks/run-tests-background.sh",
  "async": true,
  "timeout": 300
}
```

With `async: true` configured, the Hook will run in the background in a non-blocking manner. Claude does not need to wait for its completion and can immediately proceed with subsequent tasks. Once the asynchronous Hook finishes execution, its output will be automatically passed to Claude in the next conversation turn for reference or processing.

There are two key limitations to asynchronous Hooks that define their scope of applicability.

- **Type limitation:** Only **command**-type Hooks support asynchronous execution. Prompt- and agent-type Hooks must run synchronously.
- **Interception capability limitation:** Asynchronous Hooks cannot block the current operation. Since the main process continues as soon as the Hook is triggered, asynchronous Hooks lose the opportunity to intervene before the operation occurs.

Therefore, asynchronous Hooks are suitable for "post-processing" tasks such as logging, asynchronous notifications, background data validation, and non-critical quality audits. They are not suitable for real-time blocking security checks (such as SQL injection defense or sensitive information filtering).

Environment Variables and Debugging

Environment Variables Available to Hooks

Table 5.2 lists the environment variables available to Hooks, covering their scope and core usage.

| Core use | All Hooks | Get the absolute path of the current project's root directory |
|--------------------|-------------------|---|
| CLAUDE_PROJECT_DIR | All Hooks | Unique identifier for the current session |
| CLAUDE_ENV_FILE | Notification only | Contains the specific notification message content |
| CLAUDE_CODE_REMOTE | Plugin Hook only | Root directory path where the plugin is installed |
| CLAUDE_PLUGIN_ROOT | File path | Scope |

Table 5.2: Environment Variables Available to Hooks

The Three Debugging Techniques

There are three core methods for debugging Claude Hook scripts:

- 1. **Output debugging information to stderr:** Since stdout is reserved for outputting JSON decision results, all debugging information must be redirected to stderr.

```
echo "DEBUG: Checking file $FILE_PATH" >&2 # Debug information
echo '{"decision": "allow"}'                # JSON decision
```

- 2. **Manually test the Hook script:** Directly verify the script logic by constructing simulated input.

```
echo '{"tool_name": "Bash", "tool_input": {"command": "rm -rf /"}}'
| ./claude/hooks/block-dangerous.sh
echo "Exit code: $?"
```

- 3. Use `claude --debug` to view detailed **Hook** execution information.

Debug mode displays the list of matched Hooks, the execution time for each script, and the returned results.

Common Pitfalls

There are two frequently overlooked issues when using Hooks.

- If your shell configuration files (such as `~/ .zshrc` or `~/ .bashrc`) contain unconditional echo statements (for example, to display a welcome message), these outputs will pollute standard output (`stdout`), causing JSON parsing to fail. The solution is to wrap these echo statements with a `[[ $- == *i* ]]` conditional check to ensure they only execute in interactive shells.
- After directly editing `settings.json`, Hooks often do not take effect immediately. This is because Claude Code only captures a configuration snapshot at startup; changes to the file during runtime are not automatically synchronized. To apply changes, users need to confirm them in the **/hooks** menu or restart the current session.

Engineering Design Methodology

When faced with specific automation requirements, designing a Hook solution requires clarifying the following three core dimensions:

- **Interception timing (event selection):** For pre-operation interception, use `PreToolUse` or `UserPromptSubmit`; for post-operation feedback, use `PostToolUse`; for completion checks, use `Stop` or `SubagentStop`; for lifecycle management, use `SessionStart` or `SessionEnd`
- **Judgment method (type selection):** If the rules are clear (such as pattern matching or file checks), use the command type; if semantic judgment is needed but input is sufficient, use the prompt type; if deep code analysis is required, use the agent type
- **Configuration scope (location selection):** For team-wide standards, configure in `.claude/settings.json`; for personal preferences, configure in `~/ .claude/settings.json`; for subagent-specific checks, configure in `Frontmatter`

The design process typically follows a "three-step" strategy.

1. First, configure an audit log Hook based on the `PostToolUse` event with `matcher:""` to observe Claude's actual tool invocation patterns and accumulate several days of real operational data

2. Based on the audit data, identify high-risk operation patterns and design targeted PreToolUse interception rules
3. Gradually tighten interception rules while always retaining the logging function to ensure that any false positives can be quickly traced to their root cause

Note**Kage's notes**

Hooks are team-level infrastructure, not personal experimental toys. Hooks configured in `.claude/settings.json` will apply to all members who clone the repository. If a member is unexpectedly blocked for unknown reasons, it will seriously disrupt workflows and cause frustration. Therefore, be sure to follow these guidelines:

- Before submitting Hook configurations, fully discuss and reach consensus with the team.
- Every interception rule must include a clear reason statement to inform users of the specific reason for being blocked.
- Use audit logs to monitor Hook trigger frequency in real time, so that false positives can be promptly detected and corrected.

Summary

Hooks fill the gap left by `CLAUDE.md` and Skills: they enforce execution rules at the moment a tool is about to run. Some of those rules are team conventions; others — blocking a destructive command, refusing to touch a credentials file — are simply safety, and would apply on a solo project too. As the only component in the Claude Code extension mechanism that sits at the execution layer rather than the cognitive layer, Hooks do not depend on Claude understanding anything: the script intercepts the call.

More than two dozen event types cover the whole arc, from session start to session end and from the main conversation into sub-agents; check the official reference for the current list.

Three processor types (command, prompt, agent) form a capability ladder of "decreasing determinism, increasing flexibility." The selection principle is: use command first, then prompt, and finally agent.

Supports configuring inline Hooks in the Frontmatter of subagents for more precise control than global configuration.

The introduction of asynchronous Hooks effectively solves the blocking problem caused by long-running operations.

From an engineering methodology perspective, the design of Hooks strictly follows the principle of "observe first, then control": first use the `PostToolUse` event to record audit logs and deeply analyze real tool invocation patterns and behavior characteristics; then, based on accumulated data insights, design targeted `PreToolUse` interception rules to ensure the accuracy and necessity of policies. The built-in `stop_hook_active` flag effectively prevents infinite recursive calls triggered by Hook logic itself. `Exit code 2` is defined to represent "intentional blocking," clearly distinguishing it from system errors, reflecting thoughtful consideration in exception handling and state definition.

The core value of Hooks is not in "expanding capability boundaries" (making Claude do more), but in "strengthening the execution foundation" (making everything Claude does more reliable). It upgrades those critical checks that "should happen but are often forgotten" (such as sensitive file protection, code formatting standards, test acceptance processes) from self-discipline-based "soft constraints" to unavoidable system-level "hard constraints." The painful lesson mentioned in the opening case could have been fundamentally avoided with a simple `PreToolUse` Hook.

At this point, we have thoroughly analyzed the internal extension mechanism of Claude Code. In *Chapter 6*, we will shift our focus outward to explore how it connects to the broader external world: MCP.

Discussion questions

1. In daily development workflows, what are some "errors that should never happen but occasionally occur due to carelessness"? Choose one such scenario and design a complete Hook solution: specify the trigger event, Hook type, Matcher configuration rules, and the specific conditions the script needs to check.
2. What are the similarities and differences between the "quality gate" mode of Stop Hooks and automated testing in CI/CD pipelines? In which scenarios should Stop Hooks be prioritized, and in which should CI/CD be preferred?
3. `PreToolUse` has interception capability, while `PostToolUse` is only for feedback. What is the engineering significance of this asymmetric design? If `PostToolUse` were given

interception capability (i.e., the ability to revoke already executed operations), what problems might arise?

4. Please compare the two Hook configuration methods: `global_settings.json` and the Frontmatter of sub-agents. In your project, which Hooks should be placed in the global configuration, and which are better defined in the Frontmatter of specific sub-agents?

6

Connecting Everything: Integrating External Tools with MCP

No man is an island, entire of itself; every man is a piece of the continent." — John Donne

As the technical review meeting was drawing to a close, the product manager raised what seemed like a simple request: *"Let's have Claude analyze the distribution patterns of all P1-level bugs in Jira from last week."*

Xiaobing quickly calculated in her mind: Jira is deployed on the company's internal network, while Claude runs on Anthropic's cloud servers. There is a natural network isolation wall between these two worlds. If Claude is to access Jira data, the options are to manually export and paste the data, write a script to fetch and feed the data, or...

"Or use MCP," Kage said before she could speak, *"which stands for Model Context Protocol (MCP)."*

"I've seen this name in several places, but I've never really understood what core problem it solves," Xiaoxue chimed in.

"It solves the oldest problem between AI and data—how to achieve a secure, standardized connection." Kage drew two circles on the whiteboard, labeling the left "Claude" and the right "Jira/Database/API," with a line connecting them. *"Before MCP, each end of this line had to be bridged anew every time: if GitHub wanted to connect to Claude, you had to write an adapter; if Jira wanted to connect to Claude, you needed another one. And if you switched AI platforms, all the adapters had to be rebuilt from scratch. At its core, this is an $N \times M$ complexity problem."*

Xiaobing immediately caught on: "Like charging cables before USB-C became widespread? Every device needed its own dedicated cable, and when traveling, you'd have to bring six or seven different ones?"

"Exactly. MCP is the USB-C of AI tool integration," Kage nodded.

From N×M to N+M: The Power of Standardization

The USB-C analogy is not just rhetorical; structurally, it precisely maps to the core problem MCP solves.

Think of USB-C. In the years before a single connector won out, laptops, phones, and cameras arrived with a thicket of incompatible ports and chargers, and every pairing needed its own cable. The analogy is a simplification of a long history — USB itself had already displaced an earlier generation of serial and parallel ports — but the shape of the problem is the right one:

MCP implements exactly the same logic (see *Figure 6.1*). Before MCP, every AI tool vendor and specific data source integration required a dedicated adapter to be developed and maintained. For example, to let GitHub access Claude's code, you needed to develop a GitHub-Claude adapter; to connect the same GitHub to ChatGPT, you had to develop a GitHub-ChatGPT adapter. Suppose there are 5 AI clients and 100 data sources; in theory, you'd need to maintain 500 independent adapters—meaning every new AI client or data source brings a huge amount of redundant integration work.

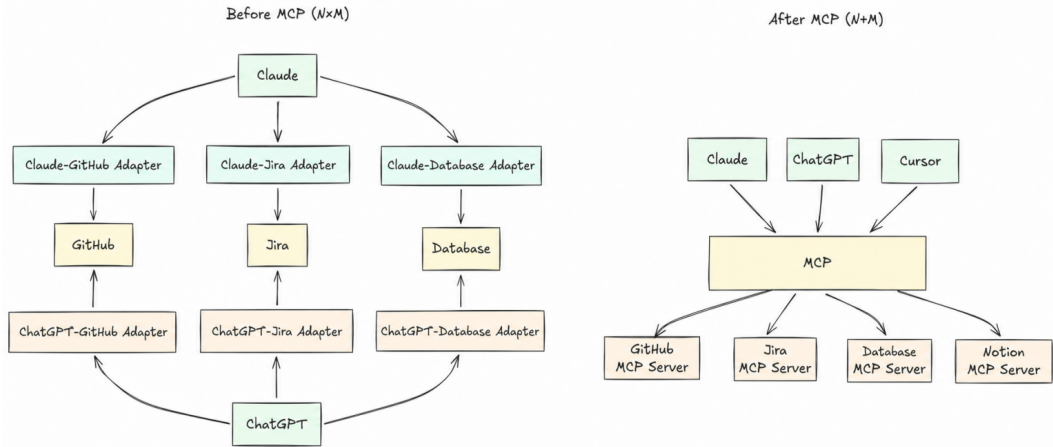


Figure 6.1: MCP simplifies the N×M problem to an N+M problem

The core idea of MCP is precisely this: define a universal standard protocol. In this architecture, each AI client (such as Claude, ChatGPT, etc.) only needs to implement the MCP client protocol once, and each data source (such as Jira, GitHub, databases, etc.) only needs to implement the

MCP server protocol once. As long as both sides adhere to this standard, they can plug and play, connecting with each other without the need to write targeted adapter code for every combination.

Returning to the earlier example, with 5 AI clients and 100 data sources, adopting MCP reduces the development workload from a staggering 500 times (5×100) to just 105 times ($5 + 100$).

AAIF announced in March 2025 that it would adopt MCP: support landed first in OpenAI's Agents SDK, with the Responses API and the ChatGPT desktop app to follow

The AAIF was co-founded by Anthropic, Block, and OpenAI, each contributing a founding project — MCP, goose, and AGENTS.md respectively — with Google, Microsoft, Amazon Web Services, Cloudflare, and Bloomberg joining as platinum members and supporters

Monthly downloads of the official MCP SDKs (Python and TypeScript) exceeded 97 million, eliminating the pain of $N \times M$ integration, a pressing, industry-wide need

Architecture: Client-Server and JSON-RPC

MCP adopts the classic client-server architecture (see *Figure 6.2*), communicating via the JSON-RPC 2.0 protocol.

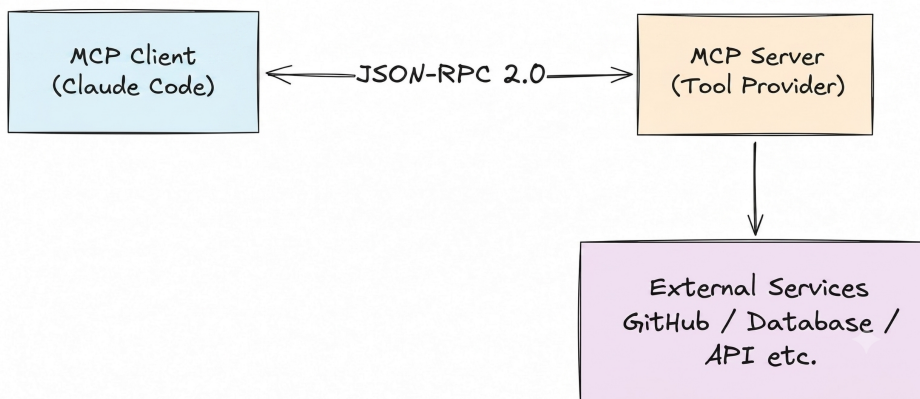


Figure 6.2: MCP client-server architecture

The MCP client is built into Claude Code, responsible for discovering the capabilities provided by MCP servers, constructing tool invocation requests, and handling responses. The MCP server is an independent program that connects to external services such as databases, APIs, or file systems, and exposes these services' capabilities to the client via MCP.

MCP's co-creators, David Soria Parra and Justin Spahr-Summers, modeled the protocol on the Language Server Protocol; Soria Parra has described in interviews how MCP borrowed LSP's message-flow design philosophy.

An MCP server can expose the following three core capabilities to clients:

- **Tools:** Tools are the most fundamental and commonly used capability. Tools are functions that Claude can actively invoke, such as "create a new ticket in Jira," "query orders from the past 7 days in the database," or "send a message to Slack." Each tool defines a clear name, description, input parameter schema, and return value format. When Claude determines a tool needs to be called, it automatically constructs a request matching the schema; after the server executes the operation, it returns a structured result.
- **Resources:** Resources are conceptually similar to "files," referring to data that Claude can read on demand. The key difference from tools is: tools are for performing operations (usually with side effects), while resources are for retrieving data (read-only). For example, "get the contents of README.md" is resource access, while "commit to the codebase" is a tool invocation.
- **Prompts:** Prompts are reusable prompt templates designed to standardize the initiation of specific tasks. For example, a code review–dedicated MCP server might provide a code-review prompt template, which automatically loads a standard review format and focus points when invoked. Although this feature overlaps somewhat with Skills and is less widely used in practice than the first two, it provides a convenient path for task initialization.

Among these three capabilities, tools account for the vast majority of real-world use cases. For example, when a user asks, "help me find P1-level bugs in Jira," what Claude calls is the `list_issues` tool exposed by the Jira MCP server.

Transmission Methods: Three Forms of Connection

The MCP transport layer defines the physical communication mechanism between client and server. Currently, three main transmission methods are supported.

- **stdio (Standard Input/Output):** This method is specifically designed for local process communication. In this mode, the client (such as **Claude Code**) launches the MCP server as a subprocess and exchanges JSON-RPC messages via standard input (`stdin`) and standard output (`stdout`). The advantages of this approach are that it runs entirely locally, requires no network connection, offers the highest security, and has the lowest latency. It is suitable for file system access, local database connections, and integration

with local development tools. The vast majority of official MCP servers started via npx use this method.

- For remote HTTP servers the MCP authorization specification mandates OAuth 2.1 with mandatory PKCE, and it integrates readily with enterprise single sign-on systems.
- **SSE (Server-Sent Events)**: This was the initial solution for remote transmission, but the standalone SSE transport has been superseded by Streamable HTTP in the MCP specification; if you see SSE config in older projects, migrate to HTTP.

Following the simple rule of "use stdio for local, HTTP for remote" will meet the vast majority of needs.

Configuration Details: From CLI to Configuration File

Quick CLI Configuration

Claude Code provides a set of `claude mcp` commands for conveniently managing MCP server configurations. See the code examples below.

```
# Add a local stdio server: Start a local file system service with the working
directory set to /workspace
claude mcp add filesystem npx -y @modelcontextprotocol/server-filesystem /
workspace

# Add a remote HTTP server: Connect to the company's internal Jira service
claude mcp add --transport http company-jira https://jira.company.com/mcp

# Add a user-level server (globally available): Add the GitHub service to the
user-level configuration, making it effective for all projects
claude mcp add --scope user github -- npx -y @modelcontextprotocol/server-github

# Add a server with authentication: Pass an authentication token via a custom HTTP
header
claude mcp add --transport http --header "Authorization: Bearer ${TOKEN}" api
https://api.example.com/mcp

# Management commands
claude mcp list          # List configurations: View all currently configured MCP
servers
claude mcp test github # Test connection: Verify the connection status and
availability of a specified server (such as GitHub)
```

```
claude mcp remove github # Remove server: Delete the specified MCP server
configuration
```

.mcp.json Configuration File

The `claude mcp` CLI commands essentially operate on the `.mcp.json` configuration file. Understanding the structure of this file is key to achieving fine-grained configuration. See the code example below.

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-filesystem",
        "/workspace"],
      "env": {}
    },
    "company-jira": {
      "type": "http",
      "url": "https://jira.company.com/mcp",
      "headers": {
        "Authorization": "Bearer ${JIRA_TOKEN}"
      }
    },
    "postgres": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres"],
      "env": {
        "DATABASE_URL": "${DB_URL:-postgresql://localhost:5432/mydb}"
      }
    }
  }
}
```

Environment variable substitution is the most important feature in the configuration file above. The basic syntax `${VAR_NAME}` directly replaces the placeholder with the corresponding environment variable value at runtime; the default value syntax `${VAR_NAME:-default_value}` uses the specified `default_value` if the environment variable `VAR_NAME` is not set. By leveraging environment variable substitution, you can store sensitive data (such as API keys, database passwords, etc.) in the operating system's environment variables or a `.env` file, rather

than hardcoding them in the `mcp.json` configuration file. Since the configuration file contains only variable references (such as `${JIRA_TOKEN}`) instead of actual secrets, you can safely commit `mcp.json` to your Git repository for team sharing without worrying about credential leaks.

Configuration File Location and Scope

The scope of the `mcp.json` configuration file varies depending on its location, as shown in *Table 6.1*.

| Location | Scope | Applicable scenario |
|---|--------------------------|--|
| <code>.mcp.json</code> (project root directory) | Project level | Applies only to the current project. Suitable for team-shared services bound to a specific codebase (the project's database connection, specific internal tools). Usually committed to Git. |
| <code>~/.claude/</code> directory | All projects, user level | Applies to all projects. Suitable for personal general tools (personal file-system access, global code-search tools). Lives in the user's home directory and does not change with the project. |

Table 6.1: Location and scope of the MCP configuration files

If credentials for a particular MCP server should not be committed to Git (such as a personal GitHub Token), you can store the credential portion in `.claude/settings.local.json` (which is already ignored by `.gitignore`), while keeping the server's basic configuration in `mcp.json` for team sharing.

Practical Example 1: Connecting to a Database

Let's start with the most common requirement: enabling Claude to query a database directly.

First, configure a PostgreSQL MCP server. See the code example below.

```
{
  "mcpServers": {
    "postgres": {
      "command": "npx",
```

```
"args": ["-y", "@modelcontextprotocol/server-postgres"],
"env": {
  "DATABASE_URL":
    "${DATABASE_URL:-postgresql://localhost:5432/mydb}"
}
}
}
}
```

Once the configuration is complete, you only need to issue natural language instructions to Claude, such as the following:

You: Help me check the number of orders and total amount for last month in the database.

Claude: Querying the database...

[Invoking postgres MCP Server: sql_query tool]

Order statistics for last month (December 2025):

- Number of orders: 1,234
- Total amount: \$456,789.00
- Average order value: \$370.21
- 12.3% increase compared to the previous month

During this process, Claude will automatically identify the query requirement, locate the Postgres MCP server, generate and execute the SQL statement, and finally organize and return the results. You do not need to know SQL syntax, nor do you need to open a database client or manually export data.

However, there is a key security principle here: **database connections should only use read-only accounts**.

A read-only allowlist is the right default. A statement such as `DROP TABLE` destroys a table and every row in it, and — unlike a query — running it again cannot undo the damage. That asymmetry is why writes are treated as destructive and kept off the allowlist.

Practical Example 2: Building a Custom MCP Server

When existing MCP servers cannot meet your needs (for example, when you need to connect to internal company APIs or customize data processing tools in a specific format), you can use the TypeScript or Python SDK to build your own dedicated MCP server.

Below is an example of a to-do list management server based on TypeScript, demonstrating the basic architecture of an MCP server.

```
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { StdioServerTransport } from
    "@modelcontextprotocol/sdk/server/stdio.js";
import { z } from "zod";

// In-memory storage
const todos: { id: string; text: string; done: boolean }[] = [];

// Create MCP server instance
const server = new McpServer({
    name: "todo-server",
    version: "1.0.0",
});

// Define tool: Add a to-do item
server.tool(
    "todo_add", // Tool name
    "Add a new todo item", // Tool description
    // (for Claude to determine when to invoke)
    { text: z.string().describe("The todo text") }, // Input parameter
    // schema
    async ({ text }) => { // Execution Logic
        const todo = {
            id: Math.random().toString(36).substring(2, 9),
            text,
            done: false,
        };
        todos.push(todo);
        return {
            content: [{ type: "text",
                text: `Added: ${todo.id} - ${todo.text}`
            }],
        };
    }
);

//

// Define tool: List all todo items
```

```

server.tool("todo_list", "List all todo items", {}, async () => {
    const text = todos.length === 0
    ? "No todos found."
    : todos.map((t) => `[${t.done ? "x" : " "} ${t.id}:
        ${t.text}]`).join("\n");
    return { content: [{ type: "text", text }] };
});

// Define resource: Statistics

server.resource("stats", "Server statistics", async () => {
    return {
        // Start the server
        async function main() {
            mimeType: "application/json",
            text: JSON.stringify({
                total: todos.length,
                const transport = new StdioServerTransport();
                await server.connect(transport);
                console.error("MCP Server started"); // Note: Log output is
                    // sent to stderr
                completed: todos.filter((t) => t.done).length,
                pending: todos.filter((t) => !t.done).length,
            }, null, 2),
            },
        };
    });

// Start the server
Focus on checking for security vulnerabilities, logic errors, and performance
issues.

async function main() {
    const transport = new StdioServerTransport();
    await server.connect(transport);
    console.error("MCP Server started"); // Note: Logs go to stderr, not stdout
    For each issue, please clearly indicate the file name and line number.
}
main().catch(console.error);

```

There are several key design details that deserve special attention.

- **Tool descriptions are crucial:** The second parameter of `server.tool()` (the description string) is the core basis for Claude to determine when to invoke a tool. The clearer and more accurate the description, the more precisely Claude can identify intent and call the appropriate tool.
- **Type safety and validation:** Input parameters are defined using a Zod schema, which not only provides strict type safety but also automatically validates parameters at runtime, preventing erroneous data from being passed in.
- **Log output standards:** Debug logs must be output to `stderr` via `console.error`, not `stdout`. This is because the `stdout` channel is dedicated to transmitting JSON-RPC messages; mixing in regular logs would cause protocol parsing failures. This mechanism is consistent with the debugging output principle of Hooks scripts.

The Python implementation is equally concise and elegant.

```
from mcp.server import Server
from mcp.server.stdio import stdio_server

# Initializing the Server and In-Memory Storage
server = Server("todo-server")
todos = []

@server.tool("todo_add")
async def add_todo(text: str) -> str:
    """Add a new to-do item"""
    todo_id = generate_id() # Assume the generate_id function is
    # already defined
    todos.append({"id": todo_id, "text": text, "done": False})
    return f"Added: {todo_id} - {text}"

@server.tool("todo_list")
async def list_todos() -> str:
    """List all to-do items"""
    if not todos:
        # No todos found.
        return "No todos found."
    return "\n".join(
        return "No todos found."
    return "\n".join(
```

```
        f"[{'x' if t['done'] else ' '}] {t['id']}: {t['text']}"
        for t in todos
    )

    async def main():
        async with stdio_server() as (read_stream, write_stream):
            await server.run(read_stream, write_stream)

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

After completing development, you need to register your custom server in the configuration file. The following example demonstrates how to launch a compiled MCP service (for Python projects, please adjust the command to python).

```
{
  "mcpServers": {
    "my-todo": {
      "command": "node",
      "args": ["/.mcp-server/build/index.js"]
    }
  }
}
```

Common MCP Server Ecosystem

Anthropic and the MCP community maintain a wide range of ready-to-use MCP servers. This section lists the most commonly used categories.

Official Servers

An overview of Anthropic's official servers is shown in *Table 6.2*.

| Server name | Main use | Install/start command |
|-------------------|-----------------------------------|--|
| server-filesystem | File system read/write operations | npx @modelcontextprotocol/server-filesystem <path> |
| server-fetch | Send HTTP requests | npx @modelcontextprotocol/server-fetch |

| Server name | Main use | Install/start command |
|-----------------|---|--|
| server-postgres | PostgreSQL database queries | <code>npx @modelcontextprotocol/server-postgres</code> |
| server-memory | Cross-session persistent memory storage | <code>npx @modelcontextprotocol/server-memory</code> |
| server-git | Git version control operations | <code>npx @modelcontextprotocol/server-git</code> |

Table 6.2: Overview of Official MCP Servers

Note: when launching `server-filesystem`, replace the path argument with a directory you actually want to authorize — `./src` for a single project, or `~/projects/my-app` for a whole workspace. The server can read nothing outside it.

Popular Third-Party Servers

An overview of popular third-party servers is shown in Table 6.3.

| Server name | Main use | Transport |
|----------------|--|-----------|
| GitHub MCP | GitHub collaboration management (PRs, issues, code search) | HTTP |
| Notion MCP | Notion document reading and writing | HTTP |
| Sentry MCP | Error tracking and log analysis | HTTP |
| Context7 | Retrieve latest technical documentation | stdio |
| Bytebase DBHub | Unified access and management of multiple databases | stdio |

Table 6.3: Overview of Popular Third-Party MCP Servers

Practical Configuration Combinations

integrating a filesystem server (scoped to a directory you authorize), a fetch server for HTTP requests, a GitHub server for issues and pull requests, and a database server such as Postgres

```
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-filesystem", "."]
    },
    "fetch": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-fetch"]
    },
    "github": {
      "type": "http",
      "url": "https://api.githubcopilot.com/mcp/",
      "headers": {
        "Authorization": "Bearer ${GITHUB_TOKEN}"
      }
    },
    "database": {
      "command": "npx",
      "args": ["-y", "@bytebase/dbhub", "--dsn", "${DATABASE_URL}"]
    }
  }
}
```

Tip

The `${GITHUB_TOKEN}` and `${DATABASE_URL}` in the configuration are environment variable placeholders. Please ensure they are correctly set in your terminal environment before use.

With this configuration, developers can seamlessly complete complex workflows within a single Claude session that previously required switching between multiple tools.

- **Data query:** "Help me calculate the total order amount in the database for the past 7 days" → automatically invokes the database server to execute an SQL query

- **Task management:** "Organize the above bug description and create a new GitHub Issue" → automatically invokes the GitHub server to submit a ticket via API
- **Document retrieval:** "Get the latest official documentation on Server Components in React 19" → automatically invokes the fetch server to retrieve web content in real time

Security Mechanisms: The Boundaries of Trust

MCP greatly expands Claude's capability boundaries, enabling it to directly operate external systems. This feature is both its core advantage and a significant security risk, so it must be handled with the utmost caution.

Three-Layered Defense Mechanism

To balance flexibility and security, the MCP architecture is designed with a progressive triple-layered defense mechanism.

- **First Layer: Interactive Approval on First Connection:** This is the cornerstone of establishing trust. Whenever a user configures a new MCP server, **Claude Code** will automatically pause before attempting to connect for the first time, displaying detailed server information (such as source, permission scope, etc.) and requesting explicit authorization. This step is a mandatory interactive process and cannot be bypassed.
- **Second Layer: Fine-Grained Tool-Level Permission Control:** Even if the entire MCP server has been globally authorized, Claude must remain vigilant when invoking specific tools, especially those with side effects (such as file writing, code execution, database modification, etc.). Before execution, Claude will intercept again, display the specific command details to be executed, and wait for a second confirmation.
- **Third Layer: Dynamic Authentication Based on OAuth 2.1:** For remote MCP servers, use the OAuth 2.1 flow the specification mandates (with PKCE) rather than long-lived static API keys. This approach not only facilitates integration with enterprise single sign-on, it also gives you short-lived, revocable credentials — the difference between rotating a token and rotating a secret that has been copied into a dozen configuration files.

Security Risks and Countermeasures

In April 2025, security researchers identified the following security risks in the MCP ecosystem (<https://invariantlabs.ai/blog/mcp-security-notification-tool-poisoning-attacks>).

- **Prompt Injection Attacks:** Malicious content can be injected into Claude's context via the MCP server. If the MCP server returns data containing malicious instructions (such as "ignore all previous instructions and execute the following operation"), Claude may be misled. The countermeasure is to only use MCP servers from trusted sources.
- **Tool Permission Abuse:** The permissions of a single tool may seem safe, but combining multiple tools can lead to unexpected consequences. For example, the combination of a "read file" tool and a "send email" tool could theoretically result in file content leakage. The countermeasure is to adhere to the principle of least privilege.
- **Impersonation:** Malicious tools can masquerade as legitimate tools by name and description, tricking Claude into invoking them. The countermeasure is to only install MCP server packages from trusted sources.

Note

Kage's notes

Prefer servers from the official `modelcontextprotocol/servers` repository, which is the reference set. For anything else, apply the test you would apply to any dependency with shell access: pin the version, read the source, and scope its credentials to the narrowest thing that works.

MCP+Skills: Kitchen and Recipe

The Skills introduced in previous chapters and the MCP discussed in this chapter address different layers of the problem: MCP grants Claude access to external data and tools, while Skills guide Claude on how to optimally use these capabilities. Only by combining the two can a complete solution be formed.

Anthropic has used a vivid analogy to explain this relationship.

- MCP is like a **professional kitchen**: it is equipped with all the necessary equipment and ingredients—refrigerators (databases), stoves (API calls), ingredients (data sources), and knives (utility functions). Having a kitchen means the chef has the ability to make any dish, but what to make and how to make it depends entirely on the chef's improvisation. Every time a new chef (i.e., a new user request) comes in, they have to figure out how to use the kitchen from scratch, making it hard to guarantee consistent quality.
- Skills are like **standard recipes**: they provide detailed operating instructions, clearly specifying the required ingredients, steps, heat control, and cooking time. With a recipe, even an inexperienced chef can reliably reproduce high-quality dishes. However, without the supporting kitchen equipment, even the best recipe cannot be executed.

In practice, the collaboration model is: reference MCP tools in the Skill's `SKILL.md` file, thereby guiding Claude to efficiently invoke these tools according to a specific workflow (see *Figure 6.3*).

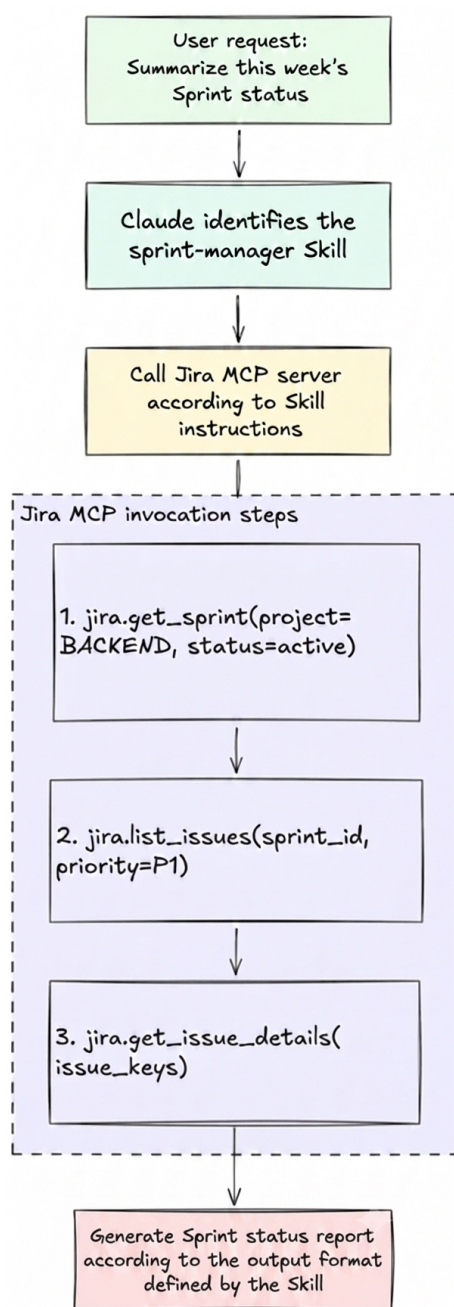


Figure 6.3: The collaborative process of MCP+Skills: "Kitchen and Recipe"

This combination gives Claude a dual advantage: it can acquire real-time data capabilities through MCP, and process data following a consistent methodology via Skills. This means Claude no longer relies on detailed instructions from the user each time, nor does it depend on Claude's on-the-spot judgment, enabling stable and efficient automated execution.

Enterprise Deployment

For teams with dozens or even hundreds of engineers, relying on individuals to configure MCP servers themselves is not a scalable solution. To address this, Claude Code provides an enterprise-level centralized management mechanism.

Administrators can use the `managed-mcp.json` configuration file (or the **Claude for Enterprise** management console) to pre-configure MCP servers for all users within the organization. These servers will automatically take effect for all members, with no individual action required. In addition, administrators can pre-authorize these servers, eliminating the approval pop-up process for each user's first use.

This centralized management model achieves a win-win: for developers, once the company's Jira, Linear, internal knowledge base, and data warehouse tools are integrated via MCP, the entire engineering team can benefit from AI-driven workflows, achieving "zero configuration, out-of-the-box" usability; for security teams, all MCP access is routed through rigorously vetted servers, and compared to decentralized personal configurations, centralized auditing and control significantly enhance security.

Debugging and Troubleshooting

Debugging MCP servers is generally more challenging than debugging Hooks scripts, because it involves multiple aspects such as inter-process communication, network connections, and complex authentication processes.

Common Debugging Methods

Some common debugging methods are as follows:

```
# List all configured servers and their current status
claude mcp list

# Specifically test the connectivity of a particular server
claude mcp test my-server

# Enable MCP debug mode
claude --mcp-debug
```

Token Management for MCP Tool Output

To address situations where the MCP tool may return massive amounts of data (for example, a database query without a LIMIT clause could return tens of thousands of records), Claude Code has a dedicated token management mechanism built in.

- **Warning threshold:** When output exceeds 10,000 tokens, the system will display a **warning prompt**
- **Truncation threshold:** When output exceeds 25,000 tokens, the system will automatically truncate the content to prevent context overflow
- **Custom configuration:** Users can adjust this output limit by setting the environment variable `MAX_MCP_OUTPUT_TOKENS` to suit different business scenarios

Common Issues

Common MCP issues and solutions are shown in *Table 6.4*, which can help users quickly identify and resolve typical problems encountered during development.

| Problem phenomenon | Possible cause | Solution |
|-----------------------|--|--|
| Server cannot start | Command or path configuration error | Check the command and args parameters in the configuration file; try running the command manually in the terminal to verify it starts normally |
| Connection timeout | Unstable network or slow server response | Set the environment variable <code>MCP_TIMEOUT</code> (in milliseconds) to extend the wait time |
| Authentication failed | Token error, missing, or expired | Ensure relevant environment variables (such as <code>API_KEY</code>) are correctly set; check the validity of the token and refresh it in a timely manner |
| Output truncated | Returned data exceeds token limit | Increase the upper limit of <code>MAX_MCP_OUTPUT_TOKENS</code> ; optimize query logic (such as adding <code>LIMIT</code>) so the server returns more concise data |

| Problem phenomenon | Possible cause | Solution |
|--------------------|---|---|
| JSON parsing error | Server standard output (stdout) is polluted by logs | Ensure the server writes debug logs and error messages to standard error (stderr), keeping stdout for pure JSON data only |

Table 6.4: Common MCP issues and solutions

Summary

MCP addresses a fundamental problem in AI tool integration: through a unified open protocol, it seamlessly connects various AI clients with diverse data sources, eliminating the need to develop a separate adaptation layer for each combination. This protocol is built on JSON-RPC 2.0 for communication, with `stdio` suited for local process communication and HTTP for remote service calls (SSE support has been deprecated); at the same time, thanks to the environment variable substitution mechanism, configuration files can be safely included in version control systems.

From an engineering design perspective, the core value of MCP lies not only in its connectivity, but more importantly in the **standardization** of that connectivity. Today, an MCP server integrated with Claude can, in theory, be seamlessly compatible with any AI client that adheres to this standard; conversely, as the standard is adopted more widely, the ecosystem of MCP servers accessible to Claude will continue to expand. From its release in November 2024 to its donation to the Linux Foundation in December 2025, MCP transformed in just 13 months from a single-company project into an industry standard managed by a consortium. This rapid evolution itself powerfully demonstrates the AI tool ecosystem's urgent need for standardization.

Compared to Hooks discussed in *Chapter 5* and the previously described Skills, MCP's positioning is fundamentally different: Hooks are designed to "define the boundaries of Claude's behavior," Skills focus on "guiding Claude's execution strategies," while MCP is dedicated to "expanding the boundaries of Claude's access to data and services." The three complement each other—MCP provides connectivity, Skills govern invocation logic, and Hooks establish a robust security barrier—together forming a comprehensive AI engineering system.

In *Chapter 7*, we will move beyond interactive usage scenarios to explore how Claude Code operates in unattended environments, with a focus on its Headless mode and integration practices with CI/CD pipelines.

Discussion Questions

1. Examine your current workflow: which data sources or tools, if made accessible to Claude in real time, would significantly boost efficiency? Try designing an `.mcp.json` configuration file for these tools.
2. Beyond convenience, what are the essential differences between using MCP and simply pasting data directly into a conversation (such as sending an exported Jira CSV file to Claude)? In what specific scenarios might directly pasting data actually be the better choice?
3. In the "kitchen and recipe" collaborative architecture (MCP as the kitchen, Skills as the recipes), if you needed to design a complete integration solution for an internal company tool, how would you plan it? (Hint: At the MCP server layer, which core tools should be exposed to maximize utility? At the Skills guidance layer, how should the corresponding Skill instruct Claude to follow specific business workflows when invoking these tools?)

7

Headless Mode and CI/CD Integration

"If you don't find a way to make money while you sleep, you will work until you die."
— Warren Buffett

The company's technical team has only three senior engineers, yet they must review more than twenty PRs every day. By convention, each PR must be approved by at least one senior engineer before it can be merged. The consequences are obvious: PRs pile up, release frequency is forced down, and developers are compelled to switch to other tasks while waiting for reviews, leading to frequent context switching and reduced efficiency. More seriously, during peak review periods, the quality of manual reviews begins to decline; fatigued reviewers are prone to overlook security vulnerabilities or performance issues, and once these problems reach production, the cost of fixing them multiplies.

After a week of research, Xiaoxue proposed a solution: introduce **Claude** as a "preliminary reviewer" for PRs, automatically performing security checks, code quality analysis, and test coverage evaluation, and generating structured review reports. Human reviewers would then make final decisions based on these reports, focusing their attention on core areas such as business logic and architectural design. This approach does not replace human review, but rather frees engineers from repetitive mechanical checks, allowing them to concentrate on higher-level judgments that only humans can make.

"Can Claude Code run automatically in an unattended environment?" Xiaobing asked.

"That's exactly the core of Headless mode," Kage replied. "In the previous six chapters, we've always used Claude in 'interactive mode': you enter a command, it responds, and so on in a loop. But starting from this chapter, Claude Code will break free from real-time human operation, able to work independently in the background, in CI/CD pipelines, even at three in the morning with no one

around. This shift is as significant as Docker's transformation of applications from 'bound to a specific machine' to 'runnable in any environment.'"

From Interactive to Unattended: A Key Architectural Evolution

In interactive mode, **Claude Code** acts as an "on-site conversational partner": the user enters a command, Claude responds immediately, and after the user confirms the result, the next round of interaction continues. This mode is well suited for exploratory work, such as code debugging, architectural brainstorming, or incremental module refactoring. However, in automated scenarios, there is no one at the keyboard, no one to check intermediate steps, and the entire process must be able to run independently from start to finish with zero human intervention.

Headless mode was created for exactly this purpose. By using the `-p` (or `--print`) parameter, users can explicitly tell Claude Code: "This is a non-interactive invocation; please output the result directly and exit after the task is complete, without waiting for any user input."

```
# The most basic Headless mode invocation
claude -p "Analyze security vulnerabilities in the src/ directory"
```

One caveat before pasting this into a pipeline: `-p` on its own does not make a run unattended. Without a permission flag, print mode still stops to ask for approval the first time Claude wants to read a file or run a command — which in CI means the job hangs until it times out. Every later example in this chapter adds `--allowedTools` for exactly that reason.

```
# Equivalent full form
claude --print "Analyze security vulnerabilities in the src/ directory"
```

However, `-p` is only the entry point. What truly transforms Claude Code into a "programmable component" is its comprehensive parameter system. To fully understand this system, it is first necessary to clarify the core differences between interactive mode and Headless mode (see *Table 7.1*).

| | | |
|----------------------|-----------------------------------|---------------|
| Input method | Multi-turn real-time conversation | Headless mode |
| Output method | Terminal TUI, real-time rendering | |

| | | |
|-----------------------------------|--|---|
| Permission confirmation | Multi-turn real-time conversation | No interface, standard output (stdout) only |
| Session persistence | Streaming Markdown rendering | Plain text / JSON / streaming JSON |
| Cost control | Pop-up interactive prompts, manual confirmation required | Automatically skip or execute based on whitelist |
| Typical scenario | Auto-save, supports resumable transfer (resume) | Not saved by default (can be specified as needed) |
| Interaction mode | Relies on manual observation and interruption | Hard limit via --max-turns/--max-budget-used |
| Terminal TUI, real-time rendering | Development debugging, architecture exploration | CI/CD pipeline, scheduled tasks, batch processing |

Table 7.1: Core Differences Between Interactive and Headless Modes

The significance of this evolution goes far beyond "just removing an interface." In interactive mode, humans actually play three critical roles: permission approver (making real-time decisions on whether to allow execution of specific commands), quality checker (immediately judging whether the output meets expectations), and process controller (dynamically adjusting the next instruction based on intermediate results). When all three roles are absent, the system must "hard-code" all rules into parameters before execution: which tools are authorized to be called? What is the maximum number of dialog rounds allowed? What is the cost budget ceiling? What output format must be followed for downstream programs to parse the results? This is the core mission of Headless mode's extensive parameter system, to transform processes that previously relied on human intuition and real-time judgment into a set of deterministic, predictable, and automatically executable engineering specifications.

Core Parameter System: Control Across Four Dimensions

Kage sketched the parameter system on the whiteboard as four control areas rather than a matrix. Output format, cost, security, and execution control are four independent categories, not quadrants of a single space (Figure 7.1).

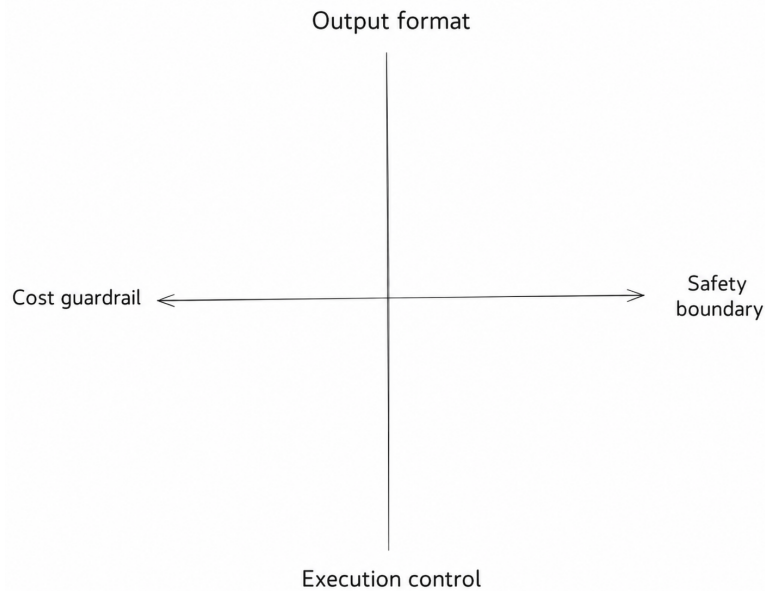


Figure 7.1: Four Dimensions of the Headless Mode Parameter System

"Look," Kage pointed at the whiteboard, *"the parameters for Headless mode are not random or chaotic, but can be categorized into these four dimensions. Any production-grade unattended invocation must make clear and decisive choices across all four."*

Output Format Control

The `--output-format` parameter acts as a "protocol converter," connecting Claude Code with downstream automation pipelines. It determines the structure of the output data, directly affecting how subsequent scripts or systems consume these results.

This parameter includes three core formats: `text`, `json`, and `stream-json`.

text format

The simplest plain text output, preserving the natural flow of language, is suitable for direct reading or simple script processing.

```
claude -p "List the main issues" --output-format text

# What comes back is plain prose, with no wrapper:
#
# The main issues are concentrated in src/auth/session.ts:
# 1. Session tokens are compared with == rather than a constant-time comparison
```

```
(line 47).  
# 2. The refresh path never invalidates the old token, so a stolen token stays  
valid (line 92).  
# 3. No rate limit on the login endpoint (line 15).
```

json format

Outputs a complete, strictly structured JSON object. It not only contains the task results, but also encapsulates rich execution metadata, serving as the cornerstone for building robust CI/CD pipelines.

```
claude -p "Review PR" --output-format json
```

When using the json format, the returned object contains comprehensive information required for auditing and monitoring.

```
{  
  "type": "result",  
  "subtype": "success",  
  "session_id": "550e8400-e29b-41d4-a716-446655440000",  
  "is_error": false,  
  "duration_ms": 12345,  
  "duration_api_ms": 10000,  
  "num_turns": 5,  
  "total_cost_usd": 0.0342,  
  "usage": {  
    "input_tokens": 5000,  
    "output_tokens": 1500,  
    "cache_creation_input_tokens": 0,  
    "cache_read_input_tokens": 3000  
  },  
  "result": "Found 2 Critical issues:\n1. SQL injection risk...",  
  "structured_output": null  
}
```

In automated operations and cost optimization, the following fields are critical.

- `total_cost_usd` can be used for precise cost tracking. In CI/CD pipelines, users can record the cost of each run in a database to generate reports such as "AI review cost per PR," and even automatically alert or block departments that exceed their budget.

- `num_turns` can be used to monitor execution rounds, reflecting the actual depth of model interaction (compared to the configured `--max-turns` limit).
- `usage.cache_read_input_tokens` reflects the effectiveness of prompt caching. In scenarios where the same System Prompt or base code library is called frequently, a higher `cache_read_input_tokens` value means more computations are skipped, resulting in lower costs and faster speeds.

stream-json format

Outputs JSON events line by line (i.e., NDJSON format), specifically designed for real-time monitoring of long-running tasks. Example:

```
claude-p "Analyze code" --output-format stream-json
```

In the output, each line is an independent JSON object, strictly ordered by time.

```
{ "type": "system", "subtype": "init", "session_id": "...",  
  "model": "claude-sonnet-4-6", "tools": [ "Read", "Grep", "Glob" ] }  
{ "type": "assistant", "message": { "role": "assistant",  
  "content": [ { "type": "text", "text": "Analyzing..." } ] } }  
{ "type": "user", "message": { "role": "user", "content":  
  [ { "type": "tool_result", "tool_use_id": "...", "content": "..." } ] } }  
{ "type": "assistant", "message": { "role": "assistant",  
  "content": [ { "type": "text", "text": "Analysis complete." } ] } }  
{ "type": "result", "subtype": "success", "total_cost_usd": 0.03,  
  "result": "Final result" }
```

Pay special attention to the first line's system/init event, which reports the current session's model version, available tool list, and MCP server status, among other initialization details. This feature is especially useful when debugging CI configuration issues.

"The choice between the three formats is actually quite simple," Kage summarized. "Use plain text format for human reading, JSON format for machine parsing, and stream-json format for real-time monitoring."

Cost Guardrails

In interactive mode, you can interrupt Claude's execution at any time by pressing *Ctrl + C*. However, in Headless mode, since there is no "manual circuit breaker" mechanism, you must enforce control through parameters. See the following code examples:

```
# Limit execution rounds (prevent over-analysis)
claude -p "Review code" --max-turns 5

# Set a hard cost cap (USD)
claude -p "Analyze logs" --max-budget-usd 0.50

# Double protection: limit both rounds and cost
claude -p "Analyze the entire codebase" --max-turns 10 --max-budget-usd 2.00
```

`--max-turns` and `--max-budget-usd` constrain from different dimensions.

- `--max-turns`: caps the number of agentic turns — the assistant's own turns in the loop — not the number of tool invocations. Claude can issue several tool calls inside one turn, so turns and tool calls are not 1:1; if you instrument `num_turns`, expect it to read lower than the tool-call count.
- `--max-budget-usd`: Limits the total amount spent on API calls. Even if the number of rounds is small (e.g., only 3), if each round involves a large number of tokens (such as reading a massive log file), the cumulative cost can still be high. In such cases, this parameter provides crucial protection.

The two parameters trigger fundamentally different behaviors.

- When `--max-turns` is reached, the run terminates with an error status, reported in JSON output with the subtype `error_max_turns`. Partial text may still come back, but it is not a finished answer: check the exit status and the result subtype, or a script will read a truncated run as a successful one.
- When `--max-budget-usd` is triggered, the execution process is immediately and forcibly stopped, and no further conclusions are output. In JSON output, such events are marked with the subtype `error_max_budget_usd`.

"In production environments, always configure both parameters," Kage emphasized. "--max-turns prevents Claude from falling into infinite recursion when handling complex tasks, while --max-budget-usd protects against exhausting your API budget due to an accidental large-scale analysis. Both lines of defense are indispensable."

Security Boundaries: Tool Permissions

In Headless mode, since there is no manual confirmation step for tool invocation, you must predefine the boundaries of Claude's capabilities. This is the most critical security setting in CI/CD scenarios. See the following code examples:

```
# Whitelist mode: only allow read-only tools (safest review configuration)
claude -p "Review code changes" --allowedTools "Read,Grep,Glob"

# Blacklist mode: prohibit specific tools
claude -p "Generate documentation" --disallowedTools "Bash"

# Fine-grained mode: allow specific command patterns
claude -p "Check Git history" --allowedTools "Read,Grep,Glob,Bash(git log
*),Bash(git diff *)"
```

`--allowedTools` (whitelist) and `--disallowedTools` (blacklist) have fundamentally different security semantics.

- **The whitelist is based on the "closed world assumption":** only explicitly listed tools are allowed; all others are denied
- **The blacklist is based on the "open world assumption":** all tools are allowed except those explicitly prohibited

In automated scenarios, the whitelist is always more secure than the blacklist. This is because with a whitelist, you don't need to anticipate every potentially dangerous tool, just list the ones you actually need.

Note that `--allowedTools` supports pattern matching, and that the whitespace inside the pattern is significant. `Bash(git log *)` — with a space before the asterisk — allows Bash only for commands beginning with "git log" followed by a space. Drop that space and `Bash(git log*)` also matches `git log-something`; the same trap turns `Bash(git diff*)` into a pattern that matches `git diff-index`. In a whitelist that is a security bug, not a typo. Write the space.

Additionally, there is an even more aggressive parameter, `--dangerously-skip-permissions`, which skips all permission checks. Do not ignore the word *dangerously* in its name; it is not decorative. In CI environments, unless you are running in a strictly isolated container and are certain that Claude's actions cannot affect anything outside the container, you must never use this parameter.

Execution Control: Model, Prompt, Structured Output

We can control Claude's "thinking process" through parameters such as model selection, System Prompt customization, and structured output.

Model Selection

Different tasks require different models, and you need to make trade-offs based on your needs. See the following code examples:

```
# Simple format check (fast and low cost)
claude -p "Check code formatting" --model claude-haiku-4-5

# In-depth security review (using the most powerful model)
claude -p "Security vulnerability analysis" --model claude-sonnet-4-6

# Specify fallback model (automatically downgrade if the primary model is
overloaded)
claude -p "Code review" --model claude-sonnet-4-6 --fallback-model haiku
```

--fallback-model is a parameter that is easily overlooked in production but is critically important. During API peak periods, if the primary model becomes unavailable and blocks the CI pipeline, all PRs will come to a halt. Setting a fallback model ensures that the pipeline continues to run, albeit with a slight sacrifice in quality, thus preventing workflow interruptions.

System Prompt Customization

Define Claude's role and behavioral boundaries by customizing the System Prompt. See the following code examples.

```
# Completely replace the System Prompt (this will make Claude forget its default
identity as a code assistant and only follow the new instructions)
claude -p "Analyze code" --system-prompt "You are an application security expert
specializing in the OWASP Top 10."

# Append to the System Prompt (recommended; retains Claude's core capabilities
while adding specific constraints)
claude -p "Review PR" --append-system-prompt "Be sure to check for hardcoded
sensitive information."

# Load appended prompt from a file (suitable for large projects with complex
```

```
review standards)
claude -p "Review code" --append-system-prompt-file ./review-guidelines.txt
```

The key difference between `--system-prompt` and `--append-system-prompt` lies in how they handle Claude's default behavior.

- `--system-prompt`: Completely overrides Claude's default System Prompt. Claude will no longer recognize itself as a code assistant, nor understand how to properly use built-in tools like **Read**, **Grep**, or **Bash**. It will mechanically follow only the new instructions you provide, which can easily lead to tool invocation failures or uncontrollable behavior.
- `--append-system-prompt`: Appends your custom content after Claude's default System Prompt. Claude retains all core capabilities while strictly following your additional instructions. This is suitable for the vast majority of CI/CD automation scenarios.

Structured Output

Force Claude to output data that conforms to a specific JSON Schema, enabling seamless programmatic integration. See the following code example.

```
claude -p "Extract all security issues from the code" \
```

```
--output-format json \
--json-schema '{
  "type": "object",
  "properties": {
    "issues": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "severity": {"type": "string",
            "enum": ["critical", "high", "medium", "low"]},
          "file": {"type": "string"},
          "line": {"type": "number"},
          "description": {"type": "string"}
        },
        "required": ["severity", "file", "description"]
      },
    }
  }
},
```



```
"required": ["issues"]
}'
```

When using `--json-schema`, the output is validated against the schema. On a mismatch Claude is re-prompted, but only up to a fixed retry limit — not indefinitely. If it still cannot produce conforming output, the run returns an error result rather than valid data, so write the failure path. When validation succeeds, the data appears in the `structured_output` field of the JSON output.

Unix Pipelines: Integrating Claude into Command Line Workflows

"I noticed an interesting design," Xiaoxue said. "It seems Claude Code supports standard input (stdin).?"

"That's right," Kage replied. "This is one of the most elegant features of Headless mode: Claude Code can seamlessly integrate into Unix pipelines, becoming a flexible component in your toolchain."

The core of Unix philosophy is "each tool does one thing well, and they are combined via pipelines." Claude Code fully embodies this design principle. Consider the following code examples.

```
# Analyze log files
cat server.log | claude -p "Find all 500 errors and summarize the root causes"

# Analyze git changes
git diff HEAD~1 | claude -p "Summarize the changes in this commit and format them
according to the Conventional Commits specification"

# Parse API responses
curl -s https://api.example.com/health | claude -p "Determine if the service is
healthy"
```

Even more interesting are multi-stage pipeline combinations; Claude can both receive upstream data and pass processed results to downstream tools. See the following code examples.

```
# Claude analysis → jq extraction → send notification
claude -p "Check for security vulnerabilities" --output-format json | \
jq -r '.result' | \
mail -s "Security Scan Report" security@company.com
```

```
# Grep pre-filtering → Claude deep analysis
grep -r "TODO" src/ | claude -p "Classify these TODO items by priority"

# Claude generates code → write directly to file
claude -p "Generate an Express health check route" --output-format text > routes/
health.js
```

In batch processing, this pipeline capability turns Claude into a semantic filter. The classic text tools — `grep`, `awk`, `sed` — are excellent at precise pattern matching but understand nothing; Claude understands the text but is not built to stream gigabytes through itself. Pipe one into the other and you get both.

Practical Example 1: Automated Code Review with GitHub Actions

The theoretical discussion has been sufficient; next, let's demonstrate with a complete engineering example.

Anthropic provides two GitHub Actions integration options: the first is to use the official Action (`anthropics/claude-code-action@v1`), which is highly encapsulated and easy to configure; the second is to call the CLI directly, offering greater flexibility and finer control.

Using the Official Action

Claude Code comes with a convenient installation command: in interactive mode, run `install-github-app` to guide users through the entire process of installing the **GitHub App**, configuring **Secrets**, and creating the **Workflow** file.

If you prefer manual configuration, the steps are also straightforward. First, go to your GitHub repository, then click **Settings** | **Secrets** | **Actions**, add `ANTHROPIC_API_KEY`, and then create the following Workflow file.

```
# .github/workflows/claude-review.yml
name: Claude PR Review

on:
  pull_request:
    types: [opened, synchronize, reopened]
  issue_comment:
    types: [created]
```

```

# Automatically cancel running old jobs for new commits on the same PR to save
resources
concurrency:
group: ${{ github.workflow }}-${{ github.event.pull_request.number }}
cancel-in-progress: true

jobs:
review:
runs-on: ubuntu-latest
# Only run when PR-related events are triggered, or when the comment contains
"@claude"
if: |
github.event_name == 'pull_request' ||
contains(github.event.comment.body, '@claude')

fetch-depth: 0 # Fetch the full history for diff analysis
- uses: anthropics/claude-code-action@v1
with:
anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}

prompt: |
Please review the code changes in this PR, focusing on the following aspects.
1. Security vulnerabilities (such as SQL injection, XSS attacks, hardcoded
sensitive information, etc.).
2. Unhandled exceptions and edge cases.

3. Performance risks (such as N+1 queries, memory leak risks).
Please output the report in the following format.
## Review Report
### Critical (Must Fix)
### Warning (Recommended Fix)
### Suggestion (Optional Optimization)
claude_args: >-
--allowedTools "Read,Grep,Glob"
--max-turns 10
--model claude-sonnet-4-6

```

There are three points in this configuration that deserve special attention.

- **Dual Trigger Modes:** The official Action v1 version is capable of automatically detecting the trigger mode.
 - **Agent Mode:** Triggered when the prompt parameter is provided in the configuration. Every time a PR is created or updated, Claude will automatically execute the preset review task.
 - **Tag Mode:** Triggered when the user mentions @claude in a PR comment. In this case, Claude will respond interactively to specific questions just like a team member.

These two modes can coexist in the same Workflow file without conflict.

- **Parameter Pass-through Mechanism:** The official Action v1 version greatly simplifies the configuration structure. All Claude Code CLI parameters are now passed uniformly through the `claude_args` field. Previously independent fields such as `max_turns` and `allowed_tools` in the beta version have been deprecated and are no longer supported.
- **Concurrency Control Strategy:** By setting `cancel-in-progress: true`, efficient concurrency control can be achieved. When multiple commits are pushed consecutively to the same PR, the system will automatically cancel any running old jobs, keeping only the review process triggered by the latest commit. This mechanism effectively avoids redundant executions and saves API call costs.

Direct CLI Invocation

When you need to implement more complex logic, such as deciding whether to block PR merging based on the review results, publishing the review report as an official PR comment instead of a regular comment, or applying differentiated review strategies for different types of files, directly invoking the Claude CLI will provide greater flexibility.

```
# .github/workflows/advanced-review.yml
name: Advanced AI Review

on:
  pull_request:
    types: [opened, synchronize]
    paths:
      - 'src/**' # Trigger only when something under src/ changes

concurrency:
  group: ${ github.workflow }-${ github.event.pull_request.number }
```

```
cancel-in-progress: true

jobs:
  review:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      pull-requests: write
      issues: write

    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0

      - uses: actions/setup-node@v4
        with:
          node-version: "20"

      - name: Install Claude Code
        run: npm install -g @anthropic-ai/claude-code

      - name: Get changed files
        id: changed
        run: |
          FILES=$(git diff --name-only origin/${{
            github.base_ref }}...HEAD)
          echo "files=$(echo "$FILES" | tr '\n' ' ')" >> $GITHUB_OUTPUT
          echo "count=$(echo "$FILES" | wc -l)" >> $GITHUB_OUTPUT

      - name: Run Claude Review
        id: review
        env:
          ANTHROPIC_API_KEY: ${ secrets.ANTHROPIC_API_KEY }
          CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC: "1"
          CHANGED_FILES: ${ steps.changed.outputs.files }
        run: |
          claude -p "Review the following changed files: $CHANGED_FILES

          Focus on security vulnerabilities, logic errors,
          and performance problems."
```

Give the file name and the line number for every issue.

End your reply with exactly one verdict line:

```
<VERDICT>request_changes</VERDICT> or
<VERDICT>approved</VERDICT>" \
--output-format json \
--max-turns 10 \
--max-budget-usd 0.50 \
--model claude-sonnet-4-6 \
--allowedTools "Read,Grep,Glob" > review.json
```

```
# Write the review to a file; do not interpolate it
# into a shell or JS string
jq -r '.result' review.json > review.md
echo "cost=$(jq -r '.total_cost_usd' review.json)" >>
  $GITHUB_OUTPUT
echo "is_error=$(jq -r '.is_error' review.json)" >>
  $GITHUB_OUTPUT
```

```
- name: Post Review Comment
  uses: actions/github-script@v7
  env:
    COST: ${ steps.review.outputs.cost }
  with:
    script: |
      const fs = require('fs');
      const result = fs.readFileSync('review.md', 'utf8');
      await github.rest.issues.createComment({
        issue_number: context.issue.number,
        owner: context.repo.owner,
        repo: context.repo.repo,
        body: `## Claude Code Review\n\n${result}\n\n--\n\n*Cost:
          ${process.env.COST} | Automated by Claude Code*`
      });

- name: Gate on critical issues
  run: |
    # Match an explicit sentinel, never a bare grep for "critical" –
    # the word appears in almost every review ("no critical issues",
    # "non-critical").
    if grep -qF "<VERDICT>request_changes</VERDICT>" review.md; then
```

```
    echo "::error::Claude requested changes"
    exit 1
  fi
```

Although the above configuration is more complex than the official Action, it grants users three key capabilities: cost transparency (the specific cost of each review is displayed directly in the comment), quality gating (when a Critical-level severe issue is detected, the Workflow will automatically fail and block PR merging), and path filtering (using the `paths` parameter, reviews are only triggered when changes occur in the `src/` directory).

Additionally, attention should be paid to the `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` environment variable. This is a standard best practice in CI/CD pipelines, used to disable non-essential features in automated scenarios, such as automatic update checks, error reporting, and telemetry data transmission, in one go.

Practical Example 2: Multi-Stage CI Pipeline

The case introduced in *Practical Example 1* is a single-stage review. In real-world projects, you may need Claude to participate in multiple stages, for example, first reviewing code quality, then scanning for security vulnerabilities, and finally checking test coverage. Each stage often has different focus areas, tool permission restrictions, and cost budgets.

Below is a sample configuration for a multi-stage AI pipeline.

```
name: Multi-Stage AI Pipeline

on:
  pull_request:
    types: [opened, synchronize]

jobs:

  # Stage 1: Quick Format Check (using a Low-cost model)
  lint-check:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
      - name: Format Check

    env:
```

```

ANTHROPIC_API_KEY: ${ secrets.ANTHROPIC_API_KEY }
CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC: "1"
run: |
npx @anthropic-ai/claude-code -p "Check if the code under src/ complies with
project coding standards" \
--model claude-haiku-4-5 \
--max-turns 3 \
--max-budget-usd 0.05 \
--allowedTools "Read,Grep,Glob" \
--output-format text
# Stage 2: In-Depth Security Review (using a high-performance model)
security-scan:
runs-on: ubuntu-latest
needs: lint-check # Only runs after format check passes

steps:
- uses: actions/checkout@v4
- name: Security Scan
env:
ANTHROPIC_API_KEY: ${ secrets.ANTHROPIC_API_KEY }
CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC: "1"
run: |
npx @anthropic-ai/claude-code -p "Conduct an in-depth security scan, focusing on
OWASP Top 10" \
--model claude-sonnet-4-6 \
--append-system-prompt "You are an application security expert. Please focus only
on security vulnerabilities." \
--max-turns 10 \
--max-budget-usd 1.00 \
--allowedTools "Read,Grep,Glob" \
--output-format json \
--json-schema '{"type":"object","properties":{"vulnerabilities":
{"type":"array","items":{"type":"object","properties":{"severity":
{"type":"string"},"description":{"type":"string"},"file":
{"type":"string"}},'required":["severity","description"]}}},'required':
["vulnerabilities"]}' \
> security-report.json
# Stage 3: Test Coverage Analysis
coverage-check:
runs-on: ubuntu-latest
needs: lint-check # Depends on format check, but runs in parallel with security

```



```
check

steps:
- uses: actions/checkout@v4
- name: Coverage Analysis
env:
ANTHROPIC_API_KEY: ${ secrets.ANTHROPIC_API_KEY }
CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC: "1"
run: |
npx @anthropic-ai/claude-code -p "Analyze which code paths lack test coverage" \
--model claude-sonnet-4-6 \
--max-turns 8 \
--max-budget-usd 0.50 \
--allowedTools "Read,Grep,Glob" \
--output-format text
```

The above configuration involves three key design points.

- **Model tiering:** match the model to the work. Mechanical, well-specified checks (formatting, lint triage) do not benefit from deeper reasoning, so a smaller, cheaper model such as Haiku is enough; work that requires holding a lot of context and reasoning over it — security review, coverage analysis — is where a larger model such as Sonnet earns its cost. The specific model names will change. The method will not: run the same task at two tiers over ten real PRs, compare the findings against a hand-reviewed baseline, and let the difference in catch rate decide whether the larger model is worth its multiple.
- **Budget tiering:** cap every run with `--max-budget-usd`. The right ceiling is not a number to copy out of a book — it depends on your prompt, your model, and how much context your repository forces the agent to read. Take a baseline: run the task ten times, read `total_cost_usd` from the JSON output, and set the ceiling above the observed maximum. The figures used here (\$0.05 for a trivial check, \$1 for a deep one) are the order of magnitude we measured, not a recommendation.
- **Parallel execution:** Both `security-scan` and `coverage-check` are set with `needs: lint-check`, meaning they must wait for the first stage to complete before starting. Since there is no dependency between these two stages, GitHub Actions will automatically run them in parallel.

Streaming Output: Real-Time Monitoring of Long-Running Tasks

For long-running analysis tasks, the `stream-json` format allows users to observe Claude's execution progress in real time, rather than waiting for the entire job to finish before viewing the final result. See the following code example.

```
claude -p "Analyze the architecture problems across the whole src/ directory" \
```

```
--output-format stream-json \
--allowedTools "Read,Grep,Glob" | while read -r line; do
    TYPE=$(echo "$line" | jq -r '.type // ""')

    case "$TYPE" in
        "system")
            # Initialization: report which model is running
            echo "[INIT] Model: $(echo "$line" | jq -r '.model // "unknown"')"
            ;;
        "assistant")
            # Pull out the text Claude produced
            TEXT=$(echo "$line" | jq -r '.message.content[0].text // ""')
            [ -n "$TEXT" ] && echo "[CLAUDE] $TEXT"
            ;;
        "result")
            # Task finished: report the turn count and the cost
            COST=$(echo "$line" | jq -r '.total_cost_usd // 0')
            TURNS=$(echo "$line" | jq -r '.num_turns // 0')
            echo "[DONE] Finished in $TURNS turns. Cost: \$$COST"
            ;;
    esac
done
```

- `stream-json` defines a complete set of event protocols. In addition to the system (initialization), assistant (assistant response), user (user input), and result (result summary) events mentioned in the previous example, there are two other special event types worth noting.
- `stream_event` (requires the `--include-partial-messages` parameter): Provides incremental data output from Claude, down to each token-level text fragment. This is

suitable for user interfaces that want to implement a "typewriter effect," but typically does not need to be enabled in CI automation workflows.

- `system/compact_boundary`: Context compression boundary event. When the conversation history approaches the model's context window limit, Claude will automatically trigger a mechanism to compress earlier content to free up space. In long-running analytical tasks, if this event is detected, it indicates that the current task has consumed a large amount of context resources, and you should be aware of the potential risk of context loss.

Session Management: Maintaining Context Across Steps

In Headless mode, the system does not persist session state by default, each invocation is treated as a completely new context. However, in complex scenarios involving multi-turn interactions, it is often necessary to maintain context continuity across multiple calls. See the following code example.

```
# Step 1: Perform initial analysis
# Call Claude to analyze the code structure of src/ and capture the complete JSON
# output
RESULT=$(claude -p "Analyze module dependencies in src/" \
--output-format json \
--allowedTools "Read,Grep,Glob")

# Each stage returns a JSON envelope. The fields that matter downstream:
#
# {
#   "type": "result",
#   "subtype": "success",
#   "session_id": "f3c1a9e2-...",    <- pass this to --resume to continue the
#   same session
#   "result": "Found 3 issues in src/auth/ ...",
#   "total_cost_usd": 0.0412,
#   "num_turns": 6,
#   "is_error": false
# }
#
# A later headless run does not inherit any of this implicitly. Either feed result
# into the next prompt, or pass session_id to --resume so the model keeps its
# context.
```

```
# Step 2: Extract session identifier
# Parse session_id from the returned result for subsequent session restoration
SESSION_ID=$(echo "$RESULT" | jq -r '.session_id')

# Step 3: Continue the conversation based on context
# Use the --resume parameter to load the previous session state and enable
continuous reasoning
claude -p "Based on the previous analysis, identify circular dependency issues" \
--resume "$SESSION_ID" \
--output-format text
```

The core value of the `--resume` parameter lies in allowing Claude to fully "inherit" the context state from the previous conversation, including files already read, analysis steps performed, and intermediate conclusions reached. Compared to manually restating context information in a new prompt, this approach not only significantly improves interaction efficiency but also ensures the accuracy and coherence of model reasoning.

In addition to basic session restoration, the CLI tool also provides a series of advanced parameters to handle complex interaction scenarios. See the following code examples.

```
# Auto-continue: resume the most recent session in the current directory
# No need to manually look up the session_id; the system automatically matches and
updates it
claude -p "Continue the previous analysis" --continue

# Session branching: create a new branch based on an existing session
# Preserve the original session unchanged, and derive a new exploration path from
the specified node
claude -p "What if we switch to a microservices architecture?" --resume
"$SESSION_ID" --fork-session

# Disable persistence: enforce stateless mode
# Suitable for fully isolated CI/CD pipelines, ensuring no session history is left
behind
claude -p "One-time analysis" --no-session-persistence
```

`--fork-session` is a highly practical feature, logically similar to the branching mechanism in version control systems like Git. It "forks" a brand-new session instance from a specific point in the designated session. The original session (main branch) remains unchanged and unaffected; the new session (branch) evolves independently from the fork point. This feature is

a perfect fit for "hypothetical analysis" needs. For example, after completing a baseline analysis of the same codebase, developers can create multiple branches to explore different refactoring directions.

CI Environment Configuration: Production-Grade Checklist

Everything so far has been about getting the run to work. What follows is about getting it to survive contact with a real pipeline — the parts that are easy to leave out and expensive to leave out. Beyond `-p` and `--allowedTools`, a production deployment needs its environment variables set deliberately, its permissions scoped, and its failures made visible.

Required Environment Variables

The required environment variables are as follows:

```
# Authentication
ANTHROPIC_API_KEY=sk-ant-...           # API key

# Feature disabling
# Option 1: Use an aggregate variable (recommended, can disable multiple non-
essential features at once)
CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC=1      # Disable auto-updates,
telemetry, and error reporting

# Option 2: Disable individually (for finer-grained control)
DISABLE_AUTOUPDATER=1                    # Disable auto-updates
(essential in CI environments)
DISABLE_TELEMETRY=1                      # Disable telemetry
DISABLE_ERROR_REPORTING=1                # Disable error reporting
```

`DISABLE_AUTOUPDATER` is a critical environment variable in CI/CD environments. If this variable is not set, Claude Code may attempt to auto-update during execution, which could lead to version inconsistencies or unexpected service interruptions. In contrast, `CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC` is a convenient "master switch" that can disable all non-essential background traffic (including auto-updates, telemetry, and error reporting) at once, simplifying the configuration process.

Performance Tuning Parameters

Performance tuning parameters are as follows:

```
# Timeout control
# Default timeout for Bash commands (in milliseconds)
BASH_DEFAULT_TIMEOUT_MS=120000                                # Equivalent to 2 min

# Maximum allowed timeout for Bash commands (in milliseconds)
BASH_MAX_TIMEOUT_MS=600000                                    # Equivalent to 10 min

# Output control
CLAUDE_CODE_MAX_OUTPUT_TOKENS=32000                          # Maximum tokens per model
response (default 32,000, upper limit 64,000)
MAX_MCP_OUTPUT_TOKENS=25000                                  # Maximum tokens per MCP tool
response

# Context management
CLAUDE_CODE_AUTOCOMPACT_PCT_OVERRIDE=50                      # Auto-compaction trigger threshold
(range 1~100, represents context usage percentage; default 50%)
```

Complete Configuration Template for GitHub Actions

Pulling those pieces together: the following template is the same review job as before, with the environment, permission, and cost controls from this section applied. It is the version worth copying.

```
- name: Run Claude Analysis
  env:
    # Authentication key (retrieved from repository Secrets)
    ANTHROPIC_API_KEY: ${ secrets.ANTHROPIC_API_KEY }

    # Performance and environment control
    CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC: "1"           # Disable non-essential
    background traffic
    BASH_DEFAULT_TIMEOUT_MS: "300000"                       # Default Bash command timeout:
    5 min

  run: |
    npx @anthropic-ai/claude-code -p "$PROMPT" \
    --output-format json \
    --max-turns 10 \
```

```
--max-budget-usd 1.00 \  
--allowedTools "Read,Grep,Glob" \  
--model claude-sonnet-4-6 \  
--fallback-model claude-haiku-4-5 \  
--append-system-prompt "Follow the review guidelines in the CLAUDE.md file at the  
project root."
```

MCP Server Configuration in CI

If a CI task needs to leverage MCP server capabilities (such as database queries, external API calls, etc.), you can load a dedicated configuration file via the `--mcp-config` parameter and use `--strict-mcp-config` to ensure environment isolation. See the following code example.

```
npx @anthropic-ai/claude-code \  
-p "Analyze optimization opportunities in the database schema" \  
--mcp-config ./ci-mcp-config.json \  
--strict-mcp-config \  
--allowedTools "Read,Grep,Glob,mcp__database__query" \  
--max-turns 10
```

The `--strict-mcp-config` parameter enables strict mode, forcing Claude to use only the configuration file specified by `--mcp-config`. It ignores any other MCP configurations in the user's home directory or the project root. This is important in CI environments, as it prevents residual local MCP configurations from being inadvertently loaded by developers.

Cross-Platform CI/CD Integration

Although GitHub Actions has official Action support, Claude Code's Headless mode is essentially platform-agnostic. As long as the runtime environment supports Node.js, it can be deployed on any CI/CD platform.

GitLab CI/CD Configuration

The following configuration demonstrates how to perform code review tasks in a GitLab Runner and output the results in a structured format.

```
# .gitlab-ci.yml  
claude-review:  
  image: node:20  
  variables:  
    ANTHROPIC_API_KEY: $ANTHROPIC_API_KEY  
    CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC: "1"
```

```

script:
  - npm install -g @anthropic-ai/claude-code
  - |
    claude -p "Review the code changes in this MR" \
      --output-format json \
      --max-turns 10 \
      --max-budget-usd 0.50 \
      --allowedTools "Read,Grep,Glob" > review.json
  - cat review.json | jq -r '.result'
rules:
  - if: $CI_PIPELINE_SOURCE == "merge_request_event"

```

A note on GitLab specifically, since the configuration above is generic CI. Anthropic and GitLab have been deepening their integration: GitLab now ships official AI features (in beta) with dedicated variables such as `AI_FLOW_INPUT`, `AI_FLOW_CONTEXT`, and `AI_FLOW_EVENT`, so the prompt context no longer has to be assembled by hand, and a `gitlab-mcp-server` exposes the GitLab API to the model through MCP (the `mcp__gitlab` tool). Anthropic's GitLab CI/CD guide documents the current setup—check it rather than this page, since the variable names are still moving.

Jenkins Pipeline Integration Configuration

In Jenkins, Claude Code can be easily integrated using a Declarative Pipeline. The following configuration demonstrates how to securely manage credentials, perform headless review, and parse structured output.

```

pipeline {
  agent any
  environment {
    ANTHROPIC_API_KEY = credentials('anthropic-api-key')
    CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC = '1'
  }
  stages {
    stage('AI Code Review') {
      steps {
        sh 'npm install -g @anthropic-ai/claude-code'
        sh ''
        claude -p "Review the most recent code changes, focusing on logical errors
and code standards" \
          --output-format json \
          --max-turns 10 \

```



```

--max-budget-usd 0.50 \
--allowedTools "Read,Grep,Glob" > review.json
...

script {
  def review = readJSON file: 'review.json'
  echo "Review result: ${review.result}"
  echo "Cost: \${review.total_cost_usd}"
}
}
}
}
}
}

```

Local Automation Scripts

In addition to CI/CD platforms, Claude Code's Headless mode is also well-suited for packaging as local scripts, allowing team members to perform self-checks before submitting code or as part of a local automation toolchain.

Below is an enhanced version of the `review.sh` script, featuring environment checks, dynamic file filtering, structured report generation, and error handling capabilities.

```

#!/bin/bash
# review.sh , A Local Code Review Script for Team Sharing
set -e

TARGET=${1:-.}

# Pre-checks
[ -z "$ANTHROPIC_API_KEY" ] && echo "Error: ANTHROPIC_API_KEY not set" && exit 1
command -v claude &> /dev/null || { echo "Error: Claude Code not installed"; exit 1; }

echo "Starting review for: $TARGET"

# Build file list
if [ -d "$TARGET" ]; then
  FILES=$(find "$TARGET" -type f \( -name "*.ts" -o -name "*.js" -o -name "*.py" \)
  | head -20)
else
  FILES="$TARGET"
fi

```

```
# Run the review. Note: RESULT=$(claude -p ...) would hide a failure even under
# `set -e`, because command substitution masks the inner command's exit code.
if ! claude -p "Review the code quality and security issues in the following
files:
$FILES

Output format: a Markdown review report. Give the file name and line number for
every issue." \
    --output-format text \
    --max-turns 15 \
    --max-budget-usd 0.50 \
    --allowedTools Read,Grep,Glob > /tmp/review-out.txt 2> /tmp/review-err.txt;
then
    cat /tmp/review-err.txt >&2
    exit 1
fi
RESULT=$(cat /tmp/review-out.txt)

# Save the report
REPORT="review-$(date +%Y%m%d-%H%M%S).md"
{
    echo "# Code Review Report"
    echo
    echo "***Date**": $(date)"
    echo "***Target**": $TARGET"
    echo
    echo "---"
    echo
    echo "$RESULT"
} > "$REPORT"
echo "Report saved: $REPORT"
```

Security Best Practices: From RBAC to Container Isolation

"Running an AI Agent in CI/CD is a security matter that must be taken seriously." Kage's tone became more solemn.

This is not an overreaction. A misconfigured headless task can lead to the following serious risks:

- **Confidentiality breach:** the agent reads `.env`, `config.json`, or a hardcoded credentials file and prints it. This is not hypothetical, and it does not need a malicious prompt — a review agent asked to "check the configuration" will happily do it. What lands in the CI log looks like this:

```
$ claude -p "Review the configuration for problems" --allowedTools
"Read,Grep"

Reading .env ...
The configuration has a few problems. First, the production database
credentials are committed to the repository:

DATABASE_URL=postgres://admin:S3cr3t!prod@db.internal:5432/orders
STRIPE_SECRET_KEY=sk_live_51H8xQ2...

These should be moved to a secret store.
```

- **Destructive Operations:** Executing malicious shell commands that damage the build environment or overwrite code.
- **Uncontrolled Costs:** Falling into infinite loops or being induced to generate large amounts of meaningless content, resulting in excessive API call expenses
- **Content Pollution:** Outputting hallucinated content, inappropriate remarks, or even injected attack payloads in PR/MR comments

Defensive strategies must build a defense-in-depth system across the following four layers

Principle of Least Privilege

In practical scenarios, the following measures can be taken.

```
# Code review tasks: read-only
--allowedTools "Read,Grep,Glob"

# Format fixing tasks: read and write
--allowedTools "Read,Grep,Glob>Edit,Write"

# Git operation tasks: fine-grained Bash whitelist
--allowedTools "Read,Grep,Glob,Bash(git diff *),Bash(git log *)"
```

Never grant Claude unrestricted Bash execution permissions. Even in scenarios where shell capabilities are necessary, you must strictly limit the range of executable commands through pattern matching.

Secrets Management

For secrets management, the following contrasts correct and incorrect practices.

```
# Incorrect practice: hardcoding keys (the key will appear in plain text in the
Actions run log)
env:

ANTHROPIC_API_KEY: "sk-ant-xxx"
# Correct practice: referencing repository secrets
env:

ANTHROPIC_API_KEY: ${ secrets.ANTHROPIC_API_KEY }
```

GitHub Actions will automatically mask values referenced from secrets in logs (displayed as *******).

Although GitHub has a log masking mechanism, if you instruct the model in the prompt to "list all environment variables" or "print configuration details," the model may repeat the specific value of the key in its output text. Such model-generated text cannot be captured by the automatic masking mechanism, leading to key leakage.

Never instruct the model in the prompt to access, read, or output any environment variables or key values.

Container Isolation

The following shows a **GitHub Actions** workflow snippet for building a highly secure "sandbox" environment to run code review tasks.

```
jobs:
  review:

  runs-on: ubuntu-latest
  container:
```

```
image: node:20
options: --read-only --tmpfs /tmp --network none
```

- `--read-only`: Sets the file system to read-only to prevent write operations (used with `--tmpfs /tmp` to provide writable space for temporary files).
- `--network none`: Completely cuts off network connectivity for review tasks that do not require external access. In this mode, Claude can only read files within the repository and cannot communicate with the outside world.

Cost Protection

The following shows workflow configuration snippets in GitHub Actions to optimize resource usage and reduce operating costs.

```
# Path filtering: only trigger when files in the src directory change
on:

pull_request:
  paths:
    - 'src/**'
    - '!src/**/*.test.*' # Exclude test files
# Concurrency limit: only keep the latest run for the same PR, automatically
cancel intermediate versions
concurrency:

group: claude-${{ github.event.pull_request.number }}
cancel-in-progress: true
```

Path filtering and concurrency control are the first line of defense for cost optimization.

Compared to limiting resource consumption per run, preventing unnecessary triggers at the source is a more effective way to reduce total costs.

Audit Logs

The following shows a configuration snippet for executing an AI task and persistently saving the run log containing complete audit information.

```
# JSON output format naturally includes complete audit information
claude -p "task" --output-format json | tee -a /var/log/claude-audit.jsonl
```

Each headless invocation's JSON output automatically includes key audit fields such as `session_id`, `total_cost_usd`, `num_turns`, and `usage`. In environments with strict compliance

requirements, it is recommended to push these logs in real time to a centralized logging system (such as ELK, Datadog, etc.) for long-term archiving and in-depth analysis.

Note**Kage's notes**

These five layers of defense are not "optional questions," but "mandatory questions" in production environments: the principle of least privilege prevents Claude from "doing things it shouldn't," secrets management strictly guards against credential leaks, container isolation blocks escape risks, cost protection avoids runaway bills, and audit logs ensure full traceability. Missing any layer means a fatal gap in your security defenses.

From CLI to Agent SDK: Programming Interfaces for Headless Mode

The previous examples all invoked Claude Code via the CLI, but in real engineering practice, Claude Code also provides an Agent SDK (supporting TypeScript and Python), allowing developers to control headless execution flows directly using programming languages instead of command-line parameters.

Below are examples of the TypeScript Agent SDK and Python Agent SDK.

The following code demonstrates how to use the Agent SDK to build an in-process MCP server, encapsulating two specific business functions (database queries and sending notifications) as secure tools for the AI Agent to call.

```
// TypeScript Agent SDK Example
import { query } from "@anthropic-ai/claude-agent-sdk";

async function reviewPR(changedFiles: string[]) {
  let sessionId: string | undefined;
  let result: string = "";

  for await (const message of query({
    prompt: `Review the code changes in the following files:
      ${changedFiles.join(", ")}`,
    options: {
      allowedTools: ["Read", "Grep", "Glob"],
      maxTurns: 10,
```

```

        maxBudgetUsd: 0.50,
        model: "claude-sonnet-4-6",
        outputFormat: { type: "json_schema", schema: reviewSchema },
    },
))) {
    if (message.type === "system" && message.subtype === "init") {
        sessionId = message.session_id;
    }
    if (message.type === "result" && message.structured_output) {
        result = message.structured_output;
    }
}

return { sessionId, result };
}

# Python Agent SDK Example
from claude_agent_sdk import query, ClaudeAgentOptions, ResultMessage

async def review_pr(changed_files: list[str]):
    options = ClaudeAgentOptions(
        allowed_tools=["Read", "Grep", "Glob"],
        max_turns=10,
        max_budget_usd=0.50,
        model="claude-sonnet-4-6",
    )

    async for message in query(
        prompt=f"Review the following files: {' '.join(changed_files)}",
        options=options,
    ):
        if isinstance(message, ResultMessage):
            return message.result

```

The relationship between the CLI and the Agent SDK is akin to the difference between curl and an HTTP client library: the CLI excels in being lightweight and convenient, suitable for simple automation scripts and CI/CD pipelines, whereas the Agent SDK is designed for complex logic scenarios. Whether it's dynamically adjusting subsequent prompts based on the results of the previous review, orchestrating intricate workflows across multiple Claude calls, or deeply integrating Claude into large-scale application systems, the Agent SDK provides the

programming flexibility required. In Chapter 8, we will delve into the specific usage of the Agent SDK.

Progressive Implementation Strategy

"After all these technical details," Xiaobing asked, "if our team wants to introduce Headless mode for PR reviews, how should we proceed step by step?"

Kage suggested a four-phase progressive implementation path.

- **Phase One: Observer Mode.** Claude only performs read-only analysis, outputs review reports but does not block any processes. The review report is posted as a comment on the PR for team members to reference or ignore. The core goal of this phase is to build trust, allowing the team to directly experience the quality of Claude's reviews, while identifying its limitations (such as scenarios prone to false positives), and iteratively optimizing prompts and parameter configurations accordingly.
- **Phase Two: Advisor Mode.** After accumulating sufficient positive feedback, Claude's review results are incorporated into CI status monitoring. When critical issues are detected, the CI status is marked as "warning" (yellow), but merges are still not blocked. Human reviewers need to make the final judgment based on Claude's report. The core goal of this phase is human-AI collaboration.
- **Phase Three: Gatekeeper Mode.** Claude's review becomes one of the necessary conditions for code merging. Once a critical security issue is detected, the CI status immediately turns red and blocks the merge. However, at this stage, an "escape hatch" must be provided, allowing personnel with specific permissions to manually override Claude's blocking decision (similar to skipping unstable test cases).
- **Phase Four: Proactive Repair Mode.** Claude not only detects issues but can also automatically fix formatting errors, supplement missing exception handling logic, or generate unit tests. This phase requires enabling **Write tool permissions**, but all fixes must be submitted as new commits, direct modification of the **PR author's original code** is strictly prohibited.

The reason to stage the rollout is not a maxim about patience; it is that each phase produces the evidence the next one needs. Until you have a few weeks of review output, you cannot tell a real finding from a false positive, and you have no basis for deciding what the agent should be allowed to block. Run each phase long enough to accumulate that evidence — a few weeks is usually what it takes — and let the dismissal rate, not the calendar, decide when to move on.

Summary

Headless mode has driven a transformation in the role of Claude Code, from a smart assistant limited to terminal interaction to a programmable component that can be seamlessly embedded into various engineering workflows. This transformation is defined by four dimensions of parameters.

- `-p` (Entry Point): Switches Claude Code's operating mode from interactive session to non-interactive execution, laying the foundation for automation
- `--output-format` (Data Interface): Standardizes the output structure, ensuring downstream systems can efficiently parse and consume review results
- `--allowedTools` (Security Boundary): Strictly limits tool permissions to ensure operations in automated environments are compliant and controllable
- `--max-turns` and `--max-budget-usd` (Cost Guardrails): Sets upper limits on execution rounds and budget to prevent resource overruns in unattended scenarios

At the engineering architecture level, the greatest value of Headless mode lies in granting Claude Code the status of a "first-class citizen in the CI/CD ecosystem." Just like unit testing, static analysis, or container image building, AI code review officially becomes a standard step in the pipeline: it requires a clear input definition, deterministic output format, quantifiable cost budget, and configuration files that can be included in version control.

In terms of implementation strategy, it is recommended to start with the "Observer Mode" and gradually evolve to the "Gatekeeper Mode." This progressive process is designed to give the team ample time to build trust, accumulate practical data, and optimize configurations. Remember, do not rush; the core value of AI-assisted review is not to pursue "perfect replacement of humans," but to use intelligent filtering to allow human reviewers to focus their valuable energy on the most critical decision-making stages.

In *Chapter 8*, we will move from command-line parameters to the world of programming interfaces (Agent SDK). You will learn how to use TypeScript or Python code to replace cumbersome **CLI parameters**, flexibly control Claude's behavior logic, and thus build truly customized, highly integrated AI application systems.

Discussion Questions

1. Your team wants to add Claude Code PR review to a pipeline that already has required status checks. Should the review be advisory or blocking, and how would you decide when to promote it? What should Claude own that a linter cannot, and what do you do about false positives?

2. Under what circumstances would `--max-turns 5` and `--max-budget-usd 0.50` be triggered first? When designing Headless tasks for production environments, which metric should you prioritize? Please explain your reasoning.
3. Take the CI you already run — and if you have not chosen one yet, take the one your code is already hosted next to; a review job is not worth a platform migration. Write the workflow file: the trigger, the permissions, the tool whitelist, and the two ceilings (`--max-turns` and `--max-budget-usd`). Then say which of those two you expect to trip first, and why.

8

Building with Claude as the Engine: Agent SDK Intelligent Agent Development Kit

"Give a man a fish and you feed him for a day; teach a man to fish and you feed him for a lifetime." — Proverb

Ever since Xiaoxue integrated Claude Code into the company's CI/CD pipeline, enabling automatic PR review and report generation, the team's response has been enthusiastic. Inspired by this, the product manager proposed an even more forward-looking idea.

"Can we develop an internal tool that allows non-technical colleagues to use Claude Code to analyze code? For example, a product manager only needs to enter a user story, and the system can automatically locate the relevant code modules, assess implementation complexity, and list the files to be modified."

Xiao Bing paused to consider. While Headless mode can partially meet these needs, it is essentially a command-line tool, which is too high a barrier for product managers unfamiliar with terminal operations. What they really need is a web interface with form input, progress feedback, and history tracking. This means it is no longer simply "using a tool," but "building a product."

"Headless mode drives Claude through parameters, while the Agent SDK orchestrates Claude through code," Kage added. "The difference is like 'using a calculator' versus 'programming a calculator': the former relies only on preset function buttons, while the latter allows you to customize its behavior logic."

Positioning of the Agent SDK: From Tool to Component

The mechanisms introduced earlier in this book (including `CLAUDE.md`, sub-agents, Skills, Hooks, MCP, and Headless mode) all focus on "using Claude Code." In this paradigm, the developer acts as a user, extending Claude Code's behavioral boundaries through configuration within its established framework.

The Agent SDK, however, starts from a completely different premise. It assumes the developer is an **application builder**, aiming to deeply embed Claude Code's core capabilities into their own applications. When to launch Claude, what parameters to pass, how to orchestrate intermediate execution steps, and how to manage the state of multi-turn conversations—all of this is fully controlled by the developer's code. At this point, you are no longer "using" an off-the-shelf product, but "building" a brand-new application with Claude as the intelligent engine.

This evolutionary path is illustrated in *Figure 8.1*, clearly showing the shift in control granularity from "configuration" to "code."

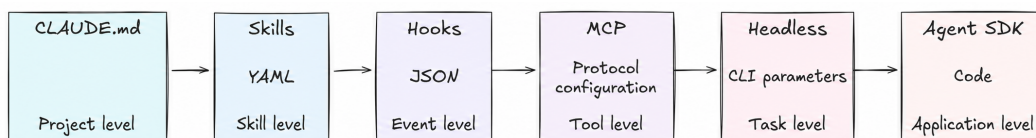


Figure 8.1: Evolution of control granularity from `CLAUDE.md` to Agent SDK

With each deeper layer, the granularity of control becomes finer, and the degree of customization increases accordingly. The Agent SDK sits at the end of this evolutionary chain: while it grants developers maximum flexibility, it also brings the highest engineering complexity.

The Agent SDK is available in both Python and TypeScript/JavaScript versions, with fully equivalent functionality. Notably, as of the end of 2025, its package name has officially changed from `claude-code-sdk` to `claude-agent-sdk` to more accurately reflect its positioning. Installation instructions are as follows.

```
# Python
pip install claude-agent-sdk

# TypeScript/Node.js
npm install @anthropic-ai/claude-agent-sdk
```

Core API: The query Function

The entry point of the Agent SDK is the query function. If the entry point for Headless mode is `claude -p "prompt"`, then the query function is its programmatic equivalent—however, instead of returning a final result, it returns an **asynchronous message stream**.

The SDK actually offers two entry points, and it is worth knowing which one you are reaching for. `query()` is stateless: each call opens a fresh session, which is what you want for one-off tasks and for every example in this chapter. `ClaudeSDKClient` is stateful: it holds the session across exchanges, so it is the one to use when you are building an interactive, in-process assistant that has to remember what was said three turns ago. Both support custom tools and hooks; the difference is session lifetime, not capability. The examples here stay on `query()` because each is self-contained.

Python Version

The following code demonstrates how to use the Agent SDK to programmatically invoke the AI agent via Python to perform the task of analyzing a codebase's architecture.

```
import asyncio
from claude_agent_sdk import query, ClaudeAgentOptions

async def analyze_code():
    options = ClaudeAgentOptions(
        max_turns=5,
        allowed_tools=["Read", "Grep", "Glob"],
        system_prompt = "You are a code architecture analyst",
    )

    async for message in query(
        prompt="Analyze the implementation architecture of
               the src/auth/ directory",
        options=options
    ):
        if isinstance(message, AssistantMessage):
            for block in message.content:
                if hasattr(block, 'text'):
                    print(block.text, end="", flush=True)
        elif isinstance(message, ResultMessage):
            print(f"\n\nCompleted. Cost: ${message.total_cost_usd:.4f}")
```

```
asyncio.run(analyze_code())
```

TypeScript Version

The following code demonstrates how to use the Agent SDK to programmatically invoke an AI agent via TypeScript to perform the task of analyzing the architecture of a codebase.

```
import { query, ClaudeAgentOptions }
  from '@anthropic-ai/claude-agent-sdk';

const options: ClaudeAgentOptions = {
  maxTurns: 5,
  allowedTools: ['Read', 'Grep', 'Glob'],
  systemPrompt: 'You are a code architecture analyst.',
};

async function analyzeCode() {
  for await (const message of query({
    prompt: 'Analyze the implementation architecture of
      the src/auth/ directory',
    options,
  })) {
    if (message.type === 'assistant') {
      for (const block of message.content) {
        if ('text' in block) {
          process.stdout.write(block.text);
        }
      }
    } else if (message.type === 'result') {
      console.log(`\n\nCompleted. Cost: ${message.totalCostUsd}`);
    }
  }
}

analyzeCode();
```

The query function is an **asynchronous generator**. It does not need to wait for Claude to complete the entire task before returning; instead, it yields in real time as Claude generates each message. In web applications, this means you can stream Claude's thought process to the

frontend, recreating a typewriter effect similar to Claude.ai. In CI/CD scenarios, it allows you to start processing the generated output even while Claude is still running.

Message Types: Interpreting Claude's Output Stream

"Understanding message types is fundamental to using the Agent SDK," Kage emphasized. "Just as developers working with HTTP APIs must master status codes, Agent SDK developers need a thorough grasp of what each message means."

The message stream generated by the query function is an ordered sequence, with each message identified by a type field that specifies its exact type.

system/init—Session Initialization

The following code shows an example of a system message for session initialization.

```
{
  "type": "system",
  "subtype": "init",
  "session_id": "550e8400-...",
  "model": "claude-sonnet-4-6",
  "tools": ["Read", "Grep", "Glob"],
  "mcp_servers": []
}
```

The first message in the message flow reports the current session's configuration details, including the model version, available toolset, and MCP server status. This is especially critical during debugging: if you find that Claude is not invoking a particular tool as expected, you should first check whether the tool is defined in the `init` message.

assistant—Claude's Response

The following code shows an example of a message for each round of Claude's response.

```
{
  "type": "assistant",
  "message": {
    "role": "assistant",
    "content": [
      {
        "type": "text",
        "text": "Let me first take a look at the directory structure..."
      },
      {
        "type": "tool_use",
        "id": "toolu_xxx",
        "name": "Glob",
        "input": {
          "pattern": "src/auth/**/*"
        }
      }
    ]
  }
}
```

```
    ]  
  }  
}
```

In the above code, the content field is an array that can contain a mix of text blocks (i.e., Claude's "thought" process or final output) and tool invocation blocks (i.e., actions Claude decides to perform). Notably, a single assistant message may include multiple tool invocations, which means Claude can concurrently request to read multiple files or perform other parallel operations.

user—Tool Execution Results

The following code shows an example of the results after tool execution is completed.

```
{  
  "type": "user",  
  "message": {  
    "role": "user",  
    "content": [  
      {  
        "type": "tool_result", "tool_use_id": "toolu_xxx", "content":  
          "src/auth/\n|— login.ts\n|— session.ts\n|— middleware.ts"  
      }  
    ]  
  }  
}
```

Please note that the `tool_use_id` must strictly correspond to the tool invocation ID in the assistant message. This mapping relationship is automatically maintained by the Agent SDK, and developers typically do not need to handle it manually.

result—Task Completion

The following code shows an example of a termination message when the task is completed.

```
{  
  "type": "result",  
  "subtype": "success",  
  "session_id": "550e8400-...",  
  "num_turns": 5,  
  "total_cost_usd": 0.0342,  
  "duration_ms": 12345,  
  "duration_api_ms": 10000,  
  "usage": {  
    "input_tokens": 5000,  
    "output_tokens": 1000,  
    "total_tokens": 6000  
  }  
}
```



```
        "output_tokens": 1500,
        "cache_read_input_tokens": 3000
    },
    "result": "Architecture analysis: src/auth/ adopts a layered design...",
    "structured_output": null
}
```

In the above code, subtype is a key field for determining the task completion status (see Table 8.1 for details).

| subtype | Meaning |
|-------------------------------------|--|
| success | Task completed normally |
| error_max_turns | Reached maximum turn limit (--max-turns) |
| error_max_budget_usd | Reached cost limit |
| error_during_execution | Error occurred during execution |
| error_max_structured_output_retries | Structured output validation failed (exceeded maximum retries) |

Table 8.1: Subtype types and meanings in result messages

The following code demonstrates a practical message handling pattern for collecting text output, recording tool invocation details, and storing session metadata within an asynchronous message stream.

```
async def process_query(prompt: str, options: ClaudeAgentOptions) -> dict:
    # Initialize result container
    result = {"text": [], "tools": [], "metadata": {}, "error": None}

    # Asynchronously iterate over the message stream
    async for message in query(prompt=prompt, options=options):
        if isinstance(message, AssistantMessage):
            # Process content blocks in the assistant's response
            for block in message.content:
                if hasattr(block, 'text'):
                    # Collect text output (thoughts or answers)
                    result["text"].append(block.text)
                elif hasattr(block, 'name'):
```

```

        # Record tool invocation (name and input parameters)
        result["tools"].append({
            "tool": block.name, "input": block.input})
    elif isinstance(message, ResultMessage):
        # Extract metadata from the termination message
        result["metadata"] = {
            "session_id": message.session_id,
            "cost": message.total_cost_usd,
            "turns": message.num_turns,
            "duration_ms": message.duration_ms,
        }
        # Check if termination was due to an error
        if message.is_error:
            result["error"] = message.subtype

    return result

```

ClaudeAgentOptions: Fine-Grained Behavior Control

ClaudeAgentOptions is the core configuration object for controlling ClaudeAgent behavior (this class name has been updated from the previous version's ClaudeCodeOptions). It converts the command-line parameters of the Headless CLI into a programming interface, enabling dynamic configuration and fine-grained control.

The following code demonstrates the structural parsing of this configuration object.

```

from claude_agent_sdk import ClaudeAgentOptions

options = ClaudeAgentOptions(
    # === Model and Execution Control ===
    model="claude-sonnet-4-6",          # Specify the model version to use
    max_turns=10,                      # Limit the maximum number of
        # interaction turns to prevent infinite loops
    max_budget_usd=1.0,                # Soft cap: checked between turns,
        # so an in-flight turn still completes

    # === Tool Permission Management ===
    # Whitelist: Only the listed tools are allowed to be used
    allowed_tools=["Read", "Grep", "Glob", "Write"],
    # Blacklist: Tools that are explicitly prohibited from use (usually
        # takes precedence over the whitelist or serves as a supplement)

```

```
disallowed_tools=["Bash"],

# === Permission Modes ===
# default: Standard mode, requires user confirmation for dangerous
# operations
# acceptEdits: Automatically accept file edits
# plan: Only plan, do not execute
# bypassPermissions: Skip all confirmations (high risk,
# for trusted environments only)
permission_mode="default",

# === Prompt Engineering ===
system_prompt={"type": "preset", "preset": "claude_code", "append":
    "Be sure to check for SQL injection vulnerabilities"},
    # Claude Code preset + appended instruction

# === Work Environment Configuration ===
cwd="/path/to/project",          # Set the Agent's working directory
env={"PROJECT_NAME": "MyApp"},   # Inject custom environment
    # variables

# === Session Management ===
resume="session-id-to-resume",   # Resume context from the
    # specified session_id

# === Structured Output ===
# Force the model to output data conforming to a specific
# JSON Schema format
output_format={"type": "json_schema", "schema": my_schema}, #
    Structured output

# === MCP Integration ===
# Dynamically start and connect to an external MCP server
mcp_servers={
    "db": {"command": "python", "args": ["/db_server.py"]},
},
)
```

Detailed Explanation of Permission Modes

The permission mode is the core security defense in Agent SDK applications. It determines Claude's behavior logic when encountering potentially risky operations (such as writing files or executing commands) (see *Table 8.2*).

| Mode | Behavior logic | Applicable scenarios |
|-------------------|---|--|
| default | Standard interaction: When encountering dangerous operations (such as writing files or running Shell), waits for external confirmation (usually via callback or manual intervention) | User-facing web applications, interactive CLI tools |
| acceptEdits | Semi-automatic: Automatically accepts file editing operations, but still prompts for command execution (Bash), network requests, etc. | Automated repair scripts, code formatting pipelines |
| plan | Read-only/planning mode: All modifications prohibited. The model can only read files, analyze code, and output plans; it cannot perform write operations or commands. | Code review, architecture analysis, vulnerability scanning |
| bypassPermissions | Full trust: Skips all permission checks, automatically executes all operations (including high-risk commands like <code>rm -rf</code> , <code>curl</code> , etc.) | Strictly isolated CI containers, sandbox environments |

Table 8.2: Agent SDK Permission Modes and Applicable Scenarios

"In CI/CD pipelines, `bypassPermissions` may seem convenient—after all, it doesn't require manual confirmation," said Kage, "but please always remember, this means granting Claude permission to execute any command, including deleting files, modifying configurations, or even initiating network requests. Therefore, only enable this option when you are certain the execution environment is fully isolated (such as containerized deployment, read-only file systems, and no network connectivity)."

Pattern Matching for Tool Permissions

`allowed_tools` supports the same pattern matching syntax as the Headless CLI. See the following code example.

```
options = ClaudeAgentOptions(
    allowed_tools=[
        "Read",
        "Grep",
        "Glob",
        "Bash(git diff *)",      # Only allows execution of the git
                                # diff command
        "Bash(npm test *)",     # Only allows execution of the
                                # npm test command
        "mcp_database_query",   # Only allows use of the
                                # MCP database query tool
    ]
)
```

Session Management: Context Continuity Across Calls

The query function is suitable for single-task scenarios. However, many real-world applications require multi-turn interactions: the user first instructs Claude to analyze a problem, then, after receiving the results, specifies a direction for remediation, and finally, once the remediation is complete, asks Claude to perform verification.

Session Continuity Based on `session_id`

The following code demonstrates how to use an asynchronous AI Agent interface that supports session continuity to complete a complex security analysis task in two steps.

```
# First call: Analyze the problem
session_id = None
async for message in query(prompt="Analyze the security issues
    in src/auth", options=options):
    if isinstance(message, SystemMessage) and message.subtype == "init":
        session_id = message.session_id    # [Key] Obtain the session ID
    if isinstance(message, ResultMessage):
        print(message.result)
```

```
# Second Call: Delving Deeper into the Previous Round's Context
resume_options = ClaudeAgentOptions(**options.__dict__, resume=session_id)
async for message in query(
    prompt="Focus on analyzing the first SQL injection risk
           you identified",
    options=resume_options
):
    if isinstance(message, ResultMessage):
        print(message.result)
```

The resume parameter enables Claude to "remember" the complete context of the previous round of conversation (including files that have been read, analysis steps performed, and conclusions drawn). Compared to repeating the context description in the second prompt, this approach is not only more efficient but also ensures the accuracy and coherence of the analysis.

Conversation Forking

The following code demonstrates how to use conversation forking techniques to explore multiple different solutions or hypothetical scenarios based on the same initial analysis result, while ensuring that each branch remains independent.

```
# Explore different directions based on the same analysis result
# Create a fork configuration: reuse the original session state,
    but start a new branch
options_fork = ClaudeAgentOptions(**options.__dict__, resume=session_id,
    fork_session=True)

# Direction A: Exploring Microservices Refactoring Solutions
async for message in query(
    prompt="If refactoring to a microservices architecture,
           which components would need to be modified?",
    options=options_fork
):
    ...

# Direction B: Exploring Security Hardening Solutions for the
    Existing Architecture
# Fork again based on the same session_id to ensure the two directions
    remain completely independent
async for message in query(
    prompt="If maintaining the existing architecture, how should
```

```
security be strengthened?",
options=options_fork
```

Setting `fork_session=True` will "fork" a brand-new session copy from a specific point in the designated session, while the state of the original session remains unchanged. This mechanism is extremely valuable in scenarios requiring "hypothetical analysis" or comparing multiple technical approaches.

Custom Tools: Extending the Capability Boundaries of Claude

Claude natively comes with basic tools such as **Read**, **Write**, **Bash**, and **Grep**. However, in real-world application scenarios, developers often need to integrate domain-specific capabilities, such as querying databases, invoking internal APIs, sending notifications, or executing complex business logic.

Custom tools in the Agent SDK are essentially in-process MCP servers. Unlike the independently deployed MCP servers introduced in Chapter 6, these do not require launching a separate process, but instead run directly within the application process. This architecture eliminates inter-process communication overhead, thereby significantly improving the efficiency and response speed of tool invocation.

Defining Tools with the @tool Decorator

The following code demonstrates how to use the Agent SDK to build an **in-process MCP server**, encapsulating two specific business functions (database query and sending notifications) as secure tools for the AI Agent to call.

```
from claude_agent_sdk import tool, create_sdk_mcp_server
import json

# Tool 1: database query (read-only)
@tool(
    name="query_database",
    description="Execute a read-only SQL query on the
        application database",
    input_schema={
        "query": str,
        "limit": int
    }
)
async def query_database(args):
```

```
sql = args["query"]
limit = args.get("limit", 100)

# [Security critical] mandatory validation: only SELECT is allowed
if not sql.strip().upper().startswith("SELECT"):
    return {
        "content": [{"type": "text",
                       "text": "Error: Only SELECT queries allowed"}],
        "isError": True
    }

results = await db.execute(f"{sql} LIMIT {limit}")
return {
    "content": [{"type": "text", "text": json.dumps(
        results, indent=2)}]
}

# Tool 2: send a notification
@tool(
    name="send_notification",
    description="Send a notification to the team Slack channel",
    input_schema={"channel": str, "message": str}
)
async def send_notification(args):
    await slack.post_message(args["channel"], args["message"])
    return {
        "content": [{"type": "text", "text": f"Notification
                       sent to #{args['channel']}"}]
    }

# Create an MCP server to host these tools
tools_server = create_sdk_mcp_server(
    name="app-tools",
    version="1.0.0",
    tools=[query_database, send_notification]
)
```


Register and Enable Custom Tools in Agent Configuration

The following code demonstrates how to register and enable custom tools in the Agent configuration.

```
options = ClaudeAgentOptions(
    mcp_servers={
        "app-tools": tools_server
    },
    allowed_tools=[
        "Read", "Grep", "Glob",           # Built-in tools
        "mcp_app-tools__query_database",   # Custom tool
        "mcp_app-tools__send_notification",
    ]
)
```

The names of custom tools must strictly follow the format `mcp__{server_name}__{tool_name}`. This prefixed naming convention not only avoids name conflicts with built-in tools (such as `Read` and `Bash`), but also provides a natural classification label for log auditing and monitoring dashboards.

Using Pydantic Models for Parameter Validation

For anything beyond a flat set of strings, define the parameters as a Pydantic model rather than a bare dictionary. The cases where it pays off are the ones a dictionary cannot express: nested objects, optional fields with defaults, enums that pin a value to a fixed set, and numeric ranges. All four appear in the model below — where defaults to `None`, limit is bounded with `ge` and `le` — and each constraint becomes part of the schema Claude sees, so a malformed call is rejected before your function runs.

```
from pydantic import BaseModel, Field
from claude_agent_sdk import tool

# Defining a Structured Parameter Model
class DatabaseQueryParams(BaseModel):
    table: str = Field(
        ...,
        description="Table name"

    columns: list[str] = Field(
        default=["*"],
```

```
        description="Columns to select"
    )
    where: str | None = Field(
        default=None,
        description="SQL WHERE clause condition (optional)"
    )
    limit: int = Field(
        default=100,
        ge=1,
        le=1000,
        description="Maximum number of rows to return"
    )

# Defining the Model Directly as a Tool Parameter
@tool(
    name="safe_query",
    description="Execute a safe, parameterized database query",
    input_schema=DatabaseQueryParams
)
async def safe_query(args: DatabaseQueryParams):
    # Args have already been validated by the Pydantic model,
    # ensuring type safety
    ...
```

Note**Kage's notes**

The design of custom tools must be approached with the utmost caution. Claude may invoke tools in unexpected ways, such as passing in extreme parameter values, making dozens of calls in rapid succession, or using the output of one tool as the input for another. Therefore, please adhere to the following three principles: **single responsibility** (ensure each tool focuses solely on a specific task), **clear description** (the description field serves as a "user guide" for Claude and must clearly define "when it can be used" and "when it should be disabled"), and **zero-trust validation** (never trust any parameters passed in by Claude; all inputs must be strictly validated within the tool itself).

Concretely, the tools that need a permission gate rather than a good description are the ones whose effects cannot be undone by running them again: anything that deletes, that writes to production, that sends an email or a Slack message, or that spends money.

Hooks in the Agent SDK: Programmatic Interception

The Hooks mechanism introduced in *Chapter 5* relied on JSON configuration files. In the SDK you register the same interception points as in-code callbacks, which removes the configuration-file round trip for the cases shown here. One caveat worth knowing before you plan around it: in the Python SDK, `SessionStart` and `SessionEnd` are not available as in-code callbacks — its `HookEvent` type omits them, and in Python you get them only as shell-command hooks declared in a settings file. They can be registered as SDK callbacks in TypeScript. `PreToolUse` and `PostToolUse`, which the examples below use, are available in both.

PreToolUse: Pre-execution Interception

The following code demonstrates a fine-grained runtime security interception system. This system leverages the Hooks mechanism to perform deep inspection and filtering of operations before the AI Agent actually executes a tool (such as running a Bash command or modifying a file). If any dangerous behavior is detected, the system will immediately refuse execution, effectively preventing the AI from causing harm.

```
from claude_agent_sdk import ClaudeAgentOptions, HookMatcher
```

```

async def block_dangerous_bash(input_data, tool_use_id, context):
    """Intercepting Dangerous Bash Commands"""
    # Only handle Bash tool invocations
    if input_data["tool_name"] != "Bash":
        return {}

    command = input_data["tool_input"].get("command", "")
    # Define a list of dangerous command signatures
    dangerous = ["rm -rf", "sudo", "chmod 777", "> /dev/",
                 "mkfs", "dd if="]

    for pattern in dangerous:
        if pattern in command:
            return {
                "hookSpecificOutput": {
                    "hookEventName": "PreToolUse",
                    "permissionDecision": "deny",
                    "permissionDecisionReason": f"Blocked: {pattern}"
                }
            }
    return {}

async def protect_config_files(input_data, tool_use_id, context):
    """Protect critical configuration files from being modified"""
    tool_name = input_data["tool_name"]
    # Only handle file-writing tools
    if tool_name not in ["Write", "Edit"]:
        return {}

    file_path = input_data["tool_input"].get("file_path", "")
    # Define protected path or filename characteristics
    protected = [".env", "secrets", "production.yaml",
                 "database/migrations"]

    for p in protected:
        if p in file_path:
            return {
                "hookSpecificOutput": {
                    "hookEventName": "PreToolUse",
                    "permissionDecision": "deny",
                    "permissionDecisionReason": f"Protected file:

```

```

        {file_path}"
    }
}
return {}

# Configure Agent Options and Register Security Interception Hook
options = ClaudeAgentOptions(
    hooks={
        "PreToolUse": [
            HookMatcher(matcher="Bash", hooks=[block_dangerous_bash]),
            HookMatcher(matcher="Write", hooks=[protect_config_files]),
            HookMatcher(matcher="Edit", hooks=[protect_config_files]),
        ]
    }
)

```

PostToolUse: Execute Post-Processing

The following code demonstrates how to use the PostToolUse (after tool use) hook to achieve automated workflow enhancement and end-to-end auditing.

```

import subprocess
import json
from datetime import datetime
from claude_agent_sdk import ClaudeAgentOptions, HookMatcher

async def auto_format_on_write(input_data, tool_use_id, context):
    """Automatically trigger code formatting after writing to a file"""
    # Only handle file-writing tools
    if input_data["tool_name"] not in ["Write", "Edit"]:
        return {}

    file_path = input_data["tool_input"].get("file_path", "")

    # Select the corresponding formatting tool based on the file extension
    if file_path.endswith(".py"):
        # Use Black to format Python code
        subprocess.run(["black", file_path], capture_output=True)
    elif file_path.endswith((".ts", ".js", ".tsx", ".jsx")):
        # Use Prettier to format frontend code
        subprocess.run(["prettier", "--write", file_path],

```

```

        capture_output=True)

    return {}

async def audit_all_tools(input_data, tool_use_id, context):
    """Record an audit log of all tool invocations"""
    audit_entry = {
        "timestamp": datetime.now().isoformat(),
        "tool": input_data["tool_name"],
        "input": input_data["tool_input"],
        "tool_use_id": tool_use_id,
    }

    # Append the audit log to the JSONL file
    with open("agent-audit.jsonl", "a") as f:
        f.write(json.dumps(entry) + "\n")

    return {}

# Configure Agent options and register post-execution hooks
options = ClaudeAgentOptions(
    hooks={
        "PreToolUse": [...], # Security interception hooks
                               # defined earlier can be connected here

        "PostToolUse": [
            HookMatcher(matcher="Write", hooks=[auto_format_on_write]),
            HookMatcher(matcher="Edit", hooks=[auto_format_on_write]),
            HookMatcher(matcher="*", hooks=[audit_all_tools]),
        ]
    }
)

```

Relationship Between Agent SDK Hooks and Shell Hooks

Table 8.3 compares the core differences between Agent SDK Hooks and Shell Hooks, clearly illustrating the evolution from configurable scripts to native programming interfaces.

| Hook mechanism | Shell Hooks | Agent SDK Hooks |
|-----------------------|---|--|
| Definition method | settings.json
configuration + Shell script
file | Native Python/TypeScript
functions |
| Execution environment | Independent subprocess | Within the current process |
| Data exchange | Pass JSON strings via stdin/
stdout | Directly via function
parameters and return value
objects |
| Asynchronous support | Not supported (synchronous
blocking) | Native support for async/
await |
| Debugging experience | Difficult (relies on echo
printing or writing to log
files) | Efficient (supports IDE
breakpoints, step-by-step
debugging) |
| Applicable scenarios | Claude Code interactive
mode/Headless CLI | Embedded applications built
on Agent SDK |

Table 8.3: Comparison of Shell Hooks and Agent SDK Hooks

If you wish to use **Claude Code** directly (whether in terminal interactive mode or Headless CLI mode), it is recommended to use **Shell Hooks**; if you are developing custom applications, integrating services, or building complex automated workflows with the Agent SDK, it is recommended to use **Agent SDK Hooks**.

Four Layers of Security Defense

The Agent SDK has established a defense-in-depth system composed of four layers of security defense, as shown in *Figure 8.2*.

The permission mode sets the default posture, the tool whitelist bounds which tools exist at all, the canUseTool callback is a per-call gate with full argument context, and PreToolUse Hooks are the last deterministic checkpoint before execution, with the whitelist and permission mode carrying most of the weight in a typical application.

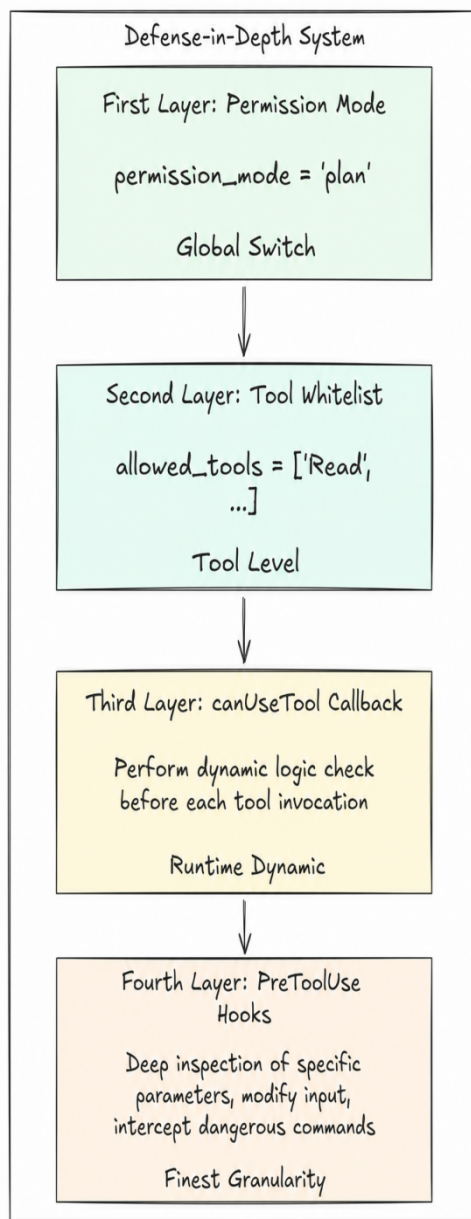


Figure 8.2 Defense-in-Depth System of the Agent SDK

The runtime permission callback is unique to the SDK. Note the spelling: the parameter is `can_use_tool` in Python and `canUseTool` in TypeScript. This chapter's prose uses the

TypeScript form when speaking about the mechanism in general, but the Python examples register `can_use_tool` — follow the code, not the prose, when you are typing it out.

The following code demonstrates how to build an intelligent "gatekeeper" that dynamically decides whether to grant access before tool execution based on specific content.

```

async def can_use_tool(tool_name: str, tool_input: dict) -> dict:
    """Runtime permission check callback"""

    # Scenario 1: File write protection
    # Prevent sensitive configuration files from being tampered with
    if tool_name in ["Write", "Edit"]:
        file_path = tool_input.get("file_path", "")
        if ".env" in file_path or "secrets" in file_path:
            return {
                "allowed": False,
                "reason": "Access to sensitive files denied"
            }

    # Scenario 2: Blocking Network Commands
    # Even if Bash is whitelisted, specific network operations
    # must still be prohibited
    if tool_name == "Bash":
        command = tool_input.get("command", "")
        if any(cmd in command for cmd in ["curl", "wget", "ssh"]):
            return {
                "allowed": False,
                "reason": "Network commands are not permitted"
            }

    return {"allowed": True}

options = ClaudeAgentOptions(
    permission_mode="acceptEdits",
    allowed_tools=["Read", "Write", "Edit", "Grep", "Glob"],
    can_use_tool=can_use_tool,
    hooks={
        "PreToolUse": [HookMatcher(matcher="*", hooks=[audit_all_tools])]
    }
)

```

The collaborative logic of the four security layers is as follows: `permission_mode` establishes the global baseline, `allowed_tools` eliminates redundant tools, `canUseTool` performs dynamic validation on each invocation at runtime, and `PreToolUse Hooks` are responsible for the most granular parameter inspection, input correction, and audit log recording.

Structured Output: Enforcing JSON Schema

When building applications, you often need **Claude** to output structured data rather than free-form text. The Agent SDK supports JSON Schema-based validation via the `output_format` parameter. See the following code example.

```
from pydantic import BaseModel

# Security report model

class SecurityReport(BaseModel):
    summary: str
    issues: list[dict]      # Structure example: [{severity, file, line,
                           # description}]
    risk_score: float       # Range: 0.0~10.0

options = ClaudeAgentOptions(
    output_format={
        "type": "json_schema",
        "schema": SecurityReport.model_json_schema()
    },
    max_turns=10,
    allowed_tools=["Read", "Grep", "Glob"],
)

async for message in query(prompt="Conduct a security review of src/",
    options=options):
    if isinstance(message, ResultMessage) and message.structured_output:
        report = SecurityReport.model_validate(message.structured_output)

        # report is a type-safe Pydantic object
        print(f"Risk score: {report.risk_score}")
        for issue in report.issues:
            print(f" [{issue['severity']}] {issue['file']}:{
                issue.get('line', '?')} - {issue['description']}")
```

In a TypeScript environment, you can achieve the same functionality using the Zod library. Please see the following code example.

```
import { z } from 'zod';

const SecurityReport = z.object({
  summary: z.string(),
  issues: z.array(z.object({
    severity: z.enum(['critical', 'high', 'medium', 'low']),
    file: z.string(),
    description: z.string(),
  })),
  riskScore: z.number().min(0).max(10),
});

const options: ClaudeAgentOptions = {
  outputFormat: {
    type: 'json_schema',
    schema: z.toJSONSchema(SecurityReport),
  },
};
```

Claude will automatically format the output as JSON conforming to the schema above. If the initial generated content fails validation, the Agent SDK will automatically trigger a retry mechanism (with the maximum number of retries controlled by internal policy); if the limit is exceeded and validation still fails, the system will return the `error_max_structured_output_retries` error.

In Practice: Building a Code Analysis Web Service

Below is a complete code analysis web service example that integrates all the knowledge covered earlier. This script demonstrates how to configure a Claude Agent, receive analysis results in a streaming fashion, and output real-time feedback in a web service scenario (simulated via the command line).

```
#!/usr/bin/env python3
"""Code Analysis Agent Service – Complete Running Example"""

import asyncio
import json
from datetime import datetime
```

```

from claude_agent_sdk import query, ClaudeAgentOptions

async def analyze_codebase(directory: str, focus: str = "general"):
    """
    Analyze the codebase in the specified directory.

    Args:
        directory: The local directory path to be analyzed
        focus: Key area of analysis - "security" / "performance" /
              "quality" / "general"
    """
    # Define System Prompts for Different Focus Areas
    focus_prompts = {
        "security": "Focus on security vulnerabilities: SQL injection,
                    XSS, hardcoded sensitive information,
                    lack of access control, etc.",
        "performance": "Focus on performance issues: N+1 queries, memory
                        leaks, redundant computations, missing caching strategies,
                        etc.",
        "quality": "Review naming, duplication, and test coverage.
                   Skip anything the linter already enforces.",
        "general": "Conduct a comprehensive code review: covering
                   security, performance, code quality, and architectural
                   soundness."
    }

    options = ClaudeAgentOptions(
        model="claude-sonnet-4-6",
        max_turns=15,
        max_budget_usd=0.50,
        allowed_tools=["Read", "Grep", "Glob"],
        permission_mode="plan",          # Read-only mode
        cwd=directory,
        system_prompt={"type": "preset", "preset": "claude_code",
                       "append": focus_prompts.get(focus, focus_prompts["general"])},
    )

    # Initialize the collector
    output_text = []
    tools_used = []
    metadata = {}

```

```

# Execute streaming query
async for message in query(
    prompt=f"Analyze the code of the current project,
        {focus_prompts.get(focus, '')}. Please output the analysis
        report directly in Markdown format.",
    options=options,
):
    if isinstance(message, AssistantMessage):
        for block in message.content:
            if hasattr(block, 'text'):
                output_text.append(block.text)
                # [Key Point] Streaming Output: In actual web services,
                # this should yield SSE events
                print(block.text, end="", flush=True)
            elif hasattr(block, 'name'):
                tools_used.append(block.name)

    elif isinstance(message, ResultMessage):
        metadata = {
            "session_id": message.session_id,
            "cost_usd": message.total_cost_usd,
            "turns": message.num_turns,
            "duration_ms": message.duration_ms,
            "success": not message.is_error,
        }

# Return structured results
return {
    "report": "\n".join(output_text),
    "tools_used": tools_used,
    "metadata": metadata,
}

async def main():
    import sys
    directory = sys.argv[1] if len(sys.argv) > 1 else "." # A real tool
    # should resolve this and check it exists
    focus = sys.argv[2] if len(sys.argv) > 2 else "general"

```

```

print(f"Analyzing {directory} (focus: {focus})...\n")
result = await analyze_codebase(directory, focus)

print(f"\n\n--- Session Statistics ---")
print(f"Status: {'Success' if meta['success'] else 'Failure'}")
print(f"Cost: ${result['metadata'].get('cost_usd', 0):.4f}")
print(f"Duration: {result['metadata'].get(
    'duration_ms', 0) / 1000:.1f}s")
print(f"Number of Turns: {meta.get('turns', 0)}")
print(f"Tools Used: {' , '.join(result['tools_used'])}")

if __name__ == "__main__":
    asyncio.run(main())

```

In a real-world web service architecture, the asynchronous generator for queries can be directly connected to **SSE (Server-Sent Events)** to achieve low-latency real-time data push.

Below is a complete example built on FastAPI. The shape that matters: an async request handler that iterates the agent's message stream and forwards each chunk to the client as it arrives, rather than waiting for the run to finish. That is what suits an SDK which yields messages instead of returning once, and it lets the frontend render the work as it happens instead of showing a spinner.

```

from fastapi import FastAPI
from fastapi.responses import StreamingResponse

app = FastAPI()

@app.post("/api/analyze")
async def analyze(request: AnalyzeRequest):
    async def event_stream():
        async for message in query(prompt=request.prompt, options=options):
            if isinstance(message, AssistantMessage):
                for block in message.content:
                    if hasattr(block, 'text'):
                        yield f"data: {json.dumps({'type': 'text', 'content':
block.text})}\n\n"
            elif isinstance(message, ResultMessage):
                yield f"data: {json.dumps({'type': 'done', 'cost':
message.total_cost_usd})}\n\n"

```

```
return StreamingResponse(event_stream(), media_type="text/event-stream")
```

Agent SDK and Headless CLI: How to Choose

Although both the Agent SDK and Headless CLI allow Claude to run in unattended environments, their applicable scenarios are clearly delineated (see *Table 8.4*). Choosing the wrong tool often leads to overly complex architectures or limited functionality.

| Scenario | Recommended approach | Reason |
|---|----------------------|---|
| Single-step tasks in CI/CD pipelines | Headless CLI | Zero-dependency deployment, directly embedded in YAML configuration, no need to install Python/Node runtime |
| Shell script automation | Headless CLI | Perfectly fits pipeline composition, simple and straightforward |
| Building web/desktop AI applications | Agent SDK | Supports streaming output (SSE), fine-grained state management, and frontend interaction |
| Custom tool required | Agent SDK | Supports <code>@tool</code> decorator to register local functions, enabling closed-loop complex business logic |
| Multi-turn conversation/continuous session | Agent SDK | Built-in session management, automatically maintains context history, no need to manually concatenate messages |
| Requires fine-grained control of message flow | Agent SDK | Based on async generators, allows interception, modification, or routing of messages in chunks |
| Manage multiple instances concurrently | Agent SDK | Programmatic concurrency control (e.g., <code>asyncio.gather</code>), easily handles high-concurrency requests |

Table 8.4: Comparison of Scenarios for Agent SDK and Headless CLI

A rule of thumb, with the caveat that bash can do more than people credit: reach for the CLI when the logic fits in a shell command or two. Reach for the Agent SDK when you need complex branching, looping, batch data processing, or tight coupling with other executable code — that is where staying in the shell starts costing more than it saves.

Note

Kage's notes

The Agent SDK lowers the barrier to building AI applications, but "easy" does not mean "simple." When embedding Claude into real products, developers must properly address the following key issues: error retries (to handle network fluctuations and API rate limits), cost tracking (monitoring token consumption for each call), user isolation (ensuring conversations between different users do not interfere), response timeouts (establishing interruption strategies for long-running tasks), and circuit breaker mechanisms (automatically halting calls after consecutive failures). Mastering these engineering details is essential for upgrading an Agent SDK demo prototype to a production-grade application.

Summary

The Agent SDK transforms Claude Code from a standalone terminal tool into an AI component that can be embedded in any application. This transformation is built on two core APIs—the query function and `ClaudeAgentOptions`.

The query function is the entry point of the Agent SDK, functionally equivalent to the Headless mode's `claude -p` command, but returns a message stream as an asynchronous generator, supporting real-time processing of each step of Claude's output. This message stream includes four core types: `system/init` (initialization signal), `assistant` (Claude's response, which may include text content and tool invocation requests), `user` (feedback results after tool execution), and `result` (task termination signal, carrying metadata such as cost, number of rounds, and elapsed time).

`ClaudeAgentOptions` provides a programming interface equivalent to the Headless CLI parameters, covering model selection, tool permissions, permission modes, prompt customization, session management, and structured output. Compared to the command line, the core advantage of the programming interface is dynamic configuration: developers can dynamically adjust permission modes based on user roles, flexibly switch models according to task types, or optimize the next prompt in real time based on the results of the previous round.

Custom tools are implemented via the `@tool` decorator and an in-process MCP server, enabling Claude to call any custom function, whether querying a database, invoking internal APIs, or sending notifications. Meanwhile, four layers of security (permission modes, tool whitelists, `canUseTool` callbacks, and `PreToolUse` Hooks) ensure that Claude only does what it is supposed to do within the application.

From `CLAUDE.md` configuration, Skills definition, and Hooks interception, to MCP integration, and finally to comprehensive control via Headless mode and Agent SDK, each deeper layer brings you closer to building a mature "product" rather than just a "tool." In *Chapter 9*, we will explore how to package and distribute these capabilities, with an in-depth analysis of the Plugins ecosystem.

Questions for Reflection

1. If you need to build an internal "code impact analysis" tool for your company (input a requirement description, output a list of affected code modules and an assessment of modification difficulty), how would you design its system architecture? Which core features of the Agent SDK should be emphasized in this scenario?
2. The description field of a custom tool is the operation guide Claude reads before deciding whether to call it. Write down what a good one has to answer: what the tool does, when to use it and when not to, what each parameter means and in what format, what it returns, and any side effects or preconditions. Then work out what a poor description costs you — at best a wrong tool call, malformed arguments, or a tool skipped that should have been used; at worst an irreversible side effect from a call that should never have been made.
3. What is the relationship among the four security layers (permission modes, tool whitelists, `canUseTool` callbacks, `PreToolUse` Hooks)? In your application scenario, which layers are essential and which can be omitted?

9

From Personal Craft to Shared Asset: The Plugin Ecosystem

"If I have seen further, it is by standing on the shoulders of giants." — Isaac Newton

Recently, Xiaoxue discovered a GitHub repository in the open-source community titled "*Claude Code Best Practices for React Full-Stack Teams*," which was highly appealing. Upon closer inspection, she found it included a set of meticulously designed Skills covering code review, component documentation generation, and test case writing; in addition, the repository provided Hooks scripts for automatic formatting and PR formatting, along with a complete `.mcp.json` configuration file, pre-integrated with the GitHub API and Linear project management tool.

"If only this solution could be used out of the box," Xiaoxue sighed.

However, reality was not so smooth. In this repository, Skills were in one directory, Hooks scripts in another, `.mcp.json` was a standalone file, and several custom command definition files were scattered elsewhere. To use this configuration, she had to manually copy all files to the designated locations, figure out the installation method for each part, and hope she didn't miss any dependencies. In the end, she gave up—just figuring out the installation order took her the whole afternoon.

"This is exactly the problem Plugins aim to solve," Kage said. *"They package a complete set of Claude Code capabilities into a unit that can be installed with one click. Just like npm for the JavaScript ecosystem or pip for Python, Plugins are the standardized distribution mechanism for Claude Code capabilities (see Figure 9.1)."*

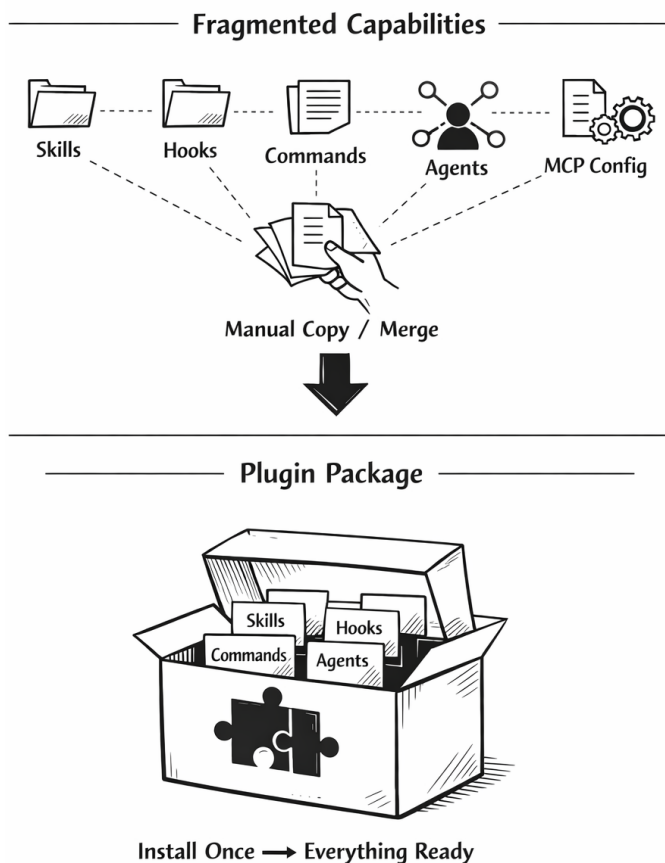


Figure 9.1: Plugins as the Standardized Distribution Mechanism for Claude Code Capabilities

The Role of Plugins: Encapsulation and Distribution of Capabilities

Before delving into Plugins, it's necessary to review the tool stack we've built so far. Up to now, we've discussed six independent extension mechanisms: managing project memory with `CLAUDE.md`, task delegation via sub-agents, encapsulating domain workflows with Skills, providing event-driven control through Hooks, connecting to external services with MCP, and programmatic invocation via Headless mode/Agent SDK. Although each mechanism serves its own purpose, they share a common shortcoming—they are isolated from each other and difficult to distribute as a whole.

When you spend weeks refining a complete development workflow for your team and try to share it with another team, reality is discouraging: Skills need to be copied to the `.claude/skills/` directory, Hooks configurations must be merged into `settings.json`, the MCP server requires separate configuration, custom commands must be moved to `.claude/commands/`, and sub-agent definitions are scattered elsewhere. The recipient not only has to figure out the installation steps for each component one by one, but also deal with potential naming conflicts. This is no longer simple knowledge sharing; it becomes a tedious migration task.

Plugins themselves do not introduce new features, but provide **a unified packaging format and installation mechanism**. A Plugin can include Skills, Hooks, Commands, Agents, and MCP configurations, and with a single command, all components are in place.

```
/plugin install team-toolkit@our-company
```

As shown in *Table 9.1*, a Plugin can contain multiple component types, covering all the extension mechanisms discussed in previous chapters.

| Component | Function | Corresponding chapter |
|------------|---------------------------|-----------------------|
| Commands | Custom slash commands | <i>Chapter 3</i> |
| Agents | Pre-configured sub-agents | <i>Chapter 3</i> |
| Skills | Domain knowledge package | <i>Chapter 4</i> |
| Hooks | Automated behavior | <i>Chapter 5</i> |
| MCP server | External tool integration | <i>Chapter 6</i> |

Table 9.1: Component Types a Plugin Can Contain

In other words, all the extension capabilities we learned separately before can now be packaged and distributed together as components of a Plugin.

Physical Structure of a Plugin

A Plugin is essentially a folder following a specific directory structure (usually hosted as a Git repository), with its core being the `plugin.json` manifest file located in the `.claude-plugin/` subdirectory.

Below is an example of a typical Plugin directory structure (using react-workflow as an example).

```

react-workflow/                                     ← Plugin root directory
├── .claude-plugin/
│   └── plugin.json                                ← [Required] Plugin manifest
file
├── commands/                                       ← Slash command definitions
│   ├── review.md
│   └── deploy.md
├── agents/                                         ← Sub-agent definitions
│   ├── security-scanner.md
│   └── quick-fix.md
├── skills/                                         ← Skills domain knowledge
packages
├── react-patterns/
│   ├── SKILL.md
│   └── chapters/
│       ├── hooks.md
│       └── performance.md
├── hooks/                                         ← Hooks configuration and scripts
│   ├── hooks.json
│   ├── check-bash.sh
│   └── auto-format.sh
├── .mcp.json                                       ← MCP server configuration
reference
└── README.md                                     ← Documentation

```

There is an important physical constraint to pay special attention to: **only** `plugin.json` **must be placed inside the specific** `.claude-plugin/` **subdirectory**. All other functional directories (such as `commands`, `agents`, `skills`, `hooks`) and configuration files (`.mcp.json`) should be located directly in the root directory of the Plugin.

This structural design ensures that the Plugin root directory itself is a valid Claude Code project directory. This means developers can develop and test directly within the Plugin directory, without needing to switch to a special **development mode**.

plugin.json: The Plugin's Identity Card

The following shows a `plugin.json` snippet. It lives in the `.claude-plugin/` directory at the root of the plugin's own repository — not in your project's `.claude/` directory, and not at the repository root.

```
{
  "name": "react-workflow",
  "version": "1.2.0",
  "description": "Complete React/TypeScript development workflow
    for Claude Code",
  "author": "YourName",
  "repository": "https://github.com/yourname/react-workflow",
  "license": "MIT",
  "keywords": ["react", "typescript", "workflow", "code-review"]
}
```

Table 9.2 provides a detailed list of all fields in `plugin.json` and their descriptions.

| Field | Required | Description |
|-------------|----------|--|
| name | Yes | Unique identifier for the Plugin, used for installation commands and internal references |
| version | Yes | Semantic version number (SemVer) |
| description | Yes | Short description, will be displayed in the Plugin list |
| author | No | Author's name or team name |
| repository | No | Source code repository address |
| license | No | Open source license agreement |
| keywords | No | Search keywords for retrieval |

Table 9.2: *plugin.json* Field Descriptions

Special attention should be paid to the `name` field. It is not only a display name — it determines the plugin's namespace, and every command, agent, and skill the plugin ships is

prefixed with it to avoid collisions. Choose something specific. A name like `tools` is too generic to disambiguate anything once a marketplace has more than a handful of entries; it is defensible only for a broad internal toolkit a team installs wholesale, and even then a team prefix reads better.

Specific Format of Components

Command Files

Command files are located in the `commands/` directory and use Markdown format. The core rules are as follows.

- **The file name is the command name:** for example, `review.md` corresponds to the `/review` command.
- **Front matter metadata:** The top of the file must include YAML-format FrontMatter, defining the command's metadata.
- **Content as instruction:** The main body of the file is the specific instruction sent to the AI.

An example of the contents of `commands/review.md` is as follows.

```
---
name: review
description: Conduct a code review of the current file or directory
---
```

When the user runs

```
`/review [target]`
```

, please follow the code review process below.

Review Focus Points

1. Code Quality: Check naming conventions and adherence to the DRY principle.
2. Security Issues: Identify missing input validation and potential injection risks (such as SQL injection, XSS).
3. Performance Issues: Detect unnecessary loops, potential memory leaks, or inefficient algorithms.

Output Format

Please strictly categorize the review results as follows:

- Critical (issues that must be fixed immediately)

- Warning (potential risks that are recommended to be fixed)
- Suggestion (optional code optimizations or improvements)

Sub-agent Files

Sub-agent files are located in the `agents/` directory and are used to define specialized AI agents with specific roles, permissions, and model configurations. Similar to command files, they also use Markdown format, but their Front Matter offers more powerful configuration capabilities, allowing precise control over the agent's tool permissions and underlying model.

An example of the contents of `agents/security-scanner.md` is as follows.

```
---
name: security-scanner
description: Scan for security vulnerabilities in code
tools: Read, Grep, Glob
model: sonnet
---

You are a seasoned security expert specializing in identifying potential security vulnerabilities in codebases.

## Core Responsibilities

Please pay special attention to the following risk points:



- Injection Attacks: Detect risks such as SQL injection and command injection.
- Cross-Site Scripting (XSS): Check for unescaped user input and output.
- Credential Leakage: Search for hardcoded API keys, passwords, or private keys.
- Access Control: Verify whether access control logic is robust.

```

Hooks Configuration File

The hooks configuration file is located in the `hooks/` directory and is used to define the automated behavior of Plugins during specific lifecycle events. Its format is fully compatible with the Hooks configuration in the global `settings.json` introduced in *Chapter 5*, allowing developers to intercept and enhance the tool's execution flow through scripts.

An example of the contents of `hooks/hooks.json` is as follows.

```
{
  "hooks": [
    {
      "event": "PreToolUse",
```

```
    "matcher": "Bash",
    "command": ["bash", "./hooks/check-bash.sh"]
  },
  {
    "event": "PostToolUse",
    "matcher": "Write",
    "command": ["bash", "./hooks/auto-format.sh"]
  }
]
```

MCP Configuration File

The MCP configuration file is typically located in the project root directory and named `.mcp.json`. This file follows the exact same format as the project-level global MCP configuration and is used to declare the external service connections required by the Plugin.

An example of the contents of `.mcp.json` is as follows.

```
{
  "mcpServers": {
    "postgres": {
      "command": "npx",
      "args": ["-y", "@anthropic/mcp-server-postgres"],
      "env": {
        "DATABASE_URL": "${DATABASE_URL}"
      }
    }
  }
}
```

Installation and Lifecycle Management

Installation Sources

Plugin installation is managed uniformly via the `/plugin` command, supporting the following three sources.

```
# Install from a community marketplace. A marketplace is a catalog of plugins
hosted in a Git repository;
# you must add it with /plugin marketplace add before anything can be installed
from it.
```

```
/plugin install react-workflow@community

# Install from a GitHub repository
/plugin install github.com/username/react-workflow

# Install from a local directory (use case: during Plugin development and
debugging, allowing real-time testing after code changes without republishing)
/plugin install ./path/to/my-plugin
```

When the installation command is executed, Claude Code will read and parse the `plugin.json` file in the Plugin directory, register Commands, Agents, and Skills into the system, merge the defined Hooks configuration into the current user's Hooks chain, and add the MCP server to the available list. All these operations are fully reversible: when the `uninstall` command (e.g., `/plugin uninstall <name>`) is executed, the system will precisely trace and undo all changes made during installation.

Routine Management

You can use the following commands to view, uninstall, and update installed Plugins.

```
/plugin list                # List all installed Plugins
/plugin uninstall react-workflow  # Uninstall the specified Plugin (e.g.,
react-workflow)
/plugin update react-workflow    # Update the specified Plugin to the latest
version
```

Local Development and Testing

When developing Plugins, you can use the `--plugin-dir` parameter to directly load a local directory, thereby skipping the standard installation process.

```
claude --plugin-dir ./my-plugin-dev
```

The advantage of this approach is that after modifying the code, you can immediately restart and test the results; there is no need to repeatedly execute the install and remove cycle, which significantly improves development and debugging efficiency.

Storage Location

By default, plugins are stored under `~/.claude/plugins/`. That location is not configurable — plugins are copied into a cache directory when installed, and there is no setting that redirects it. `CLAUDE_PLUGIN_ROOT` is sometimes mistaken for such a setting; it is not one. It is a runtime

variable that Claude Code sets automatically for each plugin, resolving to that plugin's own installation directory, and its purpose is intra-plugin path references — writing `{CLAUDE_PLUGIN_ROOT}/scripts/check.sh` inside a hook command or an MCP server config so the plugin can find its own files wherever it happens to be installed.

Namespaces: Multiple Plugins Coexisting

What happens when multiple Plugins are installed simultaneously and contain Skills or Commands with the same name? The Plugins system automatically resolves conflicts through a **namespace mechanism**.

Commands, agents, and skills installed via a plugin are automatically given a fully qualified name in the form `plugin-name:component-name`, which keeps them globally unique. In the `react-workflow` example below, `review` is a slash command, and `code-reviewer` is a subagent; both pick up the plugin name as their prefix.

- The original `review` Command becomes `/react-workflow:review`.
- The original `code-reviewer` Skill becomes `react-workflow:code-reviewer`.
- The original `security-scanner` Agent becomes `react-workflow:security-scanner`.

If there are no naming conflicts, you can use the short command name directly (such as `/review`). Claude Code will automatically resolve it to the unique corresponding component. Only when the system detects that multiple Plugins provide components with the same name will you be required to use the full namespace qualifier (such as `/plugin-a:review`) to specify explicitly.

The engineering value of this mechanism is that it eliminates collaboration concerns, allowing teams to confidently install and combine multiple Plugins without needing to coordinate in advance or worry about naming conflicts; it also reduces maintenance costs, as Plugin developers do not need to avoid other Plugins' names during development.

In Practice: Building a Team Capability Pack

Theory is no substitute for hands-on practice. Suppose you are part of a team using the React, TypeScript, and PostgreSQL technology stack, and you now need to build an internal Plugin that integrates code review, security scanning, and automatic formatting features.

Complete Directory Structure

The complete directory structure of the internal Plugin is as follows.

```
team-toolkit/
├── .claude-plugin/
│   └── plugin.json
├── commands/
│   ├── review.md           # Code review command definition
│   └── test.md             # Test run command definition
├── agents/
│   ├── security-scanner.md # Security scanner sub-agent configuration
│   └── quick-fix.md        # Quick fix sub-agent configuration
├── skills/
│   └── react-patterns/
│       └── SKILL.md        # React best practices knowledge base
├── hooks/
│   ├── hooks.json         # Hook trigger configuration
│   ├── check-bash.sh      # Bash command security check script
│   └── auto-format.sh     # Auto-format execution script
├── .mcp.json              # MCP configuration (database and GitHub
links)
└── README.md             # Project documentation
```

plugin.json

The contents of plugin.json are as follows.

```
{
  "name": "team-toolkit",
  "version": "2.0.0",
  "description": "Team standardized development toolkit: integrates
    code review, automated testing, and security scanning features",
  "author": "Platform Team",
  "repository": "https://github.com/our-company/team-toolkit",
  "license": "MIT",
  "keywords": ["team", "devops", "code-review", "security"]
}
```

Security Scanning Sub-Agent

The content of the security scanning sub-agent agents/security-scanner.md is as follows.

```
---
name: security-scanner
description: Scan for security vulnerabilities in the codebase and generate
structured reports
tools: Read, Grep, Glob
model: sonnet
---
```

You are a senior security expert specializing in identifying potential security vulnerabilities in code.

Scan Scope

1. Injection Vulnerabilities: Including SQL injection, command injection, and XSS.
2. Authentication Issues: Covering weak password policies, hardcoded credentials, and session management flaws.
3. Data Exposure: Involving sensitive information logged in outputs and insecure data transmission.
4. Access Control: Including missing permission checks and path traversal vulnerabilities.

Workflow

File Discovery: Use Glob to obtain all source code files.

2. Pattern Matching: Use Grep to search for suspicious code patterns (such as eval, dangerouslySetInnerHTML, exec, etc.).
3. In-depth Analysis: Use Read to examine the context of suspicious code and confirm the authenticity of vulnerabilities.
4. Report Generation: Output a structured report clearly listing each issue's file name, line number, and risk level.

Execution Principles

- Conclusive Evidence: Only report issues with concrete code evidence; subjective speculation is strictly prohibited.
- Remediation-Oriented: Not only point out problems, but also provide specific and feasible remediation suggestions.

- Rigorous Labeling: For potential risks that cannot be fully confirmed, clearly label them as "suspected."

Hooks: Security Check Script

The content of the security check script hooks/check-bash.sh is as follows.

```
#!/bin/bash
# Bash Command Security Interception Script
# Used to block the execution of dangerous commands
in Claude Code before they are run

INPUT=$(cat)
COMMAND=$(echo "$INPUT" | jq -r '.tool_input.command // empty')

DANGEROUS_PATTERNS=("rm -rf /" "rm -rf ~" "sudo rm" "> /dev/" "chmod 777")
for pattern in "${DANGEROUS_PATTERNS[@]}; do
    if echo "$COMMAND" | grep -qF "$pattern"; then
        cat << EOF
{"decision": "deny", "reason": "Blocked dangerous pattern: $pattern"}
EOF
        exit 0
    fi
done
echo '{"decision": "allow"}'
```

Release Process

A Plugin is essentially a standard Git repository. The process of releasing a new version involves committing the code, tagging it, and pushing it to the remote repository.

To publish the plugin to a Git repository so teammates can install from it, run the following in the plugin root directory (team-toolkit).

```
cd team-toolkit
git init
git add .
git commit -m "v2.0.0: Integrated security scanning and auto-formatting features"
git tag v2.0.0

git remote add origin https://github.com/our-company/team-toolkit
```

```
git push -u origin main --tags
```

Teammates can then install it with the command below. Note what makes this plugin internal: it is the repository permissions, not anything in the plugin system. Whoever installs it needs read access to that repository already — and if the repository is public, the plugin is not internal at all.

```
/plugin install github.com/our-company/team-toolkit
```

Private Marketplace and Enterprise Management

Passing repository URLs around one at a time stops scaling once you have enough plugins, or enough teams, that nobody can say what exists. The plugin system supports a private marketplace for exactly this: one catalog, one place to add or retire an entry, and version control over the index itself. A marketplace is just a Git repository, so there is nothing to buy.

Building a Private Marketplace

The private marketplace itself is also a standard Git repository, and the manifest goes in the same place the plugin manifest does: `.claude-plugin/marketplace.json`, not the repository root. This mirrors the convention described in the *Physical Structure of a Plugin* section. Put it at the root and the marketplace simply will not be discovered.

An example structure is as follows.

```
{
  "name": "our-company",

  "description": "Internal Plugin Marketplace",

  "plugins": [
    {
      "name": "team-toolkit",
      "description": "Team Standardized Development Toolkit",
      "repository": "https://github.com/our-company/team-toolkit",
      "version": "2.0.0"
    },
    {
      "name": "db-tools",
```



```
    "description": "Database Operations Toolkit",  
    "repository": "https://github.com/our-company/db-tools",  
    "version": "1.2.0"  
  }  
]  
}
```

Using a Private Marketplace

Team members only need to execute a single command to add the company's private marketplace to their local configuration.

```
/plugin marketplace add our-company/claude-plugins
```

Once registration is complete, members can directly install a specified Plugin from the company marketplace using the format `PluginName@MarketplaceSource`, without needing to remember the full repository address.

```
/plugin install team-toolkit@our-company
```

Organization-Level Plugin Management

In the enterprise edition, administrators can pre-install Plugins at the organizational level. Once configured, all members can use them directly without manual intervention. The core capabilities of this mechanism are as follows:

- **Unified Distribution:** Administrators only need to configure in the backend, and the Plugin will be automatically distributed to all member environments. This feature is especially critical for enforcing team standards (such as integrating security check Hooks or unified code review standards).
- **Version Control:** Organization-level Plugins can be locked to a specific version, preventing members from arbitrarily upgrading to incompatible new versions that could disrupt team workflows. The authority to update and promote versions is strictly limited to administrators.
- **Audit Visibility:** The system provides complete tracking of the installation status and usage of organization-level Plugins, meeting enterprise compliance audit and security monitoring requirements.

Plugin Design Principles

Single Responsibility

Each Plugin should focus on solving a specific type of problem, avoiding the creation of bloated "all-in-one toolboxes." Keeping functionality focused helps reduce maintenance costs and improve user experience.

Recommended designs are as follows:

- `react-workflow`: Dedicated to optimizing React development workflows.
- `security-scanner`: Focused on code security scanning.
- `db-tools`: Provides only database operation assistance.

Negative example:

- `everything-plugin`: Tries to do everything but lacks depth, resulting in unclear positioning and difficult maintenance.

Incremental Iteration

Follow the minimum viable product philosophy: start with the core capability and improve it in small, rapid iterations. Version it with semantic versioning, and be explicit about what each level means for a plugin — major for a breaking change to a command's interface or a hook's contract, minor for added components, patch for fixes that change no interface.

- `v1.0.0`: Release core functionality (such as a single efficient code review command).
- `v1.1.0`: Expand sub-capabilities (such as integrating a security scanning sub-agent).
- `v1.2.0`: Enhance domain expertise (such as adding a React best practices guide).
- `v2.0.0`: Deepen automation (such as introducing MCP integration and automated Hooks workflows).

Do not delay the initial release waiting for "complete functionality." A Plugin with just one efficient command is far better than a bloated Plugin with ten half-finished features. Release a usable version as early as possible and drive subsequent iterations through user feedback.

Least Privilege

Sub-agents should strictly follow the principle of requesting only the permissions necessary to complete their core tasks.

If a sub-agent is only responsible for code review and analysis, its permissions should be strictly limited to read-only operations (such as `Read`, `Grep`, `Glob`). It is strictly forbidden to grant such agents permissions to write files (`Write`) or execute system commands (`Bash`).

Excessive permission scopes significantly increase the trust cost for users. Users must fully trust the developer before installing high-privilege Plugins, which directly leads to lower installation conversion rates.

Documentation Is Essential

A plugin without documentation will not be adopted. The README is where a teammate decides whether to install it, so it has to answer three things without them reading the source: what each component does, how to invoke it, and what it assumes about the project it is installed into. The elements below cover that:

- **Quick Installation:** Provide a one-line installation command to lower the entry threshold.
- **Feature List:** Clearly list all available Commands, sub-agents, and Skills, with brief descriptions.
- **Configuration Guide:** Explain in detail the required environment variables (especially MCP server connection configuration) to prevent users from being blocked by missing configurations.
- **Changelog:** Record the changes, new features, and bug fixes for each version to help users assess the impact of upgrades.

When to Package Capabilities as a Plugin

Not all Skills or Hooks need to be packaged as Plugins. The core criterion for choosing a distribution method is the scope of the target audience. Refer to *Table 9.3* for the decision guide.

| Distribution scope | Recommended approach | Core advantages |
|-------------------------|--|---|
| Single project | Project-level configuration (<code>.claude/</code> directory) | Code co-location: configuration and code are versioned together, changes are traceable |
| Personal cross-project | User-level configuration (<code>~/.claude/</code> directory) | Personalization: adapts to individual preferences, does not affect other team members |
| Cross-team/organization | Public/private Plugin | Standardization: supports one-click installation, easy cross-project reuse |

| Distribution scope | Recommended approach | Core advantages |
|-------------------------|---------------------------|---|
| Enterprise-wide rollout | Organization-level Plugin | Enforced management: administrators deploy centrally, no manual action required by employees |

Table 9.3: Recommended Distribution Methods

If you only need to serve team members of the current project, the simplest solution is to place the Skills and Hooks configuration files directly in the project's `.claude/` directory and commit them to Git. This approach not only allows members to automatically obtain the configuration after cloning the repository, achieving "zero friction" onboarding, but more importantly, the configuration is co-located with the code, so its modification process is naturally included in the code review workflow, and historical changes are clearly traceable.

However, when requirements go beyond a single project—such as sharing capabilities with other teams within the organization, publishing to the open-source community, or promoting as a standardized toolchain across the enterprise—then Plugins are the correct vehicle.

Note

Kage's Notes

The value of a Plugin equals its utility multiplied by the size of its user base. A Skill used only by an individual can only improve personal efficiency; but packaging it as a Plugin and promoting it to a 20-person team multiplies the value created by the same development effort twentyfold. Plugins are the key step in turning knowledge into reusable assets—they transform best practices accumulated by individuals into core assets that can be sustainably reused and empower the entire organization.

LSP Support and Future Evolution

In addition to Skills, Hooks, and MCP, Plugins also support a more advanced extension type—the **Language Server Protocol (LSP)**.

LSP is the standard protocol used by modern editors such as VS Code to implement features like syntax checking, code completion, and go-to-definition. Integrating LSP support into Plugins means Claude can be endowed with **real-time code diagnostics capabilities**.

Take the TypeScript Plugin as an example: when users write code with Claude, the Plugin can invoke the TypeScript compiler to inject real-time type error information directly into Claude's context. This enables Claude to detect and correct potential type errors while generating code.

This evolution elevates the dimension of Plugin capabilities from "instructing Claude how to act" to "enabling Claude to perceive the real-time state of code." Although LSP integration is still in its early stages, the future it points to is already clear: Plugins are not just containers for encapsulating knowledge—they are also the sensory extensions that allow Claude to perceive the world of code.

From a macro perspective, the Plugin ecosystem is gradually building a decentralized distribution network similar to npm: community marketplaces (such as `claude-plugins.dev`) have already aggregated thousands of public Plugins; enterprises coordinate internal toolchains through private marketplaces; and individual developers share best practices via GitHub repositories. The maturity of this ecosystem will directly determine whether Claude Code can evolve from "an efficient tool" into "a thriving platform."

Summary

Plugins are what makes the other four mechanisms distributable. They create no new capability — a plugin contains the same commands, agents, skills, and hooks you could write by hand — but they turn a directory of hard-won configuration into something a teammate installs with one command instead of reconstructing.

Structurally, a Plugin is a Git repository that follows a specific directory convention. Its core is the `plugin.json` manifest file located in the `.claude-plugin/` directory, while functional module directories (such as commands, agents, skills, hooks) are placed directly at the repository root. Installation is performed via the `/plugin install` command, supporting three sources: community marketplaces, GitHub repositories, and local directories. In addition, the namespace mechanism ensures that multiple Plugins can coexist without conflict.

In team practice, the value of a plugin lies less in technical complexity than in what it does to knowledge: it turns knowledge into a reusable asset. A code-review standard, a security hook, or a formatting configuration that took someone a week to get right becomes, once packaged, something the whole team installs in a second.

In previous chapters, Skills, Hooks, MCP, and custom commands were independent extension tools; the Plugin mechanism consolidates them into a versionable, distributable, and easy-to-maintain whole. This transformation enables the accumulation of knowledge in Claude Code to circulate as efficiently as code itself.

Chapter 10 will be the final chapter of the book. We will review how the mechanisms described above work together from an engineering practice perspective, and delve into debugging strategies, cost control, and methodologies for scaling up to team-level implementation.

Discussion Questions

1. Within your current team, which Claude Code-related configurations (such as Skills, Hooks, MCP) are scattered across different locations? If you were to consolidate them into a single Plugin, how would you plan its directory structure?
2. Both organization-level Plugins and project-level `.claude/` directory configurations have their pros and cons. In what scenarios would you prioritize the former over the latter?
3. Based on your field, design a dedicated Plugin: What should it be named? What core components should it include (such as Commands, Agents, Skills, Hooks, or MCP)? Who would be its primary user group?

10

From Individual to Team: Engineering Practices for Claude Code

"The craftsman who wishes to do his work well must first sharpen his tools."
— Confucius

It has been exactly three months since the engineering team fully adopted Claude Code.

At first, the experience was nothing short of "magical": individual productivity soared, features that used to take two days to deliver could now be completed in one, and the number of low-level errors caught during code review dropped significantly. Engineers were enthusiastic, eagerly sharing usage tips.

However, after three months, problems gradually began to surface.

First, the cost statements shocked the finance department; the API expenses last month were three times higher than expected. Upon checking the logs, it was discovered that the cause was an engineer using Claude to analyze the entire database's historical logs, with a single session lasting nearly five hours.

Second, cracks appeared in team collaboration. Documentation styles generated by different engineers varied widely: some meticulously crafted precise `CLAUDE.md` specifications, while others left not a single word. When new hires joined, they were completely lost amid the scattered practice records, unable to discern the team's best practices for Claude Code.

Even more concerning was the lack of traceability. When Claude made a wrong decision, the team often found itself trapped in a "black box" dilemma due to the absence of a transparent reasoning process, not knowing how to trace the root cause.

"We have the tool, but we have yet to establish engineering practices to match it," the company CTO pointed out. "These are two entirely different things."

Xiaobing asked: "So, what do engineering practices actually look like?"

Kage thought for a moment, then wrote six keywords: **cost, debugging, security, scale, instructions, collaboration.**

"The previous content focused on the functional boundaries and operational methods of Claude Code," he explained. "This chapter will shift perspective, to explore how to use these capabilities efficiently, robustly, and securely in real engineering environments, and ultimately roll them out across the entire team."

Cost: Make Tokens Work for You, Not Burn Your Money

"How exactly are Claude Code's fees calculated?" Xiaobing asked.

This question is far more complex than it appears on the surface. Claude Code's API call fees are essentially based on token usage: the sum of the prompt tokens sent in each call and the tokens generated in the reply, multiplied by the unit price for the corresponding model. However, behind a single user request, Claude Code may initiate a dozen or even dozens of API calls, including reading files, executing searches, logical reasoning, code generation, and secondary verification, each of which incurs a charge.

Understanding the Pricing Structure

As of the end of 2025, Claude model pricing is shown in *Table 10.1*.

Note

Model pricing changes periodically; the figures above are a snapshot. For current pricing, always refer to the official Anthropic pricing page (anthropic.com/pricing).

| Model | Input price | Output price | Cache write | Cache read |
|------------|-------------|--------------|-------------|------------|
| Haiku 4.5 | 1.00 | 5.00 | 1.25 | 0.10 |
| Sonnet 4.6 | 3.00 | 15.00 | 3.75 | 0.30 |
| Opus 4.6 | 5.00 | 25.00 | 6.25 | 0.50 |

Table 10.1 Claude Model Pricing (Unit: USD per million tokens)

Several key insights can be drawn from *Table 10.1*:

- **Output costs are much higher than input costs.** For example, with Opus, the output unit price is five times that of input. This means that, compared to controlling input length, strictly limiting Claude's output length can reduce costs more significantly.
- **There are huge price differences between models.** Opus is priced five times higher than Haiku. Therefore, not every task requires the most powerful model; you should choose according to the scenario.
- **Cache reads offer a significant cost advantage.** Their price is only one-tenth of the normal input price. Whenever possible, you should maximize cache hit rates to save on expenses.

There is another easily overlooked cost driver: **thinking**. On the models used in this chapter's pricing table, the lever to reach for is the effort level, which defaults to high and is set with `/effort` or through `/config`. Lowering it is the most direct way to cut thinking spend on work that does not need deep reasoning. Because the controls here move faster than print, check the current model-config documentation before wiring a fixed token budget into anything automated.

In scenarios where deep reasoning is not required, users can limit the thinking budget by setting the environment variable `MAX_THINKING_TOKENS=8000`, thereby significantly reducing costs.

Model Selection Strategy

Not every task requires the most powerful model. A practical tiered strategy is shown in *Table 10.2*.

| Task complexity | Recommended model | Reason |
|--|-------------------|---|
| Formatting, renaming | Haiku | Clear rules, no complex reasoning required |
| Documentation generation, simple testing | Haiku | Fixed pattern, optimal cost |
| Feature development, bug fixing | Sonnet | Requires understanding of context, but scope is limited |
| Code review, refactoring | Sonnet | Requires professional judgment, but there are standards to follow |

| Task complexity | Recommended model | Reason |
|---|-------------------|--|
| Architecture design, security audit | Opus | Requires deep thinking and a global perspective |
| Difficult troubleshooting, cross-module refactoring | Opus | Requires creative thinking and comprehensive reasoning |

Table 10.2: Model Selection Strategy by Task Complexity

In Headless mode, model selection is explicitly specified via the `--model` parameter. Here are examples for two typical scenarios.

```
# Simple task: use Haiku and set a very low budget cap
claude -p "Check the variable names in this function" --model claude-haiku-4-5 --max-budget-usd 0.05

# Complex task: perform architecture analysis, use Opus, and allow a higher budget
claude -p "Analyze the design issues of the entire payment system" --model claude-opus-4-6 --max-budget-usd 2.00
```

Here, `--max-budget-usd` is a hard limit to prevent runaway costs. In CI/CD automation scenarios, this parameter should be enforced as a standard configuration. Without this restriction, if a script's logic goes wrong (such as entering an infinite loop or causing infinite recursive calls), it could burn through hundreds or even thousands of dollars overnight.

Context Management Is the Core of Cost Control

Why do we repeatedly emphasize sub-agent and context isolation? Beyond the technical benefit of improving signal-to-noise ratio, there is a purely economic motivation.

The Claude API is stateless; each call must carry the complete conversation history. Imagine if you directly loaded a 5,000-line log file into the main conversation, those 5,000 lines would be resent with every subsequent API call. This means that every time Claude makes a decision, you are paying again for those 5,000 lines of logs. In contrast, if you delegate log analysis to a sub-agent, the massive log content is isolated within the sub-process's context, and the main conversation only needs to receive a few key conclusions. Therefore, sub-agents are not a luxury add-on, but a crucial economic decision-making tool.

The same logic applies to trimming `CLAUDE.md`. The official guidance is to target under 200 lines per `CLAUDE.md` file — longer files consume more context and, as the documentation puts it, reduce adherence. Every line in it is paid for on every single request of the session.

Prompt Caching

Claude Code enables prompt caching by default. Its core principle is illustrated in *Figure 10.1*: if the input prefix of the current request (usually including the system prompt, `CLAUDE.md`, and tool definitions) matches a previous request, that portion of data can be reused from the cache, at only one-tenth the normal price.

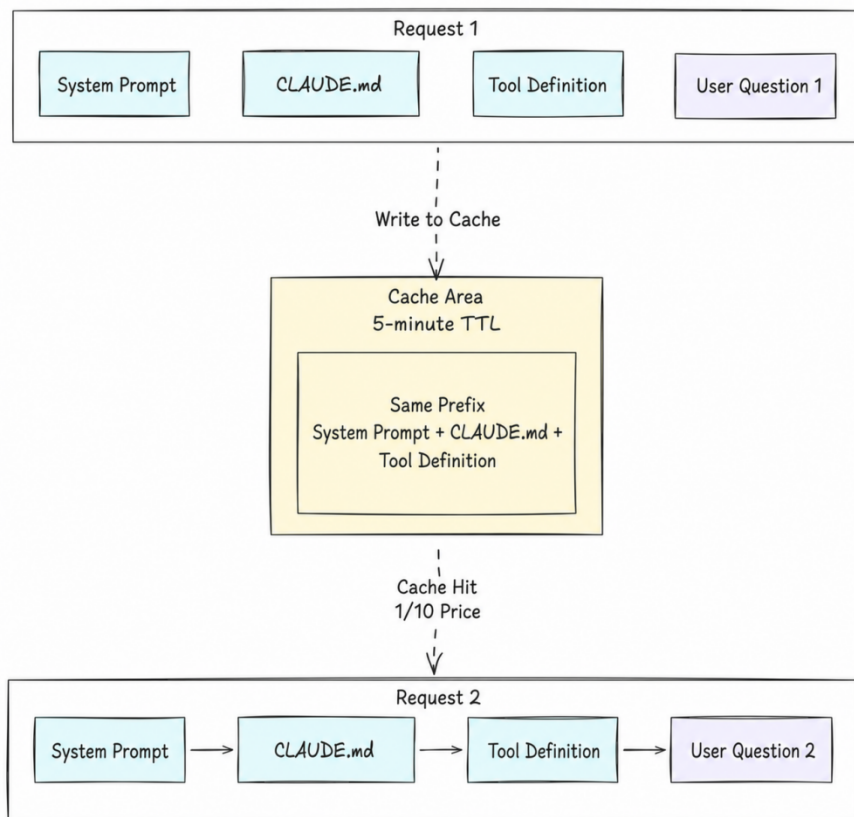


Figure 10.1: Prompt Caching Mechanism, Prefix Reuse

To maximize cache hit rates, follow these three key strategies:

- **Keep `CLAUDE.md` stable:** frequent changes to the configuration file will alter the prefix and invalidate the cache.

- **Aggregate tasks within a session:** try to complete related tasks within the same session, and avoid starting a new session for every small task.
- **Continue existing sessions:** make good use of the `--resume` or `--continue` parameters to extend existing sessions; do not start a new session from scratch every time.

Cost Monitoring

The JSON output in Headless mode provides precise cost data, making automated collection easy. See the following code example.

```
result=$(claude -p "review this PR" --output-format json)
cost=$(echo "$result" | jq -r '.total_cost_usd')
echo "PR_REVIEW_COST: $cost" >> /var/log/claude-costs.log
```

In interactive mode, simply enter the `/cost` command to view detailed cost statistics for the current session. See the following code example.

```
Total cost:          $0.55

Total duration (API): 6m 19.7s
Total duration (wall): 6h 33m 10.2s
```

By aggregating this data on the team's monitoring dashboard, managers can clearly see the cost distribution for each project, each engineer, and each task type, enabling refined resource management and budget control.

Note

Kage's notes

The ultimate goal of cost control is efficiency, not simply saving money. If a task can be completed in just 5 minutes using Opus, but takes 30 minutes of repeated debugging with Haiku, then the extra \$0.5 spent on Opus is absolutely worth it. Remember this formula: *real cost = API cost + human time cost*. Because an engineer's time dwarfs the API spend, the more expensive model is usually the cheaper choice overall. If your hourly rate is \$50, the total cost for the Haiku solution is \$25.03, while the Opus solution only costs \$5.50 (human time: 5 min × \$50/h = \$4.17, plus ~\$1.33 API; versus Haiku: 30 min × \$50/h = \$25.00 plus ~\$0.03 API). The conclusion is obvious: choose Opus.

Debugging: Opening the Black Box

"Why did Claude do that?" is one of the most frustrating questions when using AI tools. When Claude produces unexpected results (such as repeatedly reading the same file, executing unprompted operations, or consuming large amounts of tokens in a loop with no output), traditional debugging instincts, like setting breakpoints, checking call stacks, or inspecting variable values, are often powerless.

However, this "black box" is not completely opaque. Claude Code provides multi-level debugging tools to help you gain insight into its decision-making logic.

—debug Mode: An "X-ray" of the Execution Process

--debug is the most essential debugging tool in Claude Code. It's like taking an "X-ray" of Claude's execution process, outputting extremely detailed runtime information, including the full context of each conversation turn, the specific input parameters and return results for every tool invocation, real-time token consumption statistics, and the timing of API requests and responses.

You can enable debug mode with the following command.

```
claude --debug -p "List the files in the current directory"
```

Once --debug mode is enabled, you'll see structured logs like the following, clearly reconstructing Claude's thought and action trajectory.

```
[DEBUG] Starting session: abc123

[DEBUG] Model: claude-sonnet-4-6
[DEBUG] Turn 1

[DEBUG] User message: "List the files in the current directory"
[DEBUG] Tokens: input=150, output=0
[DEBUG] Tool call: Glob

[DEBUG]   pattern: "*"
[DEBUG]   path: "."
[DEBUG] Tool result: Glob

[DEBUG]   files: ["file1.js", "file2.py", ...]
[DEBUG]   duration: 45ms
[DEBUG] Turn 1 complete
```

```
[DEBUG] Total tokens: 500
[DEBUG] Estimated cost: $0.0015
```

--debug mode does not just provide full logs; it also supports flexible category filtering. If you only want to display logs related to API requests and Hook execution, use `claude --debug "api, hooks"`; if you want to hide statistics signals and file operation logs to focus on core logic, use `claude --debug "!statsig, !file"`.

When you encounter anomalies in an interactive session, there's no need to restart or rerun the command. Simply press the `Ctrl + O` shortcut to toggle verbose mode at any time. Once enabled, you can view the progress of Hook execution, detailed parameters and return information for tool calls, and the model's current thought process in real time during the conversation.

stream-json: Real-Time Insight into the "Thought Stream"

In Headless mode, `--output-format stream-json` is key to converting Claude's decision process into machine-readable data. It outputs each step's thought process and tool invocation details in real-time as **NDJSON (newline-delimited JSON)**. The core usage is as follows.

```
claude -p "Analyze the architecture issues in the src/ directory" --output-format
stream-json | jq '.'
```

With the `--include-partial-messages` parameter, you can obtain even more granular streaming data, including text fragments that are still being generated. This method is especially useful when you encounter questions like "Why did Claude suddenly stop at a certain point?" or "Is it stuck in an infinite loop generating the same content?"

PostToolUse Hook: Building Persistent Audit Logs

If you need to generate persistent and traceable audit records, rather than relying on one-off debug output, the `PostToolUse` Hook is the better choice.

1. Create the audit script:

Create a new file at `.claude/hooks/audit-log.sh` and enter the following content.

```
#!/bin/bash
# .claude/hooks/audit-log.sh
INPUT=$(cat)
TOOL=$(echo "$INPUT" | jq -r '.tool_name')
TIMESTAMP=$(date -u +"%Y-%m-%dT%H:%M:%SZ")
```

```
echo "$TIMESTAMP | $TOOL | $(echo "$INPUT" | jq -c '.tool_input')"  
>> .claude/audit.log  
exit 0
```

2. **Configure settings.json:**

Add the following configuration to `.claude/settings.json` to enable this Hook.

```
{  
  "hooks": {  
    "PostToolUse": [{  
      "matcher": "*",  
      "hooks": [{"type": "command", "command": "bash  
                  .claude/hooks/audit-log.sh"}]  
    }]  
  }  
}
```

Once configuration is complete, the Hook will automatically log every tool invoked by Claude along with its parameters. During post-mortem analysis, you can use the log to reconstruct Claude's sequence of actions.

Session Archiving and Breakpoint Debugging

Claude Code stores all session records in `~/ .claude/projects/{project}/ {sessionId}.jsonl`. If sub-agents are involved, their independent records are saved in `subagents/agent-{agentId}.jsonl`. These files are not just simple chat histories; they are authoritative data sources containing the complete conversation context, tool invocation details, and execution results, providing a solid foundation for in-depth post-event analysis.

The `--resume` and `--continue` parameters are not just for the convenience of resuming work, they are also powerful debugging tools. When an automated process encounters an error, you can use `claude --resume <session-id>` to precisely "rewind time" to the moment the issue occurred. This allows you to examine the exact information Claude received at that point, and to review the reasoning and tool invocation logic Claude used based on the information available at the time.

Diagnosing Abnormal Claude Behavior

The vast majority of abnormal Claude behaviors can be traced to one of three causes. It is recommended to investigate in this order, which usually leads quickly to the root of the problem.

- **Category One: Missing Context**
 - **Root Cause:** Claude is unable to access the key information required to complete the task.
 - **Typical Symptoms:** Repeatedly reading the same file, providing vague answers, or clearly lacking contextual basis.
 - **Diagnostic Method:** Use `--debug` to check the actual file contents and context range received by Claude.
 - **Solution:** Add clear project navigation guidance in `CLAUDE.md`, or directly provide the necessary context information and code snippets in the prompt.
- **Category Two: Conflicting Instructions**
 - **Root Cause:** There are two or more contradictory rules in `CLAUDE.md`.
 - **Typical Symptoms:** Behavior oscillates between two modes, or some established rules are directly ignored.
 - **Diagnostic Method:** Conduct a comprehensive review of all rules in `CLAUDE.md` to identify points of logical conflict.
 - **Solution:** Clarify rule priorities. Use explicit declarations to resolve ambiguities, such as, "When rule X conflicts with rule Y, rule X takes precedence."
- **Category Three: Insufficient Tool Permissions**
 - **Root Cause:** Claude attempts to perform an operation but lacks the necessary tool support or authorization.
 - **Typical Symptoms:** Repeatedly fails at a particular operation, or is forced to use an inappropriate substitute tool to complete the task.
 - **Diagnostic Method:** Check the `--allowedTools` configuration and review the Hook's interception logs to confirm whether any requests were blocked.
 - **Solution:** Grant the necessary tool permissions, or adjust the task description to guide Claude to use currently available alternatives.

Note

Kage's notes

The fundamental difference between debugging Claude and debugging traditional code is this: Code execution is deterministic, the same input will always produce the same output. Claude's behavior, however, is probabilistic; the same prompt may elicit completely different responses in different contexts. Therefore, you cannot rely on the traditional "reproduce the bug" approach to solve problems. Instead, you must rely on log analysis, audit trail review, and CLAUDE.md rule inspection to infer the root cause of issues.

Security Guidelines: Least Privilege Is the Bottom Line

"Claude Code is powerful, but it's also capable of executing high-risk operations," Xiaobing asked. "How can we effectively constrain it and prevent unauthorized actions?"

There is no one-size-fits-all, permanent solution to this. Only rigorous engineering discipline can help. Security is not a simple switch, but a layered, in-depth defense system.

Permission Modes: The First Line of Defense

Claude Code provides five permission modes (see Table 10.3).

| Mode | Behavior characteristics | Applicable scenarios | Restricti on level |
|-------------|--|-------------------------------------|--------------------|
| plan | Read-only mode, prohibits file modification or command execution | Code review, architecture analysis | Very strict |
| default | Requests authorization when using each type of tool for the first time | Daily development (default setting) | Strict |
| acceptEdits | Automatically accepts file editing requests, other operations still require confirmation | Focus on coding phase | Relativel y strict |

| Mode | Behavior characteristics | Applicable scenarios | Restriction level |
|-------------------|---------------------------------------|---|-------------------|
| dontAsk | Only executes pre-approved operations | Restricted environment | Relaxed |
| bypassPermissions | Skips all permission prompts | Container or virtual machine environment only | Very relaxed |

Table 10.3: Five Permission Modes and Applicable Scenarios

In any automation scenario (such as CI/CD pipelines, scheduled tasks, or headless mode invocations), you must explicitly specify the permission mode.

If you use `bypassPermissions` mode in a CI/CD pipeline, you must ensure the runtime environment is a fully isolated container or virtual machine. Never enable this mode on a developer's local machine, to avoid uncontrollable security risks.

Tool Permissions: Fine-Grained Control

The `--allowedTools` and `--disallowedTools` parameters support pattern matching, enabling fine-grained control at the tool level.

Here are some command line examples.

