Claude Code 源码深度研究报告

Xiao Tan
X: x.com/tvytlx
公众号：Xiao Tan AI

April 1, 2026

Contents

1 研究范围与结论总览

1.1 这次到底研究了什么

这次不是只看某一个 prompt 文件，也不是只做“目录级扫一眼”。这次研究的核心，是把公开发布包中可分析的信息整理成系统性研究材料，并沿着以下主线做拆解：

- Claude Code 的整体架构
- 主系统提示词如何动态拼装
- AgentTool / SkillTool 的模型侧协议
- built-in agents 的角色分工
- Agent 调度链路如何跑通
- Plugin / Skill / Hook / MCP 如何接入并影响运行时
- Permission / Tool execution / Hook decision 如何协同
- 它为什么在体验上比“普通 LLM + 工具调用器”强很多

1.2 关键确认事实

本次研究重点关注了以下核心模块：

1. 主系统提示词相关模块
2. Agent Tool Prompt 相关模块
3. Skill Tool Prompt 相关模块
4. Agent 调度相关模块
5. 工具执行链与 Hook 治理相关模块

1.3 最重要的总判断

Claude Code 的强，不是来自某个“神秘 system prompt”，而是来自一个完整的软件工程系统：

- Prompt 不是静态文本，而是模块化 runtime assembly
- Tool 不是直接裸调，而是走 permission / hook / analytics / MCP-aware execution pipeline
- Agent 不是一个万能 worker，而是多种 built-in / fork / subagent 的分工系统
- Skill 不是说明文档，而是 prompt-native workflow package
- Plugin 不是外挂，而是 prompt + metadata + runtime constraint 的扩展机制
- MCP 不是单纯工具桥，而是同时能注入工具与行为说明的 integration plane

一句话总结：

> Claude Code 的价值，不是一段 prompt，而是一整套把 prompt、tool、permission、agent、skill、plugin、hook、MCP、cache 和产品体验统一起来的 Agent Operating System。

2 源码结构全景：它为什么更像 Agent Operating System

2.1 顶层结构暴露出的系统复杂度

从相关目录结构看，Claude Code 至少有这些重要模块：

- `src/entrypoints/`：入口层
- `src/constants/`：prompt、系统常量、风险提示、输出规范
- `src/tools/`：工具定义与具体实现
- `src/services/`：运行时服务，例如 tools、mcp、analytics
- `src/utils/`：底层共用能力
- `src/commands/`：slash command 与命令系统
- `src/components/`：TUI / UI 组件
- `src/coordinator/`：协调器模式
- `src/memdir/`：记忆 / memory prompt
- `src/plugins/` 与 `src/utils/plugins/`：插件生态
- `src/hooks/` 与 `src/utils/hooks.js`：hook 系统
- `src/bootstrap/`：状态初始化
- `src/tasks/`：本地任务、远程任务、异步 agent 任务

这已经说明它不是简单 CLI 包装器，而是一个完整运行平台。

2.2 入口层说明它是平台，而不是单一界面

可见入口包括：

- `src/entrypoints/cli.tsx`
- `src/entrypoints/init.ts`
- `src/entrypoints/mcp.ts`
- `src/entrypoints/sdk/`

也就是说它从设计上就考虑了：

- 本地 CLI
- 初始化流程
- MCP 模式
- SDK 消费者

这是一种平台化思维：同一个 agent runtime，可以服务多个入口和多个交互表面。

2.3 命令系统是整个产品的操作面板

命令系统暴露出非常多系统级命令，例如：

- `/mcp`
- `/memory`
- `/permissions`
- `/hooks`
- `/plugin`
- `/reload-plugins`
- `/skills`
- `/tasks`
- `/plan`
- `/review`
- `/status`
- `/model`
- `/output-style`
- `/agents`
- `/sandbox-toggle`

这说明命令系统不是“锦上添花”，而是用户与系统运行时交互的重要控制面。

2.4 Tools 层才是模型真正“能做事”的根

从 prompt 和工具名能确认的重要工具包括：

- FileRead
- FileEdit
- FileWrite

- Bash
- Glob
- Grep
- TodoWrite
- TaskCreate
- AskUserQuestion
- Skill
- Agent
- MCPTool
- Sleep

工具层的本质，是把模型从“回答器”变成“执行体”。Claude Code 的强，很大程度来自这层做得正式、清晰、可治理。

3 系统提示词系统：它的真实地位

3.1 真正的主入口

主系统提示词相关模块之所以重要，不是因为它写了一大段神奇文案，而是因为它承担了：

- 主系统提示词的总装配
- 环境信息注入
- 工具使用规范注入
- 安全与风险动作规范
- Session-specific guidance 注入
- language / output style 注入
- MCP instructions 注入
- memory prompt 注入
- scratchpad 说明注入
- function result clearing 提示注入
- brief / proactive / token budget 等 feature-gated section 注入

这说明它的 prompt 不是静态字符串，而是一个 system prompt assembly architecture。

3.2 为什么 Prompt cache boundary 很值钱

它会把 prompt 分成更适合缓存的稳定前缀和会话相关的动态后缀。这说明团队不仅在想“提示词写什么”，还在想“怎么控制 token 成本和 cache 命中率”。

这已经是基础设施级别的 prompt engineering。

3.3 它把工程行为规范写进了 prompt

这类提示词会明确要求模型:

- 不要乱加功能
- 不要过度抽象
- 不要瞎重构
- 不要假装测试过
- 先读代码再改代码
- 不要轻易创建新文件
- 有专门工具时不要乱用 shell

这些规则是它稳定性的关键来源之一。

4 Prompt 模块拆解

4.1 身份与基础定位

系统会先把自己定义为 interactive agent，明确它服务的是软件工程任务，而不是普通聊天。这决定了后续整个行为模式。

4.2 基础系统规范

系统会规定:

- 所有非工具输出都直接给用户看
- 工具运行在 permission mode 下
- 用户拒绝后不能原样重试
- 外部结果可能包含 prompt injection
- 上下文会被自动压缩

4.3 工具使用规范

它会明确告诉模型：

- 什么时候该读文件
- 什么时候该搜索
- 什么时候该编辑
- 什么时候该并行执行工具

这不是工具列表，而是工具使用语法。

5 Agent Prompt 与 built-in agents

5.1 Agent Prompt 的价值

Agent Prompt 本质上是在告诉模型：

- 何时该启动 subagent
- 何时该 fork 自己
- 何时不该用 AgentTool
- 如何写给 subagent 的 prompt

这让 agent orchestration 成为模型可理解的正式协议。

5.2 built-in agents 的意义

内建 agents 至少包括：

- General Purpose Agent
- Explore Agent
- Plan Agent
- Verification Agent

这说明它不是一个万能 worker，而是通过角色分工来提高稳定性。

5.3 Verification Agent 为什么值钱

Verification Agent 的核心不是“再看一眼”，而是主动去验证、去尝试打破实现。它要求 build、tests、type-check、真实命令输出和最终 verdict，这对提高任务完成质量非常关键。

6 Agent 调度链深挖：从调度器到运行时主循环

6.1 总体调度链

主链路可以抽象为：

1. 主模型决定调用 Agent 工具
2. 调度器解析输入并选择路径
3. 判断 fork / normal / background / remote / worktree
4. 构造 prompt messages 与 system prompt
5. 组装工具池与上下文
6. 调用子 agent runtime
7. 子 agent runtime 再进入主循环

这说明 agent execution 不是简单“开个新会话”，而是一个完整的 runtime lifecycle。

6.2 fork path 为什么重要

fork 的价值在于：

- 继承上下文
- 控制主线程噪音
- 保持 prompt cache 友好

它解决的是多 agent 系统最难的问题之一：上下文污染。

6.3 background agent 与 foreground agent

系统并不是只支持同步等待式 subagent，而是有：

- foreground sync path
- async background path
- remote launched path
- teammate spawned path

这说明它对 agent lifecycle 的设计已经非常产品化。

7 Skills / Plugins / Hooks / MCP 生态

7.1 Skill 的本质

Skill 不是文档，而是可复用的 workflow package。它让系统能把重复流程变成按需注入的 prompt 资产。

7.2 Plugin 的本质

Plugin 不是普通脚本扩展，而是 prompt、metadata 和 runtime constraints 的组合包。

7.3 Hook 的本质

Hook 是运行时治理层，它可以：

- 改输入
- 给权限建议
- 阻止继续执行
- 注入上下文

7.4 MCP 的本质

MCP 不只是工具桥，还能通过 instructions 影响模型如何理解和使用这些工具。

8 权限、Hook 与工具执行链

成熟 Agent 产品的工具调用并不是模型直接裸调，而是完整的 runtime pipeline：

- schema 校验
- validateInput
- pre-tool hooks
- permission decision
- tool call
- telemetry
- post-tool hooks
- failure hooks

这也是它比很多“会调工具的 Agent”更稳定的重要原因。

9 为什么这种 Agent 产品会这么强

最核心的原因不是模型更聪明一点，而是它把这些东西系统化了：

- Prompt architecture
- Tool runtime governance
- Permission model
- Agent specialization

- Skill workflow packaging
- Plugin / MCP extensibility
- Context hygiene
- Async / background lifecycle
- Product-level engineering

所以它真正厉害的不是某一句 prompt，而是整个 operating model。

10 最终结论

> Claude Code 的真正价值，不是一段 system prompt，而是一个把 prompt architecture、tool runtime、permission model、agent orchestration、skill packaging、plugin system、hooks governance、MCP integration、context hygiene 和 product engineering 统一起来的系统。

这也是为什么它不像一个“会调工具的聊天机器人”，而更像一个真正的 Agent Operating System。