O'REILLY®

Compliments of

harness

AI 原生软件交付

提高软件质量且加快开发速度的可靠方法

Nick Durkin, Eