The AI-Native SDLC Playbook（中文版）

> Anthropic Claude Academy 官方课程「The AI-Native SDLC Playbook」的中文整理版。原课程：https://academy.claude.com/courses/ai-native-sdlc-playbook 版权归 Anthropic 所有。本册为学习整理用途。

引言

AI 原生 SDLC Playbook 引言 第 1 课 6 分钟

组织已经开始用 AI 以一年前无法想象的速度写代码，但围绕代码的流程却没有以同样的速度跟上。

许多工程团队仍然沿用同样的审批关卡、评审、交接和策略，拖慢了通过 Claude Code 这类 agentic 编码方案获得的效率提升。

传统 SDLC

软件开发生命周期（SDLC）是把软件从想法带到生产环境的过程。大多数组织都在运行同一套六个阶段的某种变体，涵盖软件的规划、设计、构建、测试、部署和维护。传统上，每个阶段都是一段独立的流程，由不同的角色负责。产品经理写需求，技术架构师把需求变成设计，工程师实现设计，受监管企业的 QA 团队验证软件，发布团队负责发布，运维团队监控线上运行的东西。工作通过文档、工单和签字在各个阶段之间流转。

传统的软件开发生命周期（SDLC）流程繁重，是为了在每一步都确保责任明确、可控。然而，传统 SDLC 设计的初衷是在写代码、实现代码最耗时最昂贵的时代最大化效率，而如今情况已经变了。产品需求文档(PRD)、估时仪式、产品安全评审的存在，都是为了在可能持续数周、数月乃至数季度的开发工作中强制各方对齐。

传统 SDLC 还内置了许多控制手段，其前提是每一步都由人来执行。而如今创造最大价值的组织，已经围绕 agentic AI 现在能做到的事情重建了流程，同时确保人始终在环内（human in the loop）。在本指南中，我们会介绍我们 Applied AI 团队将 Claude 内部集成到 SDLC 各个阶段的最佳实践——这些实践源于我们与客户合作的经验——以加速开发、让流程跑得更快。

当代码不再是瓶颈

当代码不再是瓶颈、构建阶段跑得比传统 SDLC 允许的速度更快时，三件事会成真：

瓶颈转移到构建阶段两侧的阶段：主要是规划、评审/测试和部署，这些阶段仍然以人的速度运行。

控制手段不再匹配现实，变得难以执行。当 diff 是一个人写的时候，逐行人工评审说得通；可一旦大部分 diff 是 agent 写的，这种方式就跟不上了。

治理成本上升，因为例外情况仍然要通过每周或每月才开一次会的委员会和会议来流转。

瓶颈转移了：构建塌缩到 agent 的速度，而它两侧的阶段仍然以人的速度运行。

拿安全瓶颈举例。安全团队是按人类产出规模配置的，所以当 agent 把代码产出放大时，要么评审队列越积越长，要么代码在评审不足的情况下就发布了。受监管的组织两种结果都无法接受，所以它的安全和策略检查必须跟上 agent 的速度。

为了更好地兑现 agentic AI 的效率收益并确保其安全，传统 SDLC 生命周期需要经历与实现阶段同等程度的变革。

什么是 AI 原生 SDLC?

AI 原生 SDLC 是一个重新构想过的流程：保留旧的控制目标，换上新的执行方式。它不再是线性流，而是变成一个环，AI 嵌入到每一个节点。AI 原生 SDLC 推动自动化交接、自动触发后续 plays，从而解决传统 SDLC 各阶段之间靠人工交接、笨拙生硬的问题。

左侧是传统线性 SDLC，右侧是 AI 原生持续循环；人位于循环之上，负责发起、指挥和治理。

转变在哪里

下表标出了传统 SDLC 与由 Claude 支撑的 AI 原生 SDLC 之间这个光谱的两端。大多数组织都处在这两列之间的某个位置。

阶段 传统 SDLC AI 原生 SDLC Plan（规划）需求由委员会收集，经过工作坊和签字提炼，再手工撰写 Claude 直接从源头综合痛点，并写进 intent.md——人类可读、机器可执行 Design（设计）规格由分析师撰写，设计师再解析 需求与设计压缩进一次由 agent 主导的工作会话，由编码为 skills 的标准引导，并在 Git 中做版本管理 Build（构建）测试和代码手写，文档在主开发完成之后才补 测试和代码由 AI 生成，机构知识以带版本管理的机器可读 CLAUDE.md 文件和 skills 维护 Test（测试）在阶段边界设 QA 关卡 持续的 evals 贯穿实现过程 Deploy（部署）人类评审每一行代码，治理发生在评审周期中，而且往往并不一致 多层 agentic 评审，人工评审只保留给受监管和关键代码。治理在 AI 行动的同时即被强制实施，以 hooks 作为审批关卡 Maintain（维护）人盯着生产环境找 bug Agent 监控线上部署。任何被突破的控制带都会被诊断，并作为新的 intent.md 写回循环

把 AI 原生这一列串起来的，是已提交的产物（artifact）。每个阶段结束都把一个产物写入版本控制（包括 intent.md、spec.md、plan.md、diff 及其测试、带评审结论的 PR、事故记录），下一个阶段从读取它开始。在早期阶段，.md 文件是主要产物，因为产品负责人和 agent 都能读同一份文件并对它采取行动。从 Build 往后，产物是代码及其记录。提交链本身就是审计轨迹：谁要求了什么、agent 产出了什么、谁批准了它。

每一个需要判断力的决定，最终仍由人负责。在 agentic SDLC 的世界里，人的注意力随着需要评审的产物一起转移。

plays 如何运作

plays 是本 playbook 的核心，按六个非线性阶段（Plan、Design、Build、Test、Deploy、Maintain）分组，共同覆盖完整的生命周期。

每个 play 覆盖：

改变了什么 如何开始 落地的具体步骤 治理考量 如何衡量它是否有效

plays 是模块化的，组织可以根据自己的独特需求，在不同的时间优先改造不同的阶段。每个 play 都在“Prerequisites”下列出它的依赖，依赖图对此有更直观的呈现。

一个阶段以提交产物结束，这次提交随即启动下一个阶段。被接受的 intent.md 触发需求与设计流程，被批准的 spec.md 触发 plan mode，合并的 PR 触发流水线，生产环境中被突破的控制带写下下一个 intent.md——循环就这样继续下去。

一开始，每个步骤都要你手工提示；最终状态是一个循环：每个被接受的产物都会触发下一道关卡。人的注意力集中在关卡处——评审 agent 标记出来的内容，而不是每个阶段都从零开始。

plays 依赖图。最上面一行的 plays 没有前置依赖；实线箭头指向建立在它之上的 plays，虚线箭头表示有助益但不是必需。

这篇有帮助吗? 下一课 Capture as intent.md 14 课中的第 1 课·The AI-Native SDLC Playbook 引言 引言 引言 Stage 1: Plan Capture as intent.md Stage 2: Design Requirements and design Stage 3: Build Claude Code plan mode as the default starting point The CLAUDE.md Skills as institutional knowledge Parallel sessions and subagents Stage 4: Test Give Claude a feedback loop Continuous evals in CI Stage 5: Deploy AI in the PR review loop Hooks as approval gates CI/CD integration and

deployment Stage 6: Maintain Closing the loop on metrics Closing Closing thoughts and resources Course complete 传统 SDLC 当代码不再是瓶颈 什么是 AI 原生 SDLC？转变在哪里 plays 如何运作

第 1 阶段·将想法记录为 intent.md

AI 原生 SDLC 实战手册 将想法记录为 intent.md 第 2 课 4 分钟

软件开发流程由 intent.md 开启，而 intent.md 可能来自不同的入口：一个人产生了一个想法、有人提交了一张工单，或是一条告警暴露了一次事故（参见第 6 阶段：维护）。

当想法来自某个人时，他会先和 Claude 头脑风暴，产出一份 Markdown 格式的规格草案（proto-spec）。在传统 SDLC 中，同一个人接下来还得去说服产品团队中的某位成员，让对方和他一起——或替他——把这个想法正式写下来。

Claude 产出的这份规格草案人类可读、受版本控制，下一阶段拿过来就能直接用。这份规格草案最终保存为 intent.md。

无论意图来自事件触发还是某个人，后续步骤都是一样的：agent 写出的 intent.md，在提交之前都要经过产品负责人（product owner）的审阅与修正。

有什么变化

传统	AI 原生
一个想法要先流经积压条目（backlog）、用户故事、故事点估算和细化会议，才会有人动手做。每一次交接都发生一次所有权转移，等它传到工程团队手里时，已经和提出者最初的意思隔了好几层。	提出者直接和 Claude 头脑风暴，用自己的话把结果写进 intent.md，形成一份规格草案。这份产物写清楚了想要什么、为什么想要、在哪些约束下做；可重复的流程则编码成 skill。