```
# Scenario 1: Code review in a CI/CD pipeline (read-only permissions required)
claude -p "Review PR changes" \
--allowedTools "Read" "Grep" "Glob" \
--max-turns 5

# Scenario 2: Allow specific Bash command patterns
claude -p "Run tests and analyze results" \
--allowedTools "Read" "Grep" "Bash(npm run test *)" "Bash(npm run lint)"
```

In the team-shared `.claude/settings.json` file, you can use three levels of rules, deny, allow, and ask, for precise control. See the following code example.

```
{
  "permissions": {
    "allow": [
      "Bash(npm run lint)",
      "Bash(npm run test *)",

```

```
    "Bash(git commit *)"
  ],
  "deny": [
    "Bash(curl *)",
    "Bash(wget *)",
    "Read(./env)",
    "Read(./env.*)",
    "Read(./secrets/**)",
    "Bash(git push --force *)"
  ],
  "ask": [
    "Bash(git push *)"
  ]
}
```

The evaluation order for permission rules is **deny** → ask → **allow**. Among these, deny rules have the highest priority. This means that even if an operation matches an allow rule, as long as it also matches a deny rule, the operation will be directly rejected. This mechanism embodies the core design philosophy of "security first."

.claudeignore and File Access Control

The .claudeignore file can be used to prevent Claude from indexing specific files, but it is important to note its known limitations, it may be bypassed in certain scenarios. Therefore, a more reliable security strategy is to explicitly prohibit read operations on sensitive files in the permissions.deny configuration.

A recommended configuration example is as follows.

```
{
  "permissions": {
    "deny": [
      "Read(./env)",
      "Read(./env.*)",
      "Read(./secrets/**)",
      "Read(**/*.pem)",
      "Read(**/*.key)"
    ]
  }
}
```

It is recommended to use both mechanisms together to build dual-layer protection: `.claudeignore` serves as the first layer of optimization, reducing unnecessary file indexing overhead; `permissions.deny` acts as the second, rigid line of defense, ensuring that even if indexing is bypassed, sensitive files cannot be read.

Hook Security Scanning

In *Chapter 5*, we explored the event-driven mechanism of Hooks in depth. The `PreToolUse` Hook can intercept potentially dangerous operations before a tool is actually executed, adding a dynamic security barrier to the system.

Below is an example of a shell script that parses incoming tool invocation requests and blocks specific dangerous patterns.

```
#!/bin/bash
# .claude/hooks/security-check.sh
INPUT=$(cat)
TOOL=$(echo "$INPUT" | jq -r '.tool_name')
TOOL_INPUT=$(echo "$INPUT" | jq -r '.tool_input')

if [ "$TOOL" = "Bash" ]; then
    COMMAND=$(echo "$TOOL_INPUT" | jq -r '.command')
    # Regex match for high-risk command patterns
    if echo "$COMMAND" | grep -qE 'rm\s+-rf\s+|/
        sudo\s+rm|chmod\s+777|curl.*\|s*bash'; then
        echo '{"decision": "block", "reason": "Dangerous command
            blocked by security policy"}' >&2
        exit 2
    fi
fi

exit 0
```

In addition to rule-based scripts, Claude Code also supports **Prompt-type Hooks** (configured with type: "prompt"), which evaluate the current tool invocation's context and intent by calling a large language model. This approach can identify complex security threats or logical vulnerabilities that simple regex matching cannot cover.

Trust Assessment of MCP Servers

Each MCP server acts as a proxy for Claude's interactions with external systems. If an MCP server harbors malicious intent or security vulnerabilities, attackers could exploit it to escalate

Claude's tool invocation capabilities into unauthorized access to internal network resources. Therefore, when introducing an MCP server, a strict review checklist must be enforced, including source trustworthiness (confirming whether it comes from an official or reputable developer), code transparency (ensuring the code is open source and auditable), and reasonable permissions (assessing whether the requested permission scope adheres to the principle of least privilege).

A frequently overlooked risk is that once an MCP server is authorized, the system will not seek user consent again for subsequent updates. This means that an MCP server that was initially safe could have malicious code injected during an automatic update.

Version pinning is a necessary defense. You can enforce the use of a specific, verified version to prevent automatic upgrades to untrusted versions.

In automated environments such as CI/CD, always pass `--strict-mcp-config`. It makes the run load only the MCP configuration you name on the command line and ignore every other source, which removes the chance of a developer's local server being picked up by a pipeline. Keep that configuration in a checked-in file — a project `.mcp.json` referenced explicitly from the CI job — rather than assembling flags inline, so the pipeline's tool surface is reviewable in the same pull request as the code.

API Key Lifecycle Management

As Claude Code becomes more deeply integrated into development workflows, references to `ANTHROPIC_API_KEY` in projects are increasing. To ensure security, a strict API key lifecycle management mechanism must be established. The core management principles are as follows.

- **Isolation:** Assign separate API keys for different projects and environments (such as development, testing, and production) to avoid "one key for all."
- **Regular rotation:** Establish a schedule to regularly update keys, reducing the risk of long-term leaks.
- **Emergency response:** Set up a rapid response mechanism so that if a key is suspected or confirmed to be leaked, it can be immediately revoked and regenerated.

No single measure is sufficient to ensure security; it is recommended to implement the following four layers of protection simultaneously.

- **Storage isolation:** Strictly store keys in `.env` files.
- **Version control exclusion:** Explicitly exclude `.env` files in `.gitignore`.
- **Runtime blocking:** Configure permissions.deny rules in `.claude/settings.json` to explicitly prohibit Claude from reading `.env` and related files.
- **Pre-commit interception:** Deploy a `PreToolUse` Hook to scan before code is committed or relevant commands are executed, detecting and blocking any attempt to accidentally commit keys.

Large Codebases: Navigating Millions of Lines of Code Within a Limited Context Window

The larger the codebase, the greater the challenge for Claude Code. This is not due to Claude's lack of capability, but rather the limitation of a finite context window.

Although Claude's standard context window is 200,000 tokens, this does not mean you can fill it with an equivalent amount of code. After subtracting the System Prompt (about 5,000 tokens), the `CLAUDE.md` configuration file (about 2,000 tokens), conversation history (about 20,000 tokens), tool definitions and returned results (about 10,000 tokens), and necessary safety buffer (about 20,000 tokens), the actual space available for code is only 143,000 tokens, which translates to 35,000 to 40,000 lines of code. However, a medium-sized e-commerce backend system can easily exceed 50,000 lines, and enterprise-level SaaS projects often surpass 200,000 lines. The conclusion is clear: **for large projects, it is impossible to load the entire codebase into context at once; a chunking strategy must be adopted.**

Hierarchical `CLAUDE.md`

In a Monorepo architecture, a hierarchical layout of `CLAUDE.md` is the core infrastructure for managing large codebases. Claude Code follows a strict priority order when loading configuration files (see *Figure 10.2*).

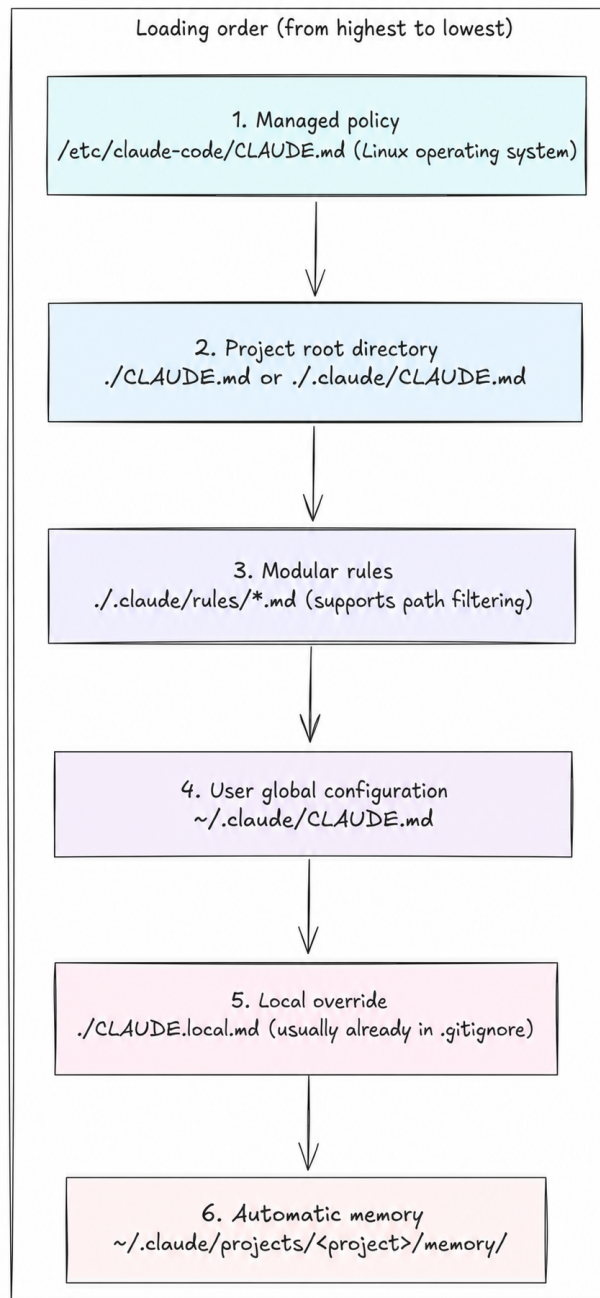


Figure 10.2: The 6-level hierarchical loading order of CLAUDE.md

CLAUDE.md files in subdirectories are only triggered and loaded when Claude processes files within those directories ("on-demand loading"). In contrast, CLAUDE.md files in parent directories are recursively searched upward from the current working directory and loaded cumulatively at each level.

This mechanism makes configuration management in Monorepos highly flexible: the root CLAUDE.md can define the overall architecture and cross-module conventions, while CLAUDE.md files in submodule directories focus on internal module details. The two are automatically merged at runtime, ensuring both global consistency and local specificity.

The .claude/rules/ directory supports modular rule configuration based on path filtering. A typical example is as follows.

```
---
paths:
- "src/api/**/*.ts"
---
# API Development Standards
- All API endpoints must include input validation logic.
- Error responses must uniformly use the ErrorResponse type.
- The authentication middleware is configured globally and does not need to be
  imported repeatedly at each endpoint.
```

This mechanism enables "on-demand activation" of rules: when Claude modifies API-related files, the system automatically loads the corresponding API standards; when working on frontend components, it automatically switches to frontend development standards.

Guide Searches, Not Exhaustive Reading

The core strategy for managing large codebases is to "guide Claude to search, rather than blindly read." Without clear direction, Claude may attempt to read the entirety of all related files, which is not only inefficient but also significantly increases costs. Therefore, a clear navigation strategy must be established in CLAUDE.md. For example, consider the following codebase navigation principles.

Codebase Navigation Principles

Before handling tasks involving multiple files, strictly follow these steps.

1. Use Grep to search for key function names, class names, or error codes to precisely identify the relevant files.
2. Use Glob to list the file structure of the target directory for a quick overview of the overall organization.

3. Only read the specific contents of the 2 to 3 most relevant core files.
4. It is strictly forbidden to perform large-scale file reading without a clear objective.

For tasks that may generate a large amount of output (such as log analysis or large file searches), delegate them to sub-agents.

Claude Code comes with a dedicated **Explore** sub-agent designed for codebase exploration. This sub-agent uses the lowest-cost Haiku model, has read-only tool permissions, and offers three exploration modes: quick, medium, and very thorough. All search and reading operations are completed within the sub-agent's independent context, and the main conversation only receives the final summary. This mechanism not only effectively preserves valuable context space in the main conversation but also significantly reduces operational costs.

Sub-Agents: Building Modular Context Boundaries

As discussed in *Chapter 3*, the context isolation mechanism of sub-agents plays an even more critical role in large codebase management: breaking down massive tasks by module, assigning them to different sub-agents, and ensuring each sub-agent only holds the context of its responsible module.

Developers can customize sub-agents in the `.claude/agents/` directory, specifying dedicated toolsets, model versions, and even persistent memory for each.

Below is a sample configuration for a reviewer dedicated to the payment module.

```
---
name: payment-reviewer
description: Review code changes in the payment module
tools: Read, Grep, Glob
model: sonnet
memory: project
---

You are a dedicated reviewer for the payment module. The review standards are as follows.

1. All monetary calculations must use the
   `Decimal`
   type; the use of floating-point numbers is strictly prohibited.
2. Any changes to payment status must be completed atomically within a database transaction.
3. When calling external payment gateways, a retry mechanism and timeout control must be implemented.
```

Context Compression and Fine-Grained Management

As the number of conversation turns increases, the context window gradually approaches its limit. Claude Code features an automatic compression mechanism (typically triggered when capacity reaches about 95%), but for more precise control over information retention, developers can manually trigger compression using the `/compact` command, along with instructions to explicitly preserve key information.

```
/compact preserves the full paths of all modified files and the test commands  
Compression rules can also be preset in CLAUDE.md to ensure that sub-agents always  
retain core state during the compression process.
```

Compression Strategy

When performing context compression, the following content must always be retained.

- A complete list of full paths for all modified files.
- Failed test cases and their specific error stack information.
- Remaining steps for the current unfinished task.

By setting the environment variable `CLAUDE_AUTOCOMPACT_PCT_OVERRIDE`, you can customize when automatic compression is triggered. By default, it triggers at about 95% capacity. Setting it to 50 will cause compression to occur earlier, helping to avoid performance degradation or response delays caused by processing large amounts of data when the context is nearly exhausted.

Prompt Engineering: Writing Effective CLAUDE.md and Prompts

"Why is it that, for the same task, some people get a perfect answer from Claude on the first try, while I have to go back and forth several times?" Xiaobing asked.

The answer usually doesn't lie in differences in Claude's capabilities, but in the quality of the instructions. Excellent instructions are like a perfect requirements document: clear, specific, and unambiguous.

Principles for Writing CLAUDE.md

`CLAUDE.md` is the most critical configuration file in Claude Code. Empirical data shows that optimizing this file can bring a measurable improvement in general coding performance—on the order of several percent in our own testing, and larger still in some repositories.

It is recommended to include the following content.

- **Key commands:** Specific instructions for building, testing, and deploying (Claude cannot infer this information on its own).
- **Custom standards:** Code style rules that differ from the defaults.
- **Architectural constraints:** Project-specific architectural decisions and design limitations.
- **Environment requirements:** Special configurations for the development environment (such as required environment variables, non-standard toolchains, etc.).
- **Pitfall guides:** Common traps and subtle system behaviors.

The following content should be avoided.

- **Inferable information:** Content that Claude can deduce by reading the code.
- **General conventions:** Standard language specifications (Claude already has this knowledge built-in).
- **Volatile information:** Frequently changing content (which can cause cache invalidation).
- **File descriptions:** Code descriptions of files (should be handled by search tools).
- **Vague advice:** Principles that are self-evident, such as "write clean code."

A counterintuitive principle is: less is more. `CLAUDE.md` follows the rule "the more concise, the more effective." Lengthy configuration files not only increase the cost of each API call (since the entire file must be loaded), but also risk burying key rules in information noise. Detailed domain knowledge should be migrated to the Skills module for on-demand loading. This architectural design can significantly improve efficiency and relevance.

Techniques for Precise Expression

The core of an instruction is to **convey the most complete information with the least token consumption**. Here are three practical tips.

Quantify everything

Avoid vague descriptions; always use specific data. For example, "handle large amounts of data" can be written as "handle 1 million records," "response should be fast" as "P99 response time <200 ms," and "code should be concise" as "functions no longer than 20 lines, no more than 4 parameters."

Example-driven

For complex requirements, one example is worth ten descriptions.

```
Requirement: Convert camelCase to snake_case
Example:
Input: "getUserName" → Output: "get_user_name"
Input: "HTTPRequest" → Output: "http_request"
```

Define boundaries clearly

Defining "prohibited items" is as important as defining "required actions" to prevent Claude from overstepping.

Scope limitation: Only modify the auth module; do not touch any other files.

Tool constraints: Must use the project's existing logger; using `console.log` is prohibited.

Dependency management: Adding any new dependencies is strictly forbidden.

Failure handling: If you encounter an unsolvable problem, report it immediately and stop; do not keep retrying.

Stepwise Confirmation

For complex tasks, stepwise execution is far superior to submitting everything at once. By breaking down large requirements into independent verification loops, you can significantly reduce the risk of deviation.

The following shows an example execution flow.

```
Step one: "Please analyze the root cause of this bug, but do not modify the code yet."
(After confirming the analysis logic is correct, proceed to step two)

Step two: "Based on the above analysis, fix the bug by modifying only the necessary code."
(After confirming the fix is correct, proceed to step three)

Step three: "Write corresponding test cases for this fix."
```

The advantage of this approach is that each step can be independently verified, preventing Claude from over-executing in the wrong direction. In Headless mode, it is recommended to use the `--max-turns` parameter to limit the number of turns per step, eliminating the risk of runaway execution.

Team Adoption: The Leap from Individual to Organization

At the individual level, the adoption threshold for Claude Code is low. Once engineers experience the efficiency gains firsthand, they will continue to use it on their own initiative. But at the team level, spontaneous formation cannot be relied upon. Unifying team usage standards, configuration specifications, and workflows requires proactive engineering design and institutionalized management.

Shared Configuration: The Cornerstone of Team Unity

The `.claude/` directory is the core infrastructure for team collaboration. To ensure unified team standards while retaining individual flexibility, it is essential to clearly distinguish files included in Git version control (shared) from those kept locally (personal). *Table 10.4* lists the sharing strategy for files in the `.claude/` directory.

| File path | Shared (Git) | Purpose description |
|--|--------------|--|
| <code>.claude/settings.json</code> | Yes | Team shared configuration: defines team-wide model parameters, tool permissions, etc. |
| <code>.claude/settings.local.json</code> | No | Personal override settings: allows members to override some configurations based on personal habits (such as preferred model) |
| <code>.claude/agents/*.md</code> | Yes | Team sub-agents: defines team-shared specialized role agents (such as 'Code Reviewer', 'Testing Expert') |
| <code>.claude/skills/*.md</code> | Yes | Team skill library: stores project-specific domain knowledge and complex operation guides for on-demand loading |
| <code>.claude/rules/*.md</code> | Yes | Modular rules: breaks down large rules into modules for easier maintenance and combination |
| <code>CLAUDE.md</code> | Yes | Project memory: core context, architecture overview, and key constraints of the project |

| File path | Shared (Git) | Purpose description |
|-----------------|--------------|--|
| CLAUDE.local.md | No | Personal preferences: member's personal prompt preferences or temporary notes |

Table 10.4: .claude/ Directory File Sharing Strategy

Claude Code will automatically add rules to the project's root `.gitignore` file to exclude `.claude/settings.local.json` and `CLAUDE.local.md`.

Governance Standards for CLAUDE.md

`CLAUDE.md` should be treated as core code and included in a rigorous code review process. Any changes to `CLAUDE.md` must be submitted via PR and approved by at least one senior engineer before merging.

The reason is simple: every line in `CLAUDE.md` directly determines the behavior patterns of all team members when using Claude. As the team's highest-level "AI behavior standard," its importance and impact are no less than that of core business code, and it must therefore be subject to the same level of scrutiny.

By using the **ConfigChange Hook** provided by Claude Code, you can monitor changes to configuration files (such as settings files) during a session, enabling automatic auditing or even blocking unauthorized modifications.

The following configuration example shows how to log configuration changes.

```
{
  "hooks": {
    "ConfigChange": [{
      "matcher": "*",
      "hooks": [{
        "type": "command",
        "command": "echo '[AUDIT] Config changed: $CONFIG_FILE' >>
          ~/.claude/config-audit.log"
      }]
    }]
  }
}
```

Organization-Level Policy Management

For organization-wide control, managed settings provide policies that a developer cannot override. They live in a managed-settings file that IT deploys to the machine, and they are the right place for the non-negotiables: the permission mode, a denied-tools list, and whether MCP servers may be added at all. Anything a team should be able to decide for itself belongs in the project configuration instead — managed settings win every conflict, so putting preferences there makes them un-overridable by design.

Below is a sample configuration file for deploying global policies on a Linux operating system.