准备工作

前置条件：无。

基础设施：为非工程师提供 Claude 访问入口（claude.ai 或 Cowork）；一份约定好的 intent.md 模板；一个共享的、受版本控制的 intent 存放地，由产品负责人照看。产品线单一时，最简单的存放地就是产品仓库里的 intent/ 目录——这样，文档链就和由它衍生出的代码待在了一起。只有当 intent 横跨多个仓库时，单独建一个 intent 仓库才值得那份额外开销；在 monorepo 中，它只是一个目录。这个存放地与已经承担记录职责的 Jira 或需求工具如何配合，见第 3 阶段：构建中的「遗留系统」一节。

这套基础设施由平台团队或工程团队一次性搭建。由于贡献者会来自组织各处，需要一名技术人员先把 intent 存放地建起来，并决定谁有写入权限。

仓库就绪之后，没有 Git 经验的贡献者不必直接使用 Git。只要接入一个连接器（例如连到 GitHub 这类版本控制系统），Claude 就能从 claude.ai 或 Cowork 里替他们把 Markdown 文件提交上去。

如何执行

提出者用自己的话向 Claude 描述问题。可以说说现在做不到什么、这个想法会影响到谁、更好的状态应该是什么样、哪些内容不在范围内。不需要使用任何正式的语言。

继续头脑风暴，直到想法变得具体。Claude 会问出分析师会问的那些问题：范围、用户、约束条件，以及怎样才算成功。

请 Claude 用组织的模板把结果写成 intent.md。该模板可以编码成一个 skill——由技术人员搭建、负责人签字确认。模板一般覆盖：问题、预期成果、受影响的用户与系统、约束条件，以及待决问题（open questions）。

提出者把 Claude 理解错的地方逐一改正。

把 intent.md 提交到共享存放地。作者与时间戳就此进入记录，产品负责人从这里接手这个想法。

效果示例

intent.md：

```
# 意图：claims 状态自助服务
Author: J. Ortiz (claims operations). Status: draft.
## Problem
Customers phone the contact center to ask where their claim is.
Handlers spend roughly a third of call time on status-only queries.
## Proposed outcome
Customers see claim status, next step and expected date in the portal.
## Affected users and systems
Claims handlers, portal team, claims-core API.
## Constraints
No new PII in the portal session. Existing authentication only.
## Open questions
Do third-party loss adjusters need access too?
```

治理考量

治理依据就是已提交的 intent.md：作者、时间戳、完整的修订历史都在里面，并记录在 intent 存放地的 Git 历史中。产品负责人负责批准——这份 intent 是被接受并流入第 2 阶段：设计，还是被驳回，以产物被合并（merge）或被关闭评审（closing review）的形式留下记录。

如何度量

领先指标：从第一次对话到 intent.md 完成提交所花的时间。直接从 intent 存放地的 Git 历史读取即可，那里记录了作者与时间戳。预期这一时间会从过去数周的「需求挖掘与细化」周期，缩短到几小时。

滞后指标：存活率，即被产品负责人接受、进入第 2 阶段：设计（而不是被关闭）的 intent.md 所占比例。接受或拒绝的决定，以产物被合并或评审被关闭来记录。另外，还可以统计同一变更在 spec.md 首次提交之后，intent.md 又被改动了多少次。

这篇内容有帮助吗?

上一课 引言 下一课 需求与设计

第 2 课（共 14 课）·AI 原生 SDLC 实战手册 将想法记录为 intent.md 引言 引言 第 1 阶段：计划 将想法记录为 intent.md 第 2 阶段：设计 需求与设计 第 3 阶段：构建 默认以 Claude Code 的 plan 模式为起点 CLAUDE.md 把 skill 沉淀为组织知识 并行会话与 subagents 第 4 阶段：测试 给 Claude 一个反馈回路 在 CI 中持续跑 evals 第 5 阶段：部署 让 AI 参与 PR 评审回路 用 hooks 充当审批闸门 CI/CD 集成与部署 第 6 阶段：维护 用指标闭合回路 结语 结语与资源 课程完成

有什么变化 准备工作 如何执行 效果示例 治理考量 如何度量

阶段 2 ·需求与设计

AI 原生 SDLC 手册 需求与设计 第 3 课 4 分钟

产品负责人批准 intent.md 后，Claude 接手它，产出一份需求与设计规格（spec），整个过程由组织在品牌、安全、合规、UX 方面的 skill 引导。

这份 spec 由产品负责人评审，但不是他写的。流程的目标是产出一份工程团队能照着排计划的 spec，并把有疑点的地方标注出来。

前端工作是最好的例子：intent.md 一被接受，产品负责人就在 Claude Design（测试版）里照 intent.md 搭出设计稿，反复迭代，再导出给 Claude Code 去实现。

什么变了 传统 AI 原生 需求与设计是两个独立阶段，由不同团队分别执行。分析师把想法正式化成需求文档，设计师再把需求解析回设计。分开是为了责任可追，但流程既慢又损耗信息。两个阶段在同一次由 prompt 驱动的会话里完成：Claude 拿 intent.md 直接产出需求与设计规格，受组织的 skill 约束，有疑点的地方直接标出。准备工作 前置条件：写好一份 intent.md；品牌、安全、合规和 UX 政策都要以 skill 的形式写下来。基础设施：一名能访问 Claude 的产品负责人即可，不需要任何工程技能。怎么执行 产品负责人打开一个会话，让组织的 skill 都可用，并附上 intent.md。产品负责人的 prompt 指向 intent、点名各项约束，并要求标出疑点。先手动跑一遍，再把它固化成组织级的斜杠命令。接着把触发器设为 intent home 中 intent.md 被接受的那一刻：一个非交互式 job 在 merge 时触发，加载组织的 skill 跑一遍，把 spec.md 作为 pull request 提交（背后的管道由阶段 5「部署」的 CI/CD 打法覆盖）。此后，产品负责人的第一次介入就是评审。还是同一位产品负责人，拿 spec 对照最初的想法评审：它解决了原来说的问题吗？intent.md 里未决的问题，是得到了回答，还是原样带到了下一轮？先处理标注出来的疑点——这些正是分析师会往上呈报的点。工程团队看到 spec 之前，产品负责人要和每个疑点对应的策略负责人逐一敲定。把 spec.md 和 intent.md 提交在一起。这一对文件记录下当初要求了什么、最后决定了什么。spec 和 intent 是否进入构建阶段，由产品负责人拍板；凡组织划为较高风险的改动，都要先咨询技术主管。这个决定永远由人来下——接受 spec 的那一刻，就启动了阶段 3「构建」里的 plan mode 打法。长什么样

prompt：

阅读附带的 intent.md，产出一份把它集成进我们现有代码库的需求与设计规格。运用你可用的 skill，让方案符合我们的品牌规范、安全策略与 UX 标准。把规格完整写进 spec.md，做到可以直接交给工程团队。清楚描述所有疑点，尤其是你无法同时满足互相冲突的政策时。

复制 prompt 治理层面的考虑

策略冲突不用等几周后的评审才暴露——spec 还在写的时候，现行政策就被读取并应用了。组织的 skill 作为约束施加在 spec 上。spec、生成它的 prompt、以及当时生效的 skill 版本，全部留在版本控制里。产品负责人对 spec 签字放行，并把标注的疑点分发给对应的策略负责人。

怎么衡量 先行指标：同一改动从 intent.md 提交到 spec.md 提交之间所花的时间（两个 Git 时间戳之差），与旧有的「需求 + 设计」周期对比。滞后指标：构建开工之后的需求返工量。数一数同一改动里提交日期晚于首个 plan.md 提交的 spec.md 提交——Git log 直接就能给出这个数。这篇有帮助吗？上一课 以 intent.md 捕获意图 下一课 让 Claude Code plan mode 成为默认起点 第 3 课（共 14 课）·AI 原生 SDLC 手册 需求与设计 引言 引言 阶段 1：计划 以 intent.md 捕获意图 阶段 2：设计 需求与设计 阶段 3：构建 让 Claude Code plan mode 成为默认起点 CLAUDE.md 让 skill 沉淀为组织知识 并行会话与 subagents 阶段 4：测试 给 Claude 一个反馈回路 在 CI 中持续跑 evals 阶段 5：部署 PR 评审循环里的 AI hook 作为审批闸门 CI/CD 集成与部署 阶段 6：维护 用指标闭合回路 结语 结语与资源 课程完成 什么变了 准备工作 怎么执行 长什么样 治理层面的考虑 怎么衡量

阶段 3 ·把 Claude Code 的 plan mode 设为默认起点

AI 原生 SDLC 实践手册 把 Claude Code 的 plan mode 设为默认起点 第 4 课 5 分钟

工程师在 plan mode 下启动 Claude Code 会话，把阶段 2（设计）产出的已批准 spec.md 交给 Claude，让 Claude 对工程师进行访谈式提问，反复迭代方案，直到工程师满意为止。

会有什么改变 传统做法 AI 原生做法 工程师读完设计稿就开始写代码。改动具体怎么做——动哪些文件、跑哪些测试——只存在于工程师的脑子里，最好的情况也只是留在 ticket 的评论里。没有人能审查它。审查者看到的第一样东西就是已经完成的 diff，到那时再返工就慢了。工作从一份书面方案开始——Claude 在 plan mode 里产出这份方案，此模式下它可以只读代码库而不做任何改动。工程师在代码写出来之前修正方案，批准后的版本提交为 plan.md，供后续阶段对照检查。

开始之前 前置条件：意图产物（intent.md 或 spec.md），如果有的话；有 CLAUDE.md 文件会很有帮助。基础设施：能访问仓库的 Claude Code。

如何执行 1. 工程师在 plan mode 下启动与 Claude 的会话。2. 工程师把 intent.md 和 spec.md 交给 Claude，请它产出一份实现方案，方案要指明哪些文件会改动、工作的先后顺序，以及能证明改动成立的测试。3. 审问这份方案：问这次改动可能破坏什么、哪一步风险最高、Claude 放弃了哪些别的选项。4. 反复迭代，直到一个从未看过这段对话的工程师，只凭方案就能把改动实现出来。5. 把批准后的方案提交为 plan.md。方案自此进入审计轨迹，PR 审查这个 play（阶段 5：部署）会拿最终的 diff 与它对账。6. 接受方案，让 Claude 动手实现。方案足够扎实时，实现往往一遍就能过。7. 实现偏离方案时，在同一个 commit 里更新 plan.md。可以考虑用 hook 强制两者保持同步。

实际长什么样

plan.md：

markdown

计划：claims 状态自助服务（来自 intent.md 2026-06-02）

Files that change

portal/src/claims/StatusPanel.tsx (new), claims-api/routes/status.py, claims-api/tests/test_status.py

Order of work

1. Add the status endpoint behind existing auth.
2. Panel against the endpoint.
3. Wire into the portal nav.

Risks

The claims-core API rate-limits at 50 rps; the panel must cache.

Proof

test_status.py covers the four claim states; screenshot matches the approved mock.

治理考量

设计评审发生在任何代码生成之前——此时改变方向还只是改一份文档的事。plan mode 本身就强制了这一点：在工程师接受方案之前，Claude 无法编辑文件。方案及其修订过程都会被记录，连同是谁批准了它。常规改动由工程师自己批准，组织认定的高风险改动则要交给技术负责人或架构师。

如何衡量 先行指标：一次实现就合并的改动占比，以及从方案批准到 PR 合并的时间（所需数据放在 PR 元数据里）。滞后指标：每次改动的返工轮数（同样取自 PR 元数据），以及合并后的 diff 与已提交的 plan.md 仍然相符的频率。

Claude Code 的 auto mode

Claude Code 也可以跑在 auto mode 下：工程师迭代并批准方案，之后 Claude 逐项应用改动而无需每处编辑都征求确认。随着后面几个 play 的护栏逐渐成熟（调校好的 CLAUDE.md、把策略固化成代码的 skill、拦截危险操作的 hook、Claude 能自己跑的测试套件），auto mode 会成为例行工作的默认模式：spec.md 写得很紧、爆炸半径小、代码早已被测试覆盖。

重心就此转移：从用户盯着 agent 做改动、逐个审查动作，转向在更长的自主会话结束后审查产物。结合 worktrees 使用时，auto mode 还能进一步释放个人和团队层面的并行能力，也是自主运行 SDLC、如阶段 6（维护）所述闭环运转的基础。

遗留系统与真相之源

现有 SDLC 流程多半已经在追踪产物了，只是不用 Markdown 文件而已。工作项可能在 Jira 里，需求放在带法规追溯能力的工具里，设计稿在 Figma 中，变更审批要走变更委员会。这些系统很难被替换——审计和监管部门已经认可它们，别的团队也依赖它们——所以 AI 原生 SDLC 必须迁就现状。流程产出的每份产物，都要指定一个系统作为真相之源，其余系统只保留副本或链接。

下面这些配置都能实现”单一真相之源”，每种产物可以各有选择：

把仓库作为真相之源。Markdown 产物是权威记录，遗留系统引用 commit 里的文件。对工程主导的组织来说，这通常是最干净的配置之一：所有记录都在一个工具里，由同一个时间戳权威把关。以遗留系统为真相。Jira、ServiceNow 或需求管理工具持有权威记录，Markdown 产物只是工作副本。Claude 在会话开始时读取记录，并在产出 spec 或 plan 的同一个会话里，通过 Model Context Protocol (MCP) 连接器把结果写回遗留系统。以链接为最低标准。所有产物都标注记录 ID，所有遗留记录都包含对应 Markdown 文件的 commit SHA。向 AI 原生 SDLC 迁移时，链接方案是很好的起步点——因为此时事实上有两个真相之源。

只要两者之间存在链接，或者其中一方被明确指定为真相之源，遗留系统和 AI 原生的 Markdown 优先系统就可以共存。

这有帮助吗? 上一课 需求与设计 下一课 CLAUDE.md 第 4 课 / 共 14 课·AI 原生 SDLC 实践手册 把 Claude Code 的 plan mode 设为默认起点 引言 引言 阶段 1：规划 记录为 intent.md 阶段 2：设计 需求与设计 阶段 3：构建 把 Claude Code 的 plan mode 设为默认起点 CLAUDE.md 把 skill 变成机构知识 并行会话与 subagents 阶段 4：测试 给 Claude 一个反馈回路 在 CI 里持续跑 evals 阶段 5：部署 AI 参与 PR 审查回路 用 hook 充当审批门禁 CI/CD 集成与部署 阶段 6：维护 用指标闭环 结语 结语与参考资料 课程完成 会有什么改变 开始之前 如何执行 实际长什么样 治理考量 如何衡量 Claude Code 的 auto mode 遗留系统与真相之源

阶段 3 ·CLAUDE.md

AI 原生 SDLC 实践手册 CLAUDE.md 第 5 课 3 分钟

CLAUDE.md 给 Claude 提供的，是新人入职第一天需要的那类上下文：约定、命令、架构，以及团队最常踩的坑。过去存在人脑里、散落在 wiki 上的知识，变成了一份文件——agent 每次会话开始都会读它，整个团队共同维护，每犯一次错就迭代一次。

开始之前 前置条件：无。基础设施：一个仓库、装好的 Claude Code，以及一名熟悉代码库的工程师。怎么执行 在仓库里运行 /init。Claude 会根据它看到的内容生成一份 CLAUDE.md 初稿。把生成的文件删减到"新人第一天需要"的程度。保留 build、test、lint 命令、真正重要的约定，以及 Claude 反复出错的地方。把 CLAUDE.md 提交进仓库根目录的 Git，让整个团队共享同一版本，变更像代码一样被评审。这里有一条好用的小规则：当 Claude 犯同一个错误两次，就把纠正写进 CLAUDE.md。让它控制在一页以内——因为 Claude 在会话开始时会把整份文件读完，任何过时的内容都在白白占用上下文。

它长什么样

CLAUDE.md：

markdown

支付服务

Commands

- Build: make build
- Test: make test (unit), make itest (integration, needs docker)
- Lint: make lint (runs in CI; fix before pushing) ## Conventions
- Java 21, Spring Boot 3. No new Lombok.
- Money is always BigDecimal, never double.
- Every endpoint needs an integration test in src/itest. ## Architecture
- api/ holds REST controllers, core/ holds domain logic, adapters/ talks to external systems.
- Kafka events are defined in schemas/; never edit generated classes. ## Things Claude gets wrong
- Do not bump dependency versions; the platform team owns them.
- The legacy v1/ package is frozen; changes go in v2/. 治理方面的考量

CLAUDE.md 受版本控制，所以 agent 依据的指令是可评审、可审计的。团队约定通过这份文件落地，对它的修改会记录在 Git 历史里，代码所有者（code owner）在 PR 评审中批准这些变更。

怎么衡量 领先指标：Claude 重复犯 CLAUDE.md 本应拦下的错误的频率。对 CLAUDE.md 的修正或改动应记录在 Git 历史中。滞后指标：从 PR 历史看，团队新成员从加入到达成第一个合入的 PR 所需的时间。这有帮助吗？上一课 把 Claude Code plan mode 作为默认起点 下一课 把 Skills 当作机构知识（institutional knowledge）第 5 课，共 14 课·AI 原生 SDLC 实践手册 CLAUDE.md 导言 导言 Stage 1: Plan Capture as intent.md Stage 2: Design Requirements and design Stage 3: Build Claude Code plan mode as the default starting point The CLAUDE.md Skills as institutional knowledge Parallel sessions and subagents Stage 4: Test Give Claude a feedback loop Continuous evals in CI Stage 5: Deploy AI in the PR review loop Hooks as approval gates CI/CD integration and deployment Stage 6: Maintain Closing the loop on metrics Closing Closing thoughts and resources Course complete 开始之前 怎么执行 它长什么样 治理方面的考量 怎么衡量

阶段 3 ·把 skill 变成机构知识

AI 原生 SDLC 实践手册 把 skill 变成机构知识 第 6 课 4 分钟

Skill 是组织把机构知识变成可执行能力的方式：规则被明确写下、纳入版本控制、被广泛套用，政策变化时集中更新。经验法则：机构知识中必须被一致执行的部分，写成 skill；本属于 CLAUDE.md 或 prompt 的内容，不要写成 skill。

开始之前 前置条件：无。手头有 CLAUDE.md 会更好——它把 agent 的工作知识留在仓库里——但 skill 并不依赖它。基础设施：一条有明确负责人的政策，外加一份书面形式的真相之源。

如何执行 挑一条目前执行得并不一致的规则——可以是安全标准、API 设计约定，也可以是品牌规范。把它写成 skill：一个目录，内含一份 SKILL.md，frontmatter 声明它何时触发，正文写明要做什么。工程师以政策负责人的真相之源为底稿，请 Claude 帮忙写成。把 skill 放进仓库的.claude/skills//，随代码一起发布；也可以做成插件在组织内分发。测试 skill 确实会触发：换几种不同说法让 Claude 做相关任务，每次都确认 skill 被加载。政策变更时，同步更新 skill，并让政策负责人签字确认这次改动。工程师在下一轮会话里自动拿到新版本。

实际长什么样

.claude/skills/secure-api-review/SKILL.md：

markdown

name: secure-api-review description: Apply the API security standard. Use whenever creating or modifying an external-facing endpoint, reviewing API code, or generating an OpenAPI spec. —

安全 API 评审

When you create or change an API endpoint: 1. Authentication: every endpoint requires the gateway JWT; no anonymous routes outside /health. 2. Input validation: validate request bodies against the OpenAPI schema and reject unknown fields. 3. Audit: every state-changing endpoint emits an audit event with actor, action, entity and timestamp. 4. Data classification: fields tagged pii in the schema must never appear in logs or error messages. Run scripts/check-endpoints.sh and include its output in your summary.

治理考量

Skill 是一种控制手段，不过属于建议性质。它让 Claude 大概率在写代码的同时套用政策，但没有任何机制强制某次会话照办。必须无条件成立的政策，需要在 skill 背后再加一层确定性的东西——比如拦下动作的 hook，或者在 PR 上重新核对政策的审查环节。skill 让违规变得罕见，hook 让违规几乎不可能发生。skill 的每次调用都会记进会话轨迹，政策负责人要像审代码一样审 skill 的改动。

如何衡量 先行指标：从政策负责人批准政策变更，到更新后的 skill 合并，所用的时间——取 skill 目录对应 PR 的时间。滞后指标：PR 审查中引用该政策的意见条数。skill 开始边写代码边套用政策后，这条数应当趋近于零。如果迟迟不降，要么 skill 没被触发，要么它的文字已经偏离了官方政策。

hook：构建期的护栏

Skill 是建议性控制，hook 是它背后那层确定性的东西。Claude 在实现阶段的大多数动作都是文件编辑和 shell 命令，所以构建期正是 hook 最容易触发的地方。

构建期 hook 可以：

拦截对受保护路径的编辑，比如生成出来的类或已冻结的包 文件编辑后自动跑 formatter 和 linter，让偏差永远攒不下来 把密钥挡在 diff 之外 支撑任何必须毫无例外成立的政策型 skill

hook 会在每个匹配它的动作上运行，所以构建期 hook 必须够快，并且只盯被改动的那个文件。更重的检查——比如整套测试套件——应该放到 commit 或 PR 阶段。

需要向真人征求批准的 hook，归阶段 5（部署）的门禁管：构建期弹出批准请求，等于把真人重新放回所有并行会话的关键路径上。

这有帮助吗？上一课 CLAUDE.md 下一课 并行会话与 subagents 第 6 课 / 共 14 课·AI 原生 SDLC 实践手册 把 skill 变成机构知识 引言 引言 阶段 1：规划 记录为 intent.md 阶段 2：设计 需求与设计 阶段 3：构建 把 Claude Code 的 plan mode 设为默认起点 CLAUDE.md 把 skill 变成机构知识 并行会话与 subagents 阶段 4：测试 给 Claude 一个反馈回路 在 CI 里持续跑 evals 阶段 5：部署 AI 参与 PR 审查回路 用 hook 充当审批门禁 CI/CD 集成与部署 阶段 6：维护 用指标闭环 结语 结语与参考资料 课程完成 开始之前 如何执行 实际长什么样 治理考量 如何衡量 hook：构建期的护栏

阶段 3 ·并行会话与 subagents

The AI-Native SDLC Playbook 并行会话与 subagents 第 7 课 4 分钟

一名工程师可以同时驱动多条工作流。

并行会话（parallel session）是另一个完整的 Claude Code 实例，在自己的 Git worktree 中处理独立任务。每个独立会话对其他会话一无所知，唯一的连接点是调度它们的工程师。

subagent 则运行在单个会话内部，是一个带有自己的上下文窗口和工具权限限制的限定作用域助手，适合处理多个任务中反复出现的子工作，比如验证应用是否按预期运行。

并行会话提高了工程师同时进行的任务数量，而 subagent 让每个会话专注于自己的任务。工程师的工作是调度并审查所有这些会话。

有什么变化 传统 AI 原生 一名工程师一次只能处理一个任务，一天/一周里有相当大一部分时间花在构建、测试和评审上。等待期间切换任务虽然可行，但上下文切换太耗神，几乎没人愿意这么做。一名工程师同时运行多个 Claude 会话，每个会话在自己的 worktree 中处理自己的任务。重复出现的子工作变成 subagent，带各自的上下文和工具权限限制。工程师的角色转向编排，最终转向构建和监控循环。

开始之前 先决条件：CLAUDE.md，因为所有会话都会读取这个文件。反馈回路（阶段 4：测试）在这里也有帮助，因为当会话可以自行验证工作时，就不那么需要工程师的监督了。基础设施：一个 Git 仓库，因为隔离来自 worktree；同时把权限设置调到合适的程度，让会话不必在组织认为安全的命令上等待审批提示。

如何执行 工程师使用计划模式那课（阶段 3：构建）产出的计划，把工作拆成触碰不同文件的任务，看清哪些地方的工作是相互独立的。共享文件的任务在同一个会话中逐个执行。每个并行任务有自己的 worktree，例如在一个终端里运行 claude —worktree feature-auth，在另一个终端里运行 claude —worktree fix-rate-limit。worktree 是在自己的分支上的独立检出，避免会话在文件上互相冲突。两个或三个会话是合理的起点。实际上限是一个人能认真审查多少条工作流，所以只有审查跟得上时，才增加会话。把重复出现的子工作变成 subagents，定义在.claude/agents/ 下的 Markdown 文件里，每个文件包含名称、何时使用它的描述，以及它允许触碰的工具。例子包括：在主 agent 完成后去除多余复杂度的代码简化器；运行应用并检查行为的验证器；以及探索代码库、汇报结果而不淹没主上下文的研究者。把定义提交进 Git，让整个团队共享。

长什么样

.claude/agents/verifier.md:

markdown

name: verifier description: Runs the app and checks the change works before the session reports done tools: Bash, Read

---

用 make run 启动应用。运行被改动的行为以及最相邻的两个流程。汇报你运行了什么、看到了什么，以及任何与 plan.md 不一致的行为。不要修复任何东西，只汇报。

治理考量会话更多意味着产出更多，所以控制手段必须来自仓库里的配置。仓库中的 hooks 和权限设置对所有会话生效，会话做了什么都会被记录并归因到运行它的工程师。

如何衡量领先指标：评审质量保持的前提下，每名工程师的并发会话数（从 OpenTelemetry 导出统计），以及一天中花在调度而非等待上的时间占比。滞后指标：每名工程师每周合并的变更数，结合 PR 历史确定的返工率一起看。

这对
你有
帮助
吗?
上一
课
Skills
作为
机构
知识
下一
课 给
Claude
一个
反馈
回路
14 课
中的
第 7
课·
The
AI-
Native
SDLC
Play-
book
并行
会话
与
sub-
agents
简介
简介
阶段
1：规
划 把
意图
记录
为 in-
tent.md
阶段
2：设
计 需
求与
设计
阶段
3：构
建
Claude
Code
计划
模式
作为
默认

阶段 4 ·给 Claude 一个反馈回路

The AI-Native SDLC Playbook 给 Claude 一个反馈回路 第 8 课 4 分钟

永远给 Claude 一条验证自己工作的途径——无论测试、构建，还是截图对比。会话会先检查自己的工作、修正自己的错误，然后工程师才看得到。

不要把反馈回路和验证器 subagent（阶段 3：构建）混为一谈。反馈回路贯穿整个任务，按工作需要反复运行。验证器 subagent 则是把最终检查打包成一种形式：等会话认为工作已经完成时，用一个全新的上下文窗口再跑一遍。这样一来，结论就不会被当初生成代码的那些假设所左右。

有什么变化 传统 AI 原生 代码能用的信号姗姗来迟。CI 要几分钟后才知道，测试人员要几天后，生产环境要几周后。当代码由 agent 产出时，信号来得晚就意味着必须有人检查它的全部输出，而这个人就成了瓶颈。会话被赋予在有人看到之前先自查工作的手段。跑测试、跑构建、截图。Claude 一直迭代到检查通过，所以交到工程师手里的东西已经过了检查。搭建这个回路的责任落在运行会话的工程师身上，下面的步骤就是写给他们的。

开始之前 先决条件：无。基础设施：一套测试套件和一个构建流程，各自一条命令就能在本地跑起来。对于 UI 工作，让 Claude 能看见结果是关键——要么给它一个浏览器工具，要么通过 MCP 接一个截图工具。

如何执行 如果目前检查工作需要一串命令外加一些环境知识，就把它包进一个单一目标，例如 make test 或 npm test，失败时以非零状态码退出。在 CLAUDE.md 的 Commands 一节里，列出每条命令，并附上一个健康输出的示例。把目标说清楚，并让它可量化，这样 Claude 不用问你就能自己检查工作，例如：“test_status.py 里的所有测试都通过”“截图与所附的 mock 一致”，或者“端点返回 200 并带有新字段”。修 bug 时，先写失败的测试。让 Claude 把 bug 复现成一个测试，跑一遍，并确认它失败的原因正是你预期的。提交这个测试。然后才让 Claude 让它通过——不许改测试本身，用最后一步提到的 test-file hook 来强制这条限制。一个在修复之前就存在、而 agent 无法改写的测试，就是 bug 已经消失的证据。做 UI 工作时，用视觉检查把回路闭合。给 Claude 一个浏览器或截图工具，把 mock 交给它，让它自己迭代。实现、截图、对比、调整。来回两三轮很正常，而且每轮结果都应该比上一轮好。把验证纳入“完成”的定义。这条指令写在 CLAUDE.md 里：“报告任务完成之前先运行测试，并展示输出。”最后，回路本身也需要保护——一个在修代码的 agent 绝不能有能力削弱对所修代码的检查。在修复任务期间阻止编辑测试文件的 hook 就能做到这一点。另一种做法是在评审时检查 diff，拒绝任何动了测试的改动。

长什么样

CLAUDE.md 验证区块：

markdown ## 验证你的工作

- Build：make build（必须以“Build succeeded”结束）
- Test：make test（全绿；绝不跳过或删除失败的测试）
- Lint：make lint（零警告）

报告任何任务完成之前，先运行以上三项，并把输出贴出来。如果测试失败，修代码，而不是修测试。治理考量 强制了什么：任务被报告完成之前的验证，以及修复期间禁止 agent 编辑测试文件的限制——两者都实现为 hooks，落在组织希望得到保障的位置。证据是什么：make test 的原始输出、构建日志，或 Claude 运行并粘贴出来的截图对比——证据来自工具链本身。记录在哪里：记在会话 transcript 里，OpenTelemetry 导出会把 transcript 转送到组织的可观测性栈；也记在 PR 的 check run 里，评审者和日后的任何审计者都能看到。谁来批准：审查该 PR 的代码所有者。机械性的证据已经附在上面，他们可以专注在意图和风险上。

如何衡量 领先指标：agent 所写变更在 CI 上的一次通过率，CI 系统本来就支持这个统计。滞后指标：每个 PR 的评审时长（来自 PR 元数据）——一旦测试接住了原本由评审人抓的问题，这个数字就该下降；以及来自事故追踪器的变更失败率。

这对你有帮助吗？上一课 并行会话与 subagents 下一课 CI 中的持续 evals 14 课中的第 8 课·The AI-Native SDLC Playbook 给 Claude 一个反馈回路 简介 简介 阶段 1：规划 把意图记录为 intent.md 阶段 2：设计 需求与设计 阶段 3：构建 Claude Code 计划模式作为默认起点 The CLAUDE.md Skills 作为机构知识 并行会话与 subagents 阶段 4：测试 给 Claude 一个反馈回路 CI 中的持续 evals 阶段 5：部署 AI 进入 PR 评审回路 hooks 作为审批闸门 CI/CD 集成与部署 阶段 6：维护 闭环度量 收尾 收尾思考与资源 课程完成 有什么变化 开始之前 如何执行 长什么样 治理考量 如何衡量

阶段 4 ·在 CI 里持续跑 evals

AI 原生 SDLC 实践手册 在 CI 里持续跑 evals 第 9 课 3 分钟

Evals 就是 AI 原生世界的阶段闸门式 QA（stage-gate QA）。落到实践上，它是一套在 agent 配置每次变化时都会运行的测试套件。换入新模型，或者重写了 prompt，eval 套件会告诉你：agent 干活的质量是否还跟以前一样。

这套 evals 应当被当作活套件来经营。模型在进步，曾经能区分好坏的评测用例会渐渐失效，就得靠持续监控不断补进新用例。

视使用场景而定，有些团队更愿意按固定节奏离线跑这些 evals，而不是每次变更都触发。下面的步骤针对的是持续评估（continuous evaluations）。

开始之前 前置条件：CLAUDE.md 与反馈回路（阶段 4：测试）。基础设施：能非交互式运行 Claude Code 的 CI，以及预算足以跑 eval 的 API key。

如何执行 平台工程师从近期工作中收集 20 到 50 个真实任务，每个任务都带着它的预期结果或可接受结果。把每个任务写成一条 eval，也就是一个 prompt 加上判定”可接受”的检查项（测试通过、lint 干净、行为不变、符合策略）。套件在 CI 中非交互式运行：既按计划任务定时跑，也在 CLAUDE.md、skills 或 hooks 发生任何变化时跑——这份配置在指挥 agent 的行为，理应获得代码享有的回归测试待遇。配置变更以运行结果为准入闸门。一次导致通过率下降的 skill 改动，合入之前必须先接受评审。每一起生产事故都要对应一条 eval，由事故归属团队编写，并作为回归测试常驻套件。

实际长什么样

.github/workflows/agent-evals.yml:

yaml name: Agent evals on: pull_request: paths: [‘CLAUDE.md’, ’.claude/**’] schedule: - cron: ’0 2 * * ’jobs: evals: runs-on: ubuntu-latest steps: - uses: actions/checkout@v4 - run: npm install -g @anthropic-ai/claude-code - name: Run eval suite env: ANTHROPIC_API_KEY: ${{ secrets.ANTHROPIC_API_KEY }} run: | for eval in evals/.json; do claude -p “$(jq -r ‘.prompt’$eval)” —allowedTools “Read,Edit,Bash(make test)”
—output-format json > result.json ./evals/check.sh “$eval”result.json done

治理考量

Evals 给 QA 一扇跟得上 agent 产出的闸门：通过率阈值作为合并检查（merge check）强制执行，每次运行都会留日志，便于跨时间比较结果；配置变更由归属团队批准放行。

如何衡量 先行指标：eval 通过率随时间的变化——套件每次运行都会报告；以及一起生产事故要多久才能沉淀为一条常驻 eval。滞后指标：CI 拦下的回归，对照事故跟踪系统中统计到的生产环境回归。这有帮助吗？上一课 给 Claude 一个反馈回路 下一课 AI 参与 PR 审查回路 第 9 课，共 14 课·AI 原生 SDLC 实践手册 在 CI 里持续跑 evals 引言 引言 阶段 1：规划 记录为 intent.md 阶段 2：设计 需求与设计 阶段 3：构建 把 Claude Code 的 plan mode 设为默认起点 CLAUDE.md 把 skill 变成机构知识 并行会话与 subagents 阶段 4：测试 给 Claude 一个反馈回路 在 CI 里持续跑 evals 阶段 5：部署 AI 参与 PR 审查回路 用 hook 充当审批门禁 CI/CD 集成与部署 阶段 6：维护 用指标闭环 结语 结语与参考资料 课程完成 开始之前 如何执行 实际长什么样 治理考量 如何衡量

第 5 阶段·AI 参与 PR 评审

AI-Native SDLC 实践手册 AI 参与 PR 评审 第 10 课 4 分钟

Claude 既给出评审，也接受评审。它对照组织的策略评审收到的 PR，也在自己提交的 PR 上回应评审意见。这让工程师在 PR 评审中可以专注于行为层面——说到底就是判断意图与风险。

有什么变化

传统 AI 原生 评审产能围绕人力的产出规划。一个 PR 要等评审者通读全部内容，评审质量随评审者当时的负荷起伏，作者在后面追着催，积压却越堆越高。所有 PR 都接受同一套评审流程，结论按严重程度分级。人力的注意力上移到更高层面——这个改动是否实现了计划中的意图、风险是否可接受。

开始之前 先决条件：第 3 阶段（Build）产出的、已更新的 CLAUDE.md 文件；如果评审流程要强制执行书面策略，还需要 skills 和定义好的 subagents。基础设施：一个安装了 Claude 集成的仓库——要么由管理员启用受管的 Code Review（research preview）服务，要么在自家 CI 里跑 claude-code-action，需要时可通过 Amazon Bedrock、Google Cloud 的 Vertex AI 或 Microsoft Foundry 调用模型（部署选项在 CI/CD 篇里有介绍）。要求 code owner 批准的分支保护策略也值得配上。

如何执行 受管的 Code Review 服务是最快的起步方式。管理员启用它并选择仓库。当需要掌控流水线，或希望 API 调用走自家云协议时，就在自己的 CI 中用 claude-code-action 跑评审（相关管道细节见 CI/CD 篇）。技术负责人把评审策略写成仓库根目录下的 REVIEW.md，按组织在意的各个关卡划分：bug 与逻辑错误；安全与漏洞；对照 spec（需求篇中的 spec.md）、实现计划（plan 模式篇中的 plan.md）和设计原则的合规性。REVIEW.md 还定义什么是 Important、什么只是 Nit，以及哪些内容要跳过。技术负责人设定人工介入的阈值。评审结论本身既不通过也不阻止一个 PR，分支保护仍然要求 code owner 的批准。平台工程师若想按评审结论把关合并，可以读取该检查运行以机器可读计数发布的严重程度统计。当评审者或作者在某条评审意见上 @claude 时，Claude 会回应这条意见并推送修复。PR 线程中同时记录请求与改动。这个修复循环由 claude-code-action 驱动。在受管服务中，评论 @claude review 则会触发一次全新的评审。对 Claude 自己开的 PR，可以更进一步——让 Claude 全程照看 PR 直到合并。团队会把这个循环包进自定义 slash command：扫清 PR 上未解决的评审意见和失败的检查，逐一处理并推送修复，直到 PR 变绿、只等 code owner 批准。评审结论会回写给 CLAUDE.md。当评审第二次标记同一类错误时，纠正内容就作为该次评审的一部分写进 CLAUDE.md；由于评审会读 CLAUDE.md，从下一个 PR 起这个错误就会被拦下。评审还会在某个改动让 CLAUDE.md 过时的时候发出提示。技术负责人每月调一次配置：给结论打分让评审者持续改进，并在 REVIEW.md 中设 Nit 数量上限。生成路径（generated paths）和 CI 已强制检查的内容不计入。

实际效果

REVIEW.md:

markdown

评审指令

Passes

Run three passes and tag each finding with its pass: – Bugs: logic errors, broken edge cases, subtle regressions – Security: injection risks, authentication gaps, PII in logs – Compliance: the change matches spec.md, plan.md and our design principles ## What Important means here Reserve Important for findings that would break behavior, leak data or breach a policy. Style and naming are nits. ## Cap the nits Report at most five nits per review; summarize the rest as a count. ## Do not report Generated files under src/gen/ and anything CI already enforces.

治理考量

职责分离得到保留，因为写代码的 agent 没有途径批准自己的代码。REVIEW.md 中的评审策略应用于所有 PR，结论、修复、评分与批准都记录在 PR 历史中，因此 PR 本身就是审计记录。批准来自走分支保护的人工——在结论的支撑下做出决定。

如何衡量 先行指标：首次评审时间——应缩短到分钟级；以及不需要人工碰分支就解决掉的评审意见占比，数据直接存在 Git 上。滞后指标：合并前拦截到的缺陷与漏洞，对照漏到生产环境的那些——数据来自 PR 历史和事故追踪器。

这篇有帮助吗? 上一课 CI 中的持续 evals 下一课 作为审批闸门的 hooks 第 10 课 / 共 14 课·The AI-Native SDLC Playbook AI 参与 PR 评审 引言 引言 第 1 阶段：计划 用 intent.md 捕捉需求 第 2 阶段：设计 需求与设计 第 3 阶段：构建 Claude Code plan 模式作为默认起点 CLAUDE.md 作为机构知识的 skills 并行会话与 subagents 第 4 阶段：测试 给 Claude 一个反馈回路 CI 中的持续 evals 第 5 阶段：部署 AI 参与 PR 评审 作为审批闸门的 hooks CI/CD 集成与部署 第 6 阶段：维护 闭环于指标 结语 结语与资源 课程完成 有什么变化 开始之前 如何执行 实际效果 治理考量 如何衡量

第 5 阶段·作为审批闸门的 hooks

AI-Native SDLC 实践手册 作为审批闸门的 hooks 第 11 课 5 分钟

构建阶段把 hooks 用作护栏：没有人参与，直接放行或拦截动作（第 3 阶段：构建）。hook 也可以选择询问：把动作暂停，直到某个具体的人批准——这正是发布闸门需要的形态。

这个做法放在第 5 阶段（Deploy），是因为发布闸门是最清晰的使用场景，但 hooks 并不专属于部署：只要 Claude 在行动，它们就会运行。例如，在第 3 阶段（Build）期间，hooks 可以拦截没有变更单就修改迁移脚本和基础设施（infra）文件的行为；在第 4 阶段（Test），当修复任务进行中，hooks 能阻止 agent 改动测试文件。

开始之前 先决条件：无。基础设施：一份书面清单，列出变更流程要求的所有审批项。

如何执行 工程负责人与变更管理、合规团队一起，把必须保留的人工审批关卡列出来——例如变更管理签核、发布授权，以及对受保护路径的修改。平台工程师把每个关卡表达成一个 hook：一个在 Claude 行动前运行的脚本，可以给出三种结果之一——放行（allow）、询问（ask）或拦截（block）。团队的 hooks 放进 Git 里的.claude/settings.json；不容妥协的 hooks 放进由平台或 IT 管理员持有的受管设置中，个别工程师无法把它们关掉。拦截应当能自我解释：当 hook 停下某个动作时，原因和获得批准的途径都要出现在 Claude 的输出里。

实际效果

项目.claude/settings.json 里一个独立的示例：

json { “hooks”: { “PreToolUse”: [{ “matcher”: “Bash”, “hooks”: [{ “type”: “command”, “com-mand”: “${CLAUDE_PROJECT_DIR}/.claude/hooks/production-gate.sh”}] }] } }

而闸门脚本本身（.claude/hooks/production-gate.sh）：

bash #!/bin/bash

生产部署需要具名的发布授权

cmd=$(jq -r ‘.tool_input.command’< /dev/stdin) if [[“$cmd”== “deploy” && “$cmd”== “produc-tion”]]; then if [-z “$RELEASE_APPROVAL”]; then echo “Production deploys need a release authorization.”>&2 exit 2 # exit 2 blocks the action; the message goes to Claude fi fi exit 0

治理考量

hooks 就是审批闸门。闸门条件每次执行、对每个人都强制执行。每一次放行与拦截的裁决都连同时间戳记入日志。闸门还定义了什么算作批准——是一张已批准的变更单，还是发布经理的签核。

受监管企业的受管设置

面向受监管企业的受管设置，由平台团队通过移动设备管理（MDM）或管理控制台下发。工程师无法编辑或覆盖其中的任何设置。见下：

json { “permissions”: { “deny”: [“Read(.env*)“,”Read(./secrets/**)“,”WebFetch”, “Bash(curl)”, ” Bash(wget)”], “allow”: [“Bash(git *)“,”Bash(make build)“,”Bash(make test)“,”Bash(make lint)”], “disableBypassPermissionsMode”: “disable”}, “allowManagedPermissionRulesOnly”: true, “sand-box”: { “enabled”: true, “failIfUnavailable”: true, “allowUnsandboxedCommands”: false, “network”: { “allowedDomains”: [“git.internal.example.com”, “registry.npmjs.org”] }, “credentials”: { “files”: [{ “path”: “~/.ssh”, “mode”: “deny”}, { “path”: “~/.aws/credentials”, “mode”: “deny”}], “envVars”: [{ “name”: “GITHUB_TOKEN”, “mode”: “deny”}] } }, “allowManagedHooksOnly”: true, “disableSide-loadFlags”: true, “allowManagedMcpServersOnly”: true, “strictKnownMarketplaces”: [{ “source”: “github”, “repo”: “example-corp/approved-plugins”}], “requiredMinimumVersion”: “2.1.193”}

这些设置在控制层面做什么：

permissions.deny 让密钥进不了 agent 的上下文，并拦截通过工具发生的任意网络外联。permis-sions.allow 预先批准安全的内循环操作，这样 deny 清单不会变成提示疲劳（prompt fatigue）。disable-BypassPermissionsMode 加上 allowManagedPermissionRulesOnly，意味着没有任何工程师、项目文件或命令行参数能扩大这些规则的适用范围。sandbox 补上 permissions 管不到的部分。对 WebFetch 做工具级的 deny，拦不住 shell 命令触网；而操作系统级的域名白名单则把外联彻底挡死——两者在不同层次上执行同一个目标。failIfUnavailable 和 allowUnsandboxedCommands 把沙箱变成一项前置条件：沙箱无法初始化时 Claude Code 拒绝启动；在沙箱里失败的命令，也不能拿到沙箱外重试。credentials 处理的是 deny 规则漏掉的场景。permissions.deny 约束的是 Claude 的文件工具，但默认情况下，沙箱中的 shell 命令仍然可能读到 ~/.ssh 或 ~/.aws/credentials。这个块会拒绝这类读取，并把列出的密钥从沙箱命令的环境中剥离。allowManagedHooksOnly 意味着只有受管设置里定义的 hooks 会运行；用户、项目和本地设置里的 hooks 都会被拦下，包括上文.claude/settings.json 的独立示例。要让这堂课的审批闸门持续生效，就把它定义在受管文件自己的 hooks 块中。disableSideloadFlags 和 strictKnownMarketplaces 意味着，工程师机器上的任何 skill、agent、hook 或 MCP server 都来自组织批准的插件市场，而不是某个主目录。市场白名单控制哪些东西可以安装；而那种为了单次运行而旁路加载插件、agent 或 MCP 配置的参数，会在启动时被拒绝。allowManagedMcpServersOnly 把 agent 的工具面变成一份由平台团队持有的白名单。requiredMinimumVersion 让低于批准底线版本的 Claude Code 拒绝启动——也就是说，这些控制措施由组织实际评估过的构建版本来执行。

把这个示例当作起点，按你自己的环境去定制。每一条 deny 规则都在拿走一部分能力，平衡点取决于仓库的数据分级。设置参考文档（settings reference）记录了所有键，包括仅限受管的那些。

如何衡量 就 hooks 本身而言：

- 先行指标：在每个审批闸门上等待的时间。每次 hook 裁决都会连同时间戳和放行或拦截的结论写入 OpenTelemetry 导出，因此每个闸门的等待都看得见。
- 滞后指标：hooks 上线前后，闸门违规漏到生产环境的次数——数据来自事故追踪器。

这篇有帮助吗？上一课 AI 参与 PR 评审 下一课 CI/CD 集成与部署 第 11 课 / 共 14 课·The AI-Native SDLC Playbook 作为审批闸门的 hooks 引言 引言 第 1 阶段：计划 用 intent.md 捕捉需求 第 2 阶段：设计 需求与设计 第 3 阶段：构建 Claude Code plan 模式作为默认起点 CLAUDE.md 作为机构知识的 skills 并行会话与 subagents 第 4 阶段：测试 给 Claude 一个反馈回路 CI 中的持续 evals 第 5 阶段：部署 AI 参与 PR 评审 作为审批闸门的 hooks CI/CD 集成与部署 第 6 阶段：维护 闭环于指标 结语 结语与资源 课程完成 开始之前 如何执行 实际效果 治理考量 受监管企业的受管设置 如何衡量

第 5 阶段·CI/CD 集成与部署

The AI-Native SDLC Playbook CI/CD 集成与部署 第 12 课 4 分钟

在 CI/CD 流水线中以非交互方式运行 Claude Code，对执行过程做沙箱化处理，让长时运行的 agent 安全执行；通过 MCP 集成暴露部署能力；在 agent 真正需要回滚之前，先演练好回滚路径。

变化在哪里

传统 AI 原生 流水线运行确定性脚本，一切需要判断的事都等着人来处理：比如给 flaky 测试做分类、写 changelog、或者排查构建为什么挂掉。部署和回滚是人在压力之下照着执行的 runbook。Claude 以非交互方式运行在流水线内部，处理那些需要判断的步骤，运行在带限定凭据的沙箱里。部署工具通过 MCP 暴露给 agent，于是那个写了变更、测了变更的工作流，也能在组织按环境定义的闸门之内，把变更发布出去并回滚。

开始之前

前置条件：AI 参与 PR 评审，以及 hooks 作为审批闸门——因为闸门必须先存在，自动化才能加速任何东西通过它们。

基础设施：装好 claude-code-action 的 CI 平台，或任何能调用 claude -p 的 runner；通过 API 获取模型访问，如果流量必须留在组织的云协议范围内，则走 Amazon Bedrock、Microsoft Foundry 或 Vertex AI；为部署目标准备 MCP servers；为 agent 任务准备沙箱 profile，不持有常驻生产凭据。

如何执行

平台工程师从只读的判断步骤开始。在流水线任务里用 claude -p 给失败的构建做分类、总结 flaky 测试，或起草 changelog。

在既有闸门之后添加写入步骤，处理修 lint、更新自动生成的文档、或通过 @claude 提及回应评审意见之类的任务。agent 写出的任何东西都经由分支保护以 PR 形式出现，agent 没有任何直接推到 main 的路径。

执行是沙箱化的。agent 任务在容器里运行，受网络策略约束，使用短时有效的限定 token，默认不持有生产凭据。

通过 MCP 暴露部署能力。Deploy、status 和 rollback 变成按环境限定的工具，于是 agent 的部署权限是一个 allowlist，而不是一份带凭据的 shell 脚本。

按环境分级授权。在开发环境，agent 可以自由部署。在生产环境，agent 准备发布、由发布经理批准，用 hook 强制生产闸门。staging 介于两者之间。

回滚应该是流水线里演练最多的路径：一条 agent 能执行的命令，并且定期在 staging 里实际跑一遍。闭环玩法（第 6 阶段：维护）在控制带被突破时会调用这条回滚，所以它必须提前被验证过。

实际效果

流水线步骤：

yaml – name: Triage failed build if: failure() run: > claude –p “Read the build log at out/build.log. Identify the most likely cause, say whether the failure looks flaky or real, and write a three–line summary for the PR thread.”» triage.md

治理考量

治理原则是：agent 可以一直行动到生产闸门之前，但不能越过它。下面的控制手段落实这一原则。

分支保护把 agent 写出的任何东西都变成 PR，没有直达 main 的路径。生产部署 hook 会拦住发布，直到具名的发布经理批准为止。每次非交互运行都以 agent 自己的身份行动，所以流水线日志能把 agent 做的事和触发它的工程师做的事区分开。按环境划分的权限层级决定 agent 在通往闸门的路上有多大行动空间。

如何衡量

先行指标：不需要呼叫人就能完成分类的流水线故障占比，数据取自 CI/CD 流水线日志。滞后指标：DevOps Research and Assessment（DORA）指标，CI 系统和部署工具本来就会输出这些数据。

这篇有帮助吗？上一课 Hooks 作为审批闸门 下一课 用指标闭环

第 12 课，共 14 课·The AI–Native SDLC Playbook

CI/CD 集成与部署 简介 简介 第 1 阶段：规划 把意图写进 intent.md 第 2 阶段：设计 需求与设计 第 3 阶段：构建 Claude Code plan mode 作为默认起点 CLAUDE.md Skills 作为机构知识 并行会话与 subagents 第 4 阶段：测试 给 Claude 一个反馈回路 CI 中的持续 evals 第 5 阶段：部署 AI 进入 PR 评审回路 Hooks 作为审批闸门 CI/CD 集成与部署 第 6 阶段：维护 用指标闭环 收尾 收尾思考与资源 课程完成

变化在哪里 开始之前 如何执行 实际效果 治理考量 如何衡量

第 6 阶段·用指标闭环

The AI–Native SDLC Playbook 用指标闭环 第 13 课 6 分钟

前面的每一个阶段都需要有人来启动，第 6 阶段则把重心转向让 Claude 自主运行，从而把环闭合。例如，一个持续运行的监控 agent 可以在 bug 工单产生后顺势创建一份 intent.md，然后依次流过需求、规划、构建、测试和评审各个阶段。第 6 阶段：维护以 headless 方式运行，阶段之间设有一道独立的置信闸门——一次确定性检查或一个对抗式评审 agent——由它决定上一阶段的产出是继续往前走，还是升级给人处理。

变化在哪里

传统 AI 原生 维护是一个被动响应的阶段。所有工单或事故都要等一个人动手处理、重启流程。凌晨 3 点的告警可能没人看见，工单可能躺在待办池里直到有人捡起来，而如果另一场火先烧起来，post-mortem 的整改动作可能根本走不到代码库。触发即启动——控制带被突破、一张工单、一条频道消息或一个计划任务，都能在没有人在链路里的情况下唤起 Claude。Claude 做诊断，只经由带闸门的路径行动，并把发现写成 intent.md，随后进入上面描述的那些阶段。人负责分流和评审这些工作，不再需要亲手启动它们。

一个确定性脚本监视着生产环境，当控制带被突破时唤起 Claude。监控控制带突破是理解这个环如何自主运转的一个好例子；本阶段末尾的 Claude Tag（公开 beta）一节则覆盖从其他渠道进来的工作。

开始之前

前置条件：intent.md——它给这个环一份结构化的产出物用来重新启动；Claude 加速的 PR 评审、hooks 作为行动边界，以及 CI/CD 的回滚路径（最高自治层级会调用它）。

基础设施：一个检测脚本可以查询的指标存储（Prometheus、CI 系统的 API 或同类物）、仓库的读权限、在 CI 中以非交互方式运行 Claude Code 的方法，或者为接收 webhooks 的服务准备的 Agent SDK。

如何执行

服务负责人或平台工程师挑选一个带稳定滚动基线的指标，比如 CI 测试失败率、部署后 5xx 率或 PR 周期时间。

他们编写检测脚本，通常是对一个滚动窗口求均值和标准差，再套用规则（Western Electric 或类似规则），让控制带既能捕捉缓慢漂移，也能捕捉尖峰。脚本纳入版本控制并配有单元测试，检测保持完全确定性，不涉及任何模型。

响应层级定义在纳入版本控制的配置里（见下面的 bands.yaml）。1σ 时脚本只记日志，2σ 时它唤起 Claude 以只读方式做诊断，3σ 时 Claude 可以行动——但也只能通过打开一个进入评审闸门的 PR，或触发预先批准的 runbook。

触发层可以是 GitHub 或 GitLab 里的定时 workflow、来自现有监控栈的 webhook，或网络内部的 cron job。Claude 以无状态方式运行，要么作为 CI runner 上的非交互步骤，要么作为沙箱容器里的 Agent SDK 服务；部署与模型访问的选项由 CI/CD 玩法覆盖。由于运行是无状态、非交互的，一个环可以在没有任何人启动的情况下开始并结束。

agent 把诊断结果按第 1 阶段：规划 的格式写成 intent.md，涵盖异常及其证据、提议的目标结果、受影响的系统，以及任何未决问题。从那里起，这个发现就像其他任何工作一样进入流水线。

服务负责人或值班工程师对队列做分流，把面向产品的发现转给产品负责人。立即修复、排期，或驳回。驳回会调校控制带，帮助降低噪声。

修复上线后，为这起事故添加一个 eval（见持续 evals 玩法），确保这类问题在未来得到防护。

实际效果

例如，一个监控 CI 测试失败率的 bands.yaml：

yaml metric: ci_test_failure_rate baseline: rolling_30d rules: western_electric tiers: 1sigma: { action: log } 2sigma: { action: diagnose, tools: “Read,Grep,Bash(gh run view *)”} 3sigma: { action: propose, routes: [pull_request, runbook:rollback-deploy] }

治理考量

层级边界由纳入版本控制的配置强制实施，权限与托管设置拒绝生产访问。每次调用、每个发现和每条分流决策都带时间戳记入日志。服务负责人对发现做分流和批准，由此产生的变更走正常的 PR 评审闸门，agent 可能触发的 runbook 都经过事先批准。

如何衡量

先行指标：从控制带被突破到分流队列里出现一份 intent.md 的时间——对照过去从事故到 post-mortem 整改动作的时间。检测脚本的日志里有突破的时间戳和事故的层级。

滞后指标：最终变成已合并修复的发现占比（拿分流队列对照真实的 PR 历史），以及同类事故的重复率——随着修复不断往 eval 套件里补充用例，重复率应当下降。

示例

CI 测试失败率突破 3σ 时，agent 隔离 flaky 测试或打开一个 revert PR，由评审闸门做决定。

窗口内有部署而部署后 5xx 率突破 3σ 时，agent 触发现有的回滚流水线。

PR 周期时间触发漂移规则时，agent 为工程领导层写一份报告——说明这套 harness 对流程指标和对生产指标一样有效。

Claude Tag：Claude 随叫随到

事故也可以通过其他途径到达，比如 Slack 或 Microsoft Teams 这类工作通讯应用。事故可以长这样：晚上 10 点事故频道里一条要求紧急修复的 Slack 消息——现在它可以立即被处理。Claude Tag（目前已在 Slack 提供公开 beta）让 Claude 以自己独立的身份成为这些频道的成员，于是每起新事故都有第一响应者，响应过程本身也进入环里，成为未来事故的记忆。

对话和机构知识留在频道中，频道里的任何人都能引导并实际执行响应。任何团队成员都可以实时检验假设、探索新选项、展开调查，频道历史则增加了可审计性。通过访问 MCP，Claude 核实指标已回到基线，在线程里确认，并把 post-mortem 写进一个纳入版本控制的 lessons 文件，供未来的调查阅读。

事故不是 Claude Tag 接手的唯一工作。通过 MCP 在工单上被 @ 到、或在频道里被叫到，Claude 都会以同样的方式分流工作。小而边界清晰的修复会以 PR 的形式穿过评审闸门；更大的事项则写成给第 1 阶段：规划 的 intent.md——到这一步，这个环开始自己喂养自己。

这篇有帮助吗？上一课 CI/CD 集成与部署 下一课 收尾思考与资源

段：维护 用指标闭环 收尾 收尾思考与资源 课程完成 变化在哪里 开始之前 如何执行 实际效果 治理考量 如何衡量 Claude Tag：Claude 随叫随到

收尾思考与资源

The AI-Native SDLC Playbook 收尾思考与资源 第 14 课 2 分钟 ## 收尾思考

模型与 harness 已经变得更先进，让组织得以改变的不仅是代码的生产方式，更是整个软件开发生命周期。这一转变始终把人的判断放在流程的中心，同时把大型企业组织的治理与监管要求纳入考量。

本指南汇集了我们 Applied AI 团队每天为客户践行的诸多真实最佳实践，希望它对你来说是一份实用、能直接上手的资源。

下面这份文档清单，就是平台团队按本指南搭起文中所述控制手段所需的全部资料，顺序大致对应你们实际推出的先后次序。

为你的组织配置 Claude Code（管理员决策图；从这里开始）设置参考与优先级，涵盖每一个 managed-only 键 Claude admin console 中的服务端托管设置 权限 沙箱化（OS 级文件系统与网络隔离）Hooks 指南与 hooks 参考 Skills 插件与私有市场（skills 与 hooks 如何在组织范围内分发）Managed MCP（对 agent 工具面的集中控制）企业部署概览（Amazon Bedrock、Vertex AI、Microsoft Foundry）企业网络配置 监控（OpenTelemetry）与分析仪表盘 Compliance API（Enterprise 活动流、聊天检索与删除）安全模型 这篇有帮助吗？上一课 用指标闭环 下一课 课程完成

第 14 课，共 14 课·The AI-Native SDLC Playbook

收尾思考与资源 简介 简介 第 1 阶段：规划 把意图写进 intent.md 第 2 阶段：设计 需求与设计 第 3 阶段：构建 Claude Code plan mode 作为默认起点 CLAUDE.md Skills 作为机构知识 并行会话与 subagents 第 4 阶段：测试 给 Claude 一个反馈回路 CI 中的持续 evals 第 5 阶段：部署 AI 进入 PR 评审回路 Hooks 作为审批闸门 CI/CD 集成与部署 第 6 阶段：维护 用指标闭环 收尾 收尾思考与资源 课程完成

收尾思考