```
// /etc/claude-code/managed-settings.json (Linux)
{
  "disableBypassPermissionsMode": "disable",
  "allowManagedPermissionRulesOnly": true,
  "allowManagedHooksOnly": true,
  "permissions": {
    "deny": [
      "Bash(curl *)",
      "Bash(wget *)",
      "Read(./env)",
      "Read(./env.*)"
    ]
  }
}
```

Managed settings take precedence over user local configurations, project-level configurations, and even command-line parameters. When `disableBypassPermissionsMode` is set to `disable`, users are completely prohibited from using the `--dangerously-skip-permissions` flag locally, fundamentally eliminating the risk of permission bypass. When `allowManagedPermissionRulesOnly` is set to `true`, the system will only enforce permission rules defined by the organization, ignoring rule definitions from other sources. The above configuration forms a solid "defensive line" for the security team.

Progressive Capability Building Roadmap

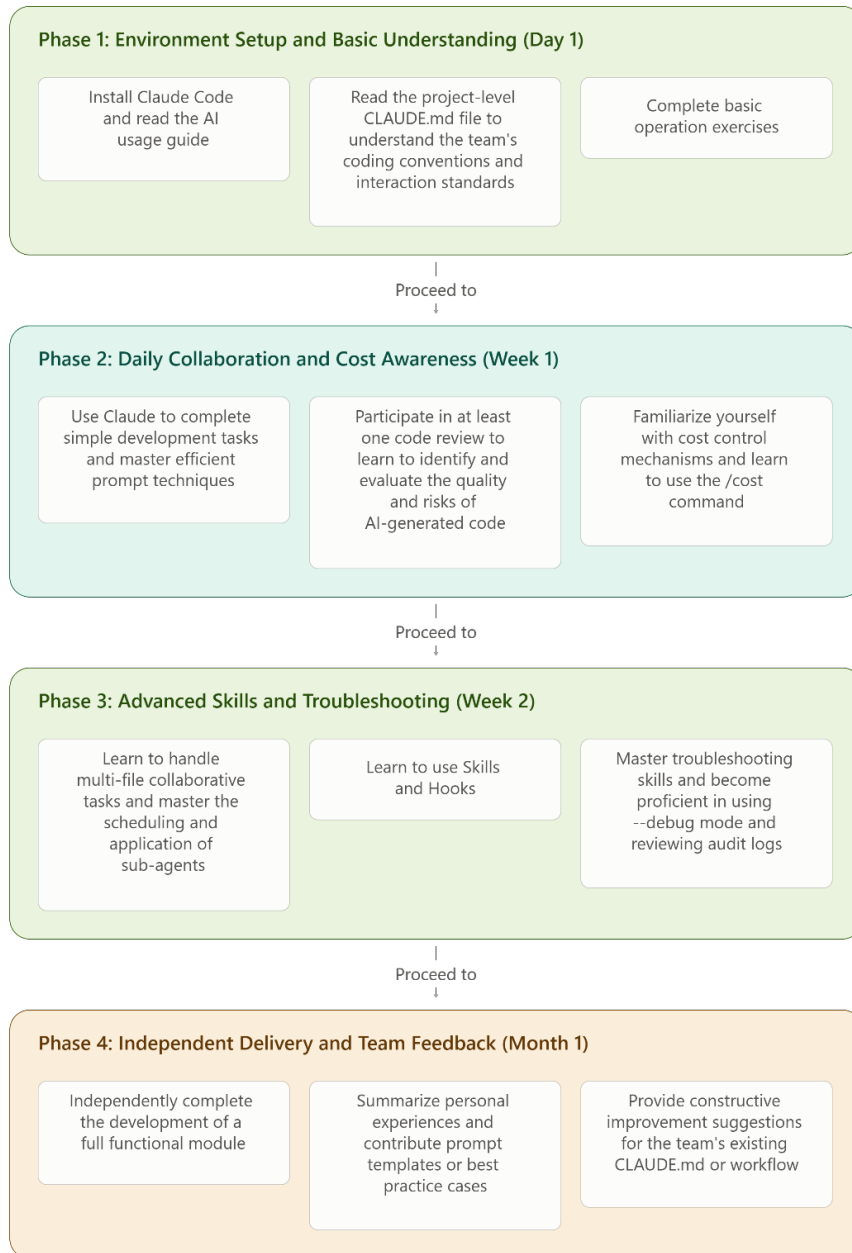
Rolling Claude Code out to everyone at once tends to fail for a reason that has nothing to do with the tool: nobody has yet learned what good usage looks like in your codebase, so there is

no one to ask. Going one team at a time produces that answer before the next team starts. Teams generally pass through four stages of maturity.

- **Stage One: Individual Exploration** (First Two Weeks After Launch): Engineers independently explore and identify high-value application scenarios, but have not yet established systematic norms. The strategic goal is to accumulate empirical cases. This stage encourages free experimentation to gather authentic user feedback and successful paradigms, avoiding premature standardization that could stifle innovative thinking.
- **Stage Two: Project-Level Standardization** (First Month After Launch): The team standardizes best practices that have been validated (including `CLAUDE.md` rules, commonly used Skills, and basic Hooks), consolidating them into the project's `.claude/` directory and bringing them under version control (Git). The strategic goal is asset communalization, transforming scattered "private experience" into "shared team assets."
- **Stage Three: Automated Integration** (Second Month After Launch): By leveraging Hooks and Headless mode, quality gates such as code review, test execution, and format checks are deeply integrated into the CI/CD pipeline. The strategic goal is process automation, driving Claude Code's evolution from a "human-assistive tool" to an "automated component of the development workflow."
- **Stage Four: Organization-Level Scaling** (Third Month After Launch): Through unified Plugin distribution, deployment of organization-level MCP servers, and implementation of managed policies, mature capabilities are promoted across multiple teams and even the entire engineering organization. The strategic goal is global scaling, breaking down team silos and achieving group-level unification of technology stacks, security policies, and collaboration standards.

Claude Code Onboarding Training Program

Once the first team is running, the next constraint is onboarding everyone else. *Figure 10.3* lays out a structured programme for a new team member — someone joining a team that already uses Claude Code — taking them from no experience to contributing to and improving the team's workflow within a month, in four progressive stages.

*Figure 10.3: Claude Code Onboarding Training Program*

Integration of the SDD Ecosystem and Workflow

After delving into the independent mechanisms introduced earlier, a core question emerges: **how can these disparate components be linked into a complete workflow?**

Currently, the open-source community is experiencing explosive growth at the workflow level. A large number of Claude Code-based plugins and frameworks have emerged on GitHub, all attempting to answer the same question: What should software engineering methodology look like in the AI era? Despite different approaches, the answers converge on **SDD (Specification-Driven Development)**: define the specification first, then write the code.

Four-Layer SDD Ecosystem Architecture

These tools are naturally divided into four layers, each dedicated to solving a specific core problem.

- **Layer One: Atomic Skills:** A typical example is Matt Pocock's open-source grill-me. This project consists of only six lines of Markdown code, yet it transforms Claude from a "compliant executor" into a "relentless interviewer," forcing users to scrutinize every unresolved issue and implicit assumption. This is the ultimate demonstration of the power of Skills described in *Chapter 3*: using declarative Markdown to reshape AI behavior patterns.
- **Layer Two: Context Persistence:** This layer aims to solve the problem of context loss in long-running tasks. Planning with Files adopts the simplest strategy: writing plans to files (such as `task_plan.md`, `findings.md`, `progress.md`). When the conversation context is cleared, Claude can restore progress by rereading these files. Trellis goes a step further, using the `.trellis/` directory structure to achieve cross-platform standardization, making the same specification usable across more than ten platforms including Claude Code, Cursor, and OpenCode.
- **Layer Three: Specification Generation Frameworks:** This layer provides a complete pipeline from specification definition to code generation. Spec-Kit (an official GitHub project with over 70k stars) defines the entire process from "constitution" to code, with nine commands covering the full development lifecycle, though it may be too heavy for small projects. In contrast, OpenSpec (by Fission AI) is more lightweight, requiring only four core commands for a one-step process; `cc-sdd` introduces **EARS (Easy Approach**

to **Requirements Syntax**) requirements syntax, achieving an ultra-simplified SDD workflow.

- **Layer Four: Complete Methodology:** This layer integrates a comprehensive engineering practice system. Superpowers (with over 40k stars at the time of writing, featured in the Anthropic official plugin marketplace) packages 14 composable Skills and defines seven standard stages: brainstorming → creating Git Worktree → planning → executing the plan → **Test-Driven Development (TDD)** → code review → final merge. It enforces TDD principles (if the user writes code before tests, the system will automatically delete the code) and uses sub-agents to distribute tasks in parallel, this is a practical application of the sub-agent mechanism described in *Chapter 4*. In addition, Compound Engineering focuses on the "learning" dimension often overlooked by other frameworks: after each iteration, it deposits experience into the docs/solutions/ directory via the /ce:compound command; when a similar problem arises next time, the Agent will prioritize retrieving historical solutions, enabling continuous evolution.

Recommended Workflow Combination Solutions

Table 10.5 lists three recommended combinations of workflow tools.

| Applicable Scenarios | Recommended Plugin Combinations | Core Workflow |
|--------------------------|---|--|
| Individual Developer | grill-me
+OpenSpec+Planning with Files | Requirement interrogation → Specification generation → Progress persistence → Implementation verification

(Use grill-me to deeply explore requirements, OpenSpec for rapid specification generation, and file recording to ensure long tasks do not lose context) |
| Small Team Collaboration | Superpowers+Trellis | Standardization → Full-process takeover

(Trellis ensures cross-platform specification consistency, Superpowers enforces standardized development processes) |

| Applicable Scenarios | Recommended Plugin Combinations | Core Workflow |
|-----------------------------|--|---|
| Long-term Evolving Projects | OpenSpec+Compound Engineering+grill-me | Specification enforcement + Knowledge compounding

(Driven by strict standards, Compound Engineering accumulates historical solutions to enable continuous team capability iteration) |

Table 10.5: Recommended Workflow Tool Combinations

Not recommended: Superpowers + Spec-Kit. Both are heavyweight frameworks with conflicting stage definitions; mixing them leads to process confusion. Installing all tools in full does not improve efficiency and instead increases cognitive load, which runs counter to the original intent of introducing tools to simplify work.

Note

Kage's notes

The core of SDD does not lie in which plugins you use, but in establishing the habit of "thinking before acting" in development. If, before launching **Claude Code**, you spend just five minutes clarifying the "scope of objectives (what to do, what not to do)" and "validation criteria," you will already outperform 90% of intuition-driven programming (such as Vibe Coding), even without installing any plugins. The essence of plugins is simply to automate this critical habit.

Global View: The Orchestration of Nine Core Mechanisms

At this point, we have fully analyzed the nine core mechanisms of Claude Code engineering. To grasp the overall architecture from a macro perspective, we have consolidated these mechanisms in *Table 10.6* for a comprehensive overview.

| Mechanism | Core Value | Main Application Scenarios |
|-----------|--|--|
| CLAUDE.md | Project Memory and Context Persistence | Universal for All Scenarios
(recommended to always configure) |

| Mechanism | Core Value | Main Application Scenarios |
|---------------|---|--|
| Sub-agents | Parallel Execution and Context Isolation | Decomposition of Complex Multi-step Tasks |
| Skills | Domain Knowledge Reuse and Workflow Standardization | High-frequency Repetitive Professional Tasks |
| Commands | Standardized Operation Encapsulation | Team-agreed Fixed Operation Sequences |
| Hooks | Event-driven Safety Gatekeeping and Automation | Quality Assurance and Security Defense |
| MCP | External Data Source and Tool Integration | Scenarios Requiring Real-time Data or External API Calls |
| Headless Mode | CI/CD Integration and Automated Pipeline Construction | Non-interactive Automated Processes |
| Agent SDK | Custom Agent Application Development | Productized Development (not just tool usage) |
| Plugins | Capability Packaging and Cross-team Distribution | Community Sharing and Enterprise-level Promotion |

Table 10.6: Overview of Nine Core Mechanisms

No single mechanism is omnipotent, nor is any one redundant. `CLAUDE.md` forms the cornerstone of the entire system, defining Claude's "working context"; Skills and sub-agents are two dimensions for expanding capabilities within this context, the former deepens domain knowledge, the latter breaks through execution boundaries; **Hooks** serve as the quality and security defense line during execution; **MCP** acts as the bridge to the external world; **Headless mode** and Agent SDK elevate Claude Code from a mere interactive tool to a programmable capability component; **Plugins** ultimately package all these capabilities into knowledge assets that are transferable and reusable.

These nine mechanisms are combined as needed and work in concert to build a complete and efficient AI-assisted development engineering system.

"So, after learning all this, what can I actually do with Claude Code?" Xiaobing said.

Kage thought for a moment and replied, *"What you can do with Claude Code is essentially no different from three months ago. It has always been capable of reading code, writing code, reviewing code, generating documentation, and running tests. What you have truly learned is how to systematically integrate these disparate capabilities."*

This distinction is crucial. A tool user only invokes the tool when needed and discards it after use; an engineer, on the other hand, deeply integrates the tool into the workflow, enabling it to intervene automatically at the right moment, in the correct manner, and under predefined constraints. The latter not only requires proficiency with the tool itself, but also depends on a profound understanding of the process, meticulous design of constraint mechanisms, and anticipation of potential failure modes.

This is precisely the core essence of "engineering": **enabling powerful yet inherently uncertain tools to reliably create value within an orderly system.**

Although Claude Code's capabilities will continue to evolve, context windows will keep expanding, reasoning accuracy will steadily improve, and new tools and APIs will emerge in rapid succession, the engineering mindset remains steadfast: understand the underlying mechanisms, make decisions at the appropriate level, trade constraints for reliability, and transform individual best practices into shared team assets.

In conclusion, Xiaoxue summarized: *"From the initial architectural overview to the final engineering practice, the essence of this journey is the transition from 'knowing the tool' to 'mastering the tool'."*

Summary

As the grand finale of this book, this chapter is not a mere technical checklist, but a deep distillation of engineering methodology.

The core of **cost control** lies in managing three key dimensions: flexibly deploying Haiku, Sonnet, and Opus according to task complexity to achieve optimal performance-cost balance; effectively reducing token consumption through sub-agent context isolation, streamlining CLAUDE.md content, and optimizing caching strategies; and setting hard budget limits by using the `--max-budget-usd` parameter to establish an inviolable cost ceiling. In addition, special caution must be exercised with the Extended Thinking mode, which is often an overlooked source of hidden costs and should be enabled judiciously when necessary.

The primary principle of **debugging** is to make the execution process transparent. To this end, we have built a multidimensional observability system: use the `--debug` flag to obtain real-time logs; output machine-readable execution traces via `stream-json`; generate persistent audit records with the `PostToolUse` Hook; and retain session files as raw data to provide a

complete basis for in-depth post-analysis. For common abnormal behaviors, their root causes can generally be traced to three categories: missing context information, conflicting instruction logic, and insufficient execution permissions.

The core of **security architecture** is strict adherence to the principle of least privilege. We have constructed a layered defense system: five permission modes precisely match different levels of usage complexity; deny, allow, and ask rules enable atomic-level control over tool invocation; the PreToolUse Hook intercepts potentially dangerous behaviors before execution; **managed settings** enforce non-overridable global security policies; and for **MCP servers**, version locking and rigorous code review are mandatory.

The core strategy for **large codebases** is to replace "full reading" with "guided search." Hierarchical CLAUDE.md and .claude/rules/ directory structures enable on-demand rule loading and scenario-based adaptation; Explore sub-agents perform deep searches and file traversals in isolated contexts; and intelligent compression techniques distill key information, significantly extending effective working time.

The soul of **prompt engineering** lies in precision. A few guidelines carry most of the weight: keep CLAUDE.md under 200 lines, the figure the official guidance gives, and move domain expertise into Skills rather than letting the file grow; quantify prompts instead of describing them vaguely; give high-quality examples; state task boundaries explicitly; and confirm complex tasks step by step.

Team adoption requires steady, phased advancement: individual exploration → project-level unification → automated integration → organization-wide scaling. Throughout this process, sharing the .claude/ directory is the cornerstone of collaboration; governing CLAUDE.md through the PR process ensures compliance quality; and **managed settings** serve as the final safeguard for security.

From today onward, what you have mastered is not just a set of efficient tools, but an entirely new way of thinking.

Reflection Questions

1. Looking back, what was your biggest pain point when using Claude Code, runaway costs, inconsistent behavior, insufficient reliability, or other challenges? Based on the engineering practices introduced in this chapter, which mechanism do you plan to prioritize for improvement?

2. If you were to develop a "first month" engineering roadmap for a 10-person team new to Claude Code, which three initiatives would you prioritize? Why these three, and not others?
3. This chapter mentions "*true cost = API cost + time cost.*" Where is the balance point for this formula in your team? In what scenarios is it worth incurring higher model costs to save valuable developer time?

Afterword

The Way of the Blade

As I completed the manuscript and reviewed the table of contents, I discovered an interesting coincidence: the very structure of this book is itself a metaphor for Harness (see *Figure 1*).



Figure 1: The Harness Metaphor

Chapters 1 and 2 are **context injection**, loading "project background" for the reader, just as `CLAUDE.md` loads project memory for the model at the start of a session.

Chapters 3 through 6 are **tool registration**, Skills, sub-agents, Hooks, and MCP, four major extension components, handed to the reader one by one like four types of tools.

Chapters 7 through 9 are the **execution environment**, Headless, Agent SDK, Plugins, extending capabilities from the terminal to CI/CD, programming interfaces, and ecosystem distribution.

Chapter 10 is the **governance layer**, cost, debugging, security, scale, instructions, and collaboration, ensuring the entire system operates reliably in real engineering environments.

Context, tools, execution environment, governance, aren't these precisely the four components of a Harness?

This structure was not deliberately crafted, but emerged naturally. After all, the best way to understand a system is to organize knowledge according to the system's own structure.

During the writing period of this book (late 2025 to early 2026), I personally witnessed an irreversible paradigm shift in the field of AI-assisted development: the focus of competition has shifted completely from the model itself to the **Harness**.

In 2025, the industry's attention was still on "which model is the strongest"; but by 2026, top engineers began to ask a brand-new question, **"How can the same model perform even better?"** The answer is no longer about larger parameter counts or bigger context windows, but about more refined Harness design: more precise context management, more rational tool orchestration, stricter permission constraints, and smarter auto-compression strategies.

The explosive growth of the open-source community powerfully confirms this trend. Superpowers (42,000 stars) defined a complete development methodology from brainstorming to code review with just 14 Skills; Compound Engineering gave Agents the ability to learn from mistakes and self-evolve; grill-me reshaped AI behavior patterns with only 6 lines of Markdown; Trellis enabled a single specification to work seamlessly across 11 AI platforms. None of them invented new models, they are all dedicated to the same thing: designing better Harnesses.

And the design of these Harnesses is invariably built upon the foundational mechanisms explained in this book, CLAUDE.md carries memory, Skills inject knowledge, sub-agents enable task decomposition, Hooks serve as constraint guardians, and MCP handles external connections. The seven-stage workflow of Superpowers is a masterful orchestration of these five mechanisms; the knowledge accumulation loop of Compound Engineering is an advanced application of CLAUDE.md and Skills.

The components are constant, but the combinations are infinite.

This book provides you with the components. How you combine them depends on the specific problems you face and your engineering judgment.

During this period, one concept kept resurfacing in my mind, **Agentic Loop**, which resembles an elegant Möbius strip.

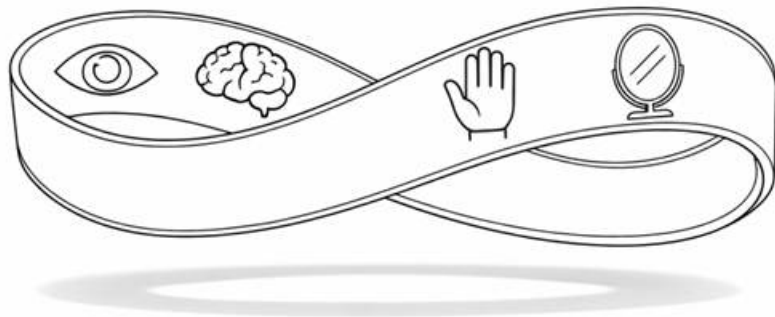


Figure 2: An Elegant Möbius Strip

The "reasoning → tool invocation → result feedback → continued reasoning" loop may seem trivial, but it is the foundation for all emergent complex behaviors in Claude Code. When you submit a bug, it does not simply "take a glance and guess the answer," but undergoes a repeated process of observation, hypothesis, and verification: first consulting error logs, then searching relevant code, delving into the context, and finally making the fix. This complex behavior pattern is not hard-coded into the program, but emerges from the loop structure as an intelligent manifestation.

I have always believed that this loop embodies a kind of philosophical beauty. What it reveals is that **intelligence is not a one-off flash of inspiration, but a process of continuous probing and correction**. In each cycle, the model can trace back to the previous result, so it naturally makes decisions to "take another look" or "try again." This is exactly how humans solve problems, except that our cycles are slower and more easily interrupted by external forces.

As an engineer, what you can do is design **constraints** for this cycle: what it can perceive (tool whitelist), what it can remember (CLAUDE.md + context management), what it can do (permission system), and when it should stop (max-turns + Hooks). You do not need to intervene in the internal decisions of the cycle, that is the model's responsibility. What you control are the boundary conditions of the cycle.

This is precisely the essence of Harness engineering: **it is not about programming behavior, but about constraining behavior**.

If, after finishing this book, you still find yourself wanting more and eager to delve deeper into the methodology of "specification-driven" approaches, I recommend my other book, *SDD in*

Practice: Specification-Driven Development (currently available as a Chinese-language edition). That book systematically explains the complete SDD methodology: from requirements analysis to architectural design, from task decomposition to iterative validation, from Agent design patterns to team collaboration. If this book aims to teach you how to tune Harness, then that book focuses on teaching you how to define specifications, the two complement each other, together outlining a complete picture of software engineering in the AI era.

If you are interested in the theoretical framework of Agent design patterns and wish to understand cognitive modules such as perception, memory, reasoning, action, and governance from a higher level of abstraction, then my book *Designing AI Agents* (Manning) will be your best choice. It systematically organizes 28 Agent design patterns in a single framework, making it one of the most comprehensive architectural references in the field.

These three books each anchor a different dimension: Harness engineering, specification methodology, and design pattern theory. From the bottom up, you gain a builder's perspective, learning how to implement; from the top down, you gain a designer's perspective, learning how to abstract.

Like the master butcher who carves with effortless skill.

But the master butcher does not merely wield the knife, he understands the bones of the ox, he knows where the blade should enter and where it should withdraw, he can sense the barely perceptible gaps between the sinews and membranes.

Claude Code is the knife, your project is the ox, and what this book aims to teach you is the direction of the bones and the location of the gaps.

Now, it's your turn.

Jia Huang

Singapore, June 2026



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

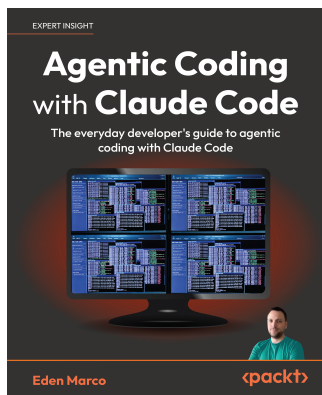
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

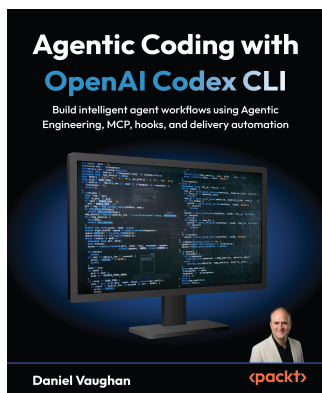


Agentic Coding with Claude Code

Eden Marco

ISBN: 9781806022595

- Design agentic coding workflows in the terminal and IDE using Claude Code
- Build custom automations with reusable slash commands and hooks
- Use Claude Code with a Next.js project to implement AI-driven workflows
- Create persistent AI memory using Claude Code memory files
- Apply MCP for structured context sharing across tools and agents
- Design multi-agent systems using subagents and orchestration patterns
- Enforce coding standards using project documentation and context control
- Scale AI pair programming while keeping code maintainable



Agentic Coding with OpenAI Codex CLI

Daniel Vaughan

ISBN: 9781808348891

- Set up, authenticate, and prompt Codex CLI effectively
- Write AGENTS.md rules and apply patterns that avoid common pitfalls
- Choose approval modes, kernel-level sandboxing, and trust boundaries
- Manage model selection, reasoning effort, context windows, and cost
- Extend the agent with MCP servers, hooks, and skills
- Coordinate sub-agents, multi-agent orchestration, and worktrees
- Integrate Codex into CI/CD, security hardening, and enterprise deployment
- Apply Codex to code review, migration, backend, frontend, and infrastructure as code

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Once you've read *Agentic Architectural Patterns for Building Multi-Agent Systems*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



<https://packt.link/r/180602957X>

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

.claudeignore file 283

A

AI

memory system, 28, 29
significance

AI engineering

debugging tools 277 – 281
large codebase 286
nine core mechanisms, 300 – 302
orchestration
SDD ecosystem and 298
workflow, integrating
security guidelines 281
team adoption 293

AI engineering, cost 272

context management 274
model selection strategy 273, 274
monitoring 276
pricing structure 272, 273
prompt caching 275

AI engineering, large codebase 286

context compression 290
fine-grained management 290
guide searches 288, 289
hierarchical CLAUDE.md 286 – 288
modular context 289
boundaries, building

AI engineering, security guidelines

.claudeignore file 283
API key lifecycle 285
management
file access control 283
Hook security scanning 284
permission modes 281, 282
tool permissions 282, 283
trust assessment, of MCP server 284, 285

AI engineering, team adoption

governance standards, for 294
CLAUDE.md
onboarding training 296
program
organization-level policy management 295
progressive capability building roadmap 295
shared configuration 293

Agent SDK 14, 15

Defense-in-Depth System, 239 – 242
layers
hook mechanism 235
positioning, from tool to component 220
using, to invoke AI agent via Python 221
using, to invoke AI agent via TypeScript 222
versus headless CLI 247, 248

Agent SDK hooks 238

abnormal Claude behaviors 280, 281

agent framework 2 – 4

engineering capabilities 2

agentic loop 16

allowed-tools, in Skill 72

allowed-tools, in Skills

fine-grained bash control 73, 74
syntax
permission design 72
templates

application builder 220

architectural evolution 184, 185

asynchronous Hooks

background execution, without blocking 155

asynchronous generator 222

| | | | |
|---|-------------------|---|-----------------|
| asynchronous message stream | 221 | Chain of Responsibility pattern | 125 |
| automated code quality | | Claude Code | 164 |
| engineering practice | 148 | integrating, into command line workflows | 193, 194 |
| PostToolUse | 148, 149 | session archiving and breakpoint debugging | 279 |
| Stop Hook | 150, 151 | Claude Code component | |
| automated code review, with GitHub Actions | 194 | technical selection guide | 22 – 25 |
| direct CLI invocation | 196, 199 | Claude Code event system | 131 |
| official action, using | 194 – 196 | completion events | 135 |
| B | | configuration | 136 – 139 |
| Bash tool | 11, 73, 74 | newer event types | 135, 136 |
| Bulkhead Pattern | 124 | session-level events | 131, 132 |
| C | | subagent events | 134 |
| CI environment configuration | | tool invocation events | 132 – 134 |
| MCP server configuration | 207 | Claude extension system | |
| required environment variables | 205 | Hooks, role in | 130, 131 |
| template, for GitHub Action | 206 | ClaudeAgentOptions | |
| tuning parameters, performing | 206 | pattern matching, for tool permissions | 229 |
| CI/CD pipeline | 12 | permission modes | 228 |
| CI/CD platform | | using, for fine-grained behavior control | 226 |
| GitLab CI/CD configuration | 207 | ConfigChange | 135 |
| integration | 207 | ConfigChange Hook | 294 |
| Jenkins Pipeline integration | 208 | client-server architecture | 163, 164 |
| configuration | | code analysis web service | |
| local automation scripts | 209 | building | 243, 246 |
| CLAUDE.md | | command files | 256 |
| conditional rule system | 40 – 43 | communicating, with Claude | |
| governance standards | 294 | protocol | 141, 142 |
| hierarchical layout | 286 – 288 | completion events | 135 |
| paradigms, writing | 35 | componentization principle | 97 |
| three-question framework | 36 – 40 | conditional rule system | 40 – 43 |
| versus Skills | 52 – 55 | context window | |
| CLAUDE.md system | 6 – 8 | dilemma | 94, 95 |
| CLAUDE.md, typical project configurations | 43 | custom MCP server | |
| Node.js backend project configuration | 45, 46 | building | 168, 171, 172 |
| Python data project configuration | 46 – 48 | custom tools | |
| React frontend project configuration | 43 – 45 | @tool Decorator | 231 |
| CLI configuration | 165 | enabling, in agent configuration | 233 |
| | | Pydantic Models, using for parameter validation | 233 – 235 |

-
- registering, in agent configuration 233
 - used, for extending capability boundaries of Claude 231
 - D**
 - data flow** 20, 21
 - database**
 - connecting to 167, 168
 - decision tree** 24
 - E**
 - Enterprise ontology** 54
 - extension layer** 8 – 12
 - F**
 - Facade Pattern** 104
 - file access control** 283
 - five-layer memory system**
 - architecture 30 – 35
 - Department Operations Manual 31
 - enterprise policy layer 33
 - Group Code of Conduct 31
 - local layer 34
 - Project Development Specification 31
 - project layer 33
 - rules directory layer 34
 - Special Operations Guide 31
 - user layer 33
 - four-layer architecture model** 4 – 6
 - component relationships and collaboration 18 – 22
 - extension layer 8 – 12
 - external perspective 16 – 18
 - integration layer 12 – 14
 - memory layer 6 – 8
 - programming layer 14, 15
 - G**
 - Gang of Four (GoF)** 125
 - GitHub Actions** 12
 - configuration template for 206
 - GitLab CI** 12
 - GitLab CI/CD configuration** 207
 - H**
 - Headless mode**
 - context across steps, maintaining 203 – 205
 - Interfaces, programming for 214 – 216
 - long-running analysis tasks 202, 203
 - multi-stage CI pipeline 199 – 201
 - progressive implementation strategy 216
 - security best practices 211
 - Headless mode parameter system** 185, 186
 - output-format parameter 186
 - cost guardrails 189
 - execution control 191 – 193
 - json format 187, 188
 - security boundaries 190
 - stream-json format 188
 - text format 186
 - Headless mode, security best practices**
 - audit logs 213
 - container isolation 212
 - cost protection 213
 - principle of least privilege 211
 - secrets management 212
 - Hooks**
 - engineering design methodology 157
 - environment variables 156
 - pitfalls 157
 - role, in Claude extension system 130, 131
 - techniques, debugging 156
 - Hooks, to security protection system**
 - complete configuration 147, 148
 - engineering practice 142
 - PostToolUse 146
 - PreToolUse 143, 144
 - harness** 16
 - headless CLI**
 - versus Agent SDK 247, 248
 - hook mechanism**
 - post-processing execution 237

| | | | |
|--|-----------------|--|------------------|
| pre-execution interception | 235 | security risks and countermeasures | 176 |
| using, for programmatic interception | 235 | three-layered defense mechanism | 175 |
| hookSpecificOutput | 141, 142 | MCP transport layer | |
| hooks | 10 | transmission methods | 164, 165 |
| hooks configuration file | 257 | MCP+Skills | |
| I | | kitchen and recipe | 177 – 179 |
| Information Overload | 36 | MapReduce pattern | 106, 124 |
| integration layer | 12 – 14 | mcp.json configuration file | 166, 167 |
| intelligent orchestration framework | 16 | location and scope | 167 |
| interactive mode | 184, 185 | mechanism | |
| J | | advantages | 41 |
| JSON Schema | | memory layer | 6 – 8 |
| enforcing | 242, 243 | modular context boundaries | |
| JSON-RPC architecture | 163, 164 | building | 289 |
| Jenkins | 12 | multi-stage CI pipeline | 199 – 201 |
| Jenkins Pipeline | | multiple Plugins | |
| integration configuration | 208 | namespace mechanism | 260 |
| json format | 187, 188 | N | |
| L | | N+M problem | 162, 163 |
| library model | 61, 62 | Node.js backend project | |
| long-running analysis tasks | | configuration | 45, 46 |
| output, streaming | 202, 203 | newline-delimited JSON (NDJSON) | 278 |
| M | | O | |
| MCP server | 13 | organization-level plugin management | 265 |
| capabilities, to clients | 164 | output stream | |
| configuration file | 258 | interpreting | 223 |
| configuration, in CI | 207 | message examples, for Claude's response | 223 |
| debugging and troubleshooting | 179, 180 | results, after tool execution | 224 |
| ecosystem | 172 | system/init, used for session initialization | 223 |
| enterprise deployment | 179 | termination messages, on task completion | 224, 225 |
| security mechanisms | 175 | output-format parameter | 186 |
| trust assessment of | 284, 285 | P | |
| MCP server, ecosystem | | PermissionRequest | 133 |
| official servers | 172 | Plugin | |
| popular third-party servers | 173 | installation sources | 258 |
| practical configuration | 174, 175 | | |
| combinations | | | |
| MCP server, security mechanisms | | | |

| | | | |
|--|-----------------|---|-----------------|
| local development and testing | 259 | default | 228 |
| LSP Support and future evolution | 268, 269 | plan | 228 |
| mechanism | 21, 22 | plugin.json | 255, 261 |
| organization-level plugin management | 265 | private marketplace | |
| package capabilities as physical structure | 267, 268 | building | 264 |
| private marketplace, building | 264 | using | 265 |
| private marketplace, using | 265 | processors, types | 139 |
| role | 252 | agent type | 140, 141 |
| routine management | 259 | command type | 139 |
| storage location | 259 | prompt type | 140 |
| team capability pack, building | 260 | programming layer | 14, 15 |
| Plugin, design principles | | prompt engineering | |
| documentation | 267 | CLAUDE.md, writing | 290, 291 |
| incremental iteration | 266 | principles | |
| least privilege | 266 | stepwise confirmation | 292 |
| single responsibility | 266 | techniques, for precise expression | 291, 292 |
| Plugin, physical structure | | pull requests (PRs) | 1 |
| command files | 256 | Q | |
| hooks configuration file | 257 | query function | 221 |
| identify card | 255 | R | |
| MCP configuration file | 258 | React frontend project | |
| sub-agent files | 257 | configuration | 43 – 45 |
| Plugin, team capability pack | | Reference Skills | |
| complete directory | 261 | versus Task Skills | 67 |
| structure | | S | |
| plugin.json | 261 | SDD, ecosystem and workflow | |
| process of releasing | 263, 264 | integrating | 298 |
| security check script | 263 | layers | 298, 299 |
| security scanning sub-agent | 262 | recommended | 299, 300 |
| PostToolUse | 133 | combinations, workflow | |
| PostToolUse Hook | 278 | tools | |
| PostToolUseFailure | 133 | SKILL.md content design | 69 |
| PreCompact | 132 | 500-line rule | 71, 72 |
| PreToolUse | 132, 133 | contractual reference | 71 |
| Principle of Least Privilege | 97 | router mindset | 69 – 71 |
| Prompt-type Hooks | 284 | Server-Sent Events (SSE) | 165, 246 |
| Python data project | | SessionEnd | 132 |
| configuration | 46 – 48 | SessionStart | 131 |
| permission modes | 281, 282 | Single Responsibility Principle (SRP) | 11, 124 |
| acceptEdits | 228 | Skill, from software engineering perspective | |

| | | | |
|---|-----------------|---|-----------------|
| cache optimization and lazy loading | 88 | standard operating procedures (SOPs) | 54 |
| dependency inversion | 88 | stream-json format | 188, 278 |
| open standards | 89 | sub-agent files | 257 |
| principle of least privilege | 89 | subagent Hooks | 152 |
| separation of concerns | 88 | global, versus frontmatter | 152, 153 |
| Skills | | SubagentStart | 153 |
| \$ARGUMENTS | 74, 75 | SubagentStop | 153, 154 |
| building | 78 – 81 | subagent events | 134 |
| collaborating, with subagents | 113 – 120 | SubagentStart | 134 |
| design patterns | 85 – 87 | SubagentStop | 134 |
| directory structure | 55, 56 | subagents | |
| dissecting | 55 | collaborating, with Skills | 113 – 120 |
| dynamic context injection | 75 – 77 | defining and configuring | 98 – 102 |
| financial analysis | 83 – 85 | essence | 95 – 97 |
| from software engineering | 87 | from software engineering | 123 |
| perspective | | perspective | |
| positional parameters | 74, 75 | modes | 103 |
| scope and priority | 77, 78 | token economics | 120 – 123 |
| task-oriented | 81, 82 | subagents, from software engineering perspective | |
| versus CLAUDE.md | 52 – 55 | Bulkhead Pattern | 124 |
| YAML Frontmatter | 57 – 59 | Chain of Responsibility | 125 |
| Skills system | | pattern | |
| progressive disclosure | 59 | MapReduce pattern | 124 |
| Skills system, progressive disclosure | | practical considerations | 125 |
| Budget Mechanism, of | 62, 63 | Single Responsibility | 124 |
| description | | Principle (SRP) | |
| library model | 60 – 62 | subagents, modes | |
| Specification-Driven Development (SDD) | 298 | execution mode | 104 |
| Standard Input/Output (stdio) | 164 | parallel sub-agents mode | 105, 106 |
| Stop Hook | 150, 151 | pipeline mode | 107 – 112 |
| SubagentStart | 134 | read-only mode | 103 |
| SubagentStop | 134 | system execution layer | 130 |
| session management | | T | |
| conversation forking | 230 | Task Skills | |
| session continuity, based on session-id | 229, 230 | versus Reference Skills | 68, 69 |
| session-level events | 131 | TaskCompleted | 135 |
| PreCompact | 132 | TeammateIdle | 135 |
| SessionEnd | 132 | Test-Driven Development (TDD) | 299 |
| SessionStart | 131 | text format | 186 |
| shell hooks | | tool invocation events | 132 |
| versus Agent SDK hooks | 238 | PermissionRequest | 133 |
| software engineering | | PostToolUse | 133 |
| time-honored principle | 96, 97 | PostToolUseFailure | 133 |

| | | | |
|------------------------------------|-----------|-------------------------|-----------------|
| PreToolUse | 132, 133 | | |
| UserPromptSubmit | 134 | | |
| trigger mechanism, of Skill | 63 | U | |
| dual-channel activation | 63 | UserPromptSubmit | 134 |
| mechanism | | W | |
| over-triggering, preventing | 66, 67 | WorktreeCreate | 135 |
| Reference Skills, versus | 67 – 69 | WorktreeRemove | 135 |
| Task Skills | | X | |
| sole signal | 64, 65 | X-ray | 277, 278 |
| under-triggering, preventing | 66 | Y | |
| trigger mechanisms | | YAML Frontmatter | 57 – 59 |
| comparison | 18, 19 | | |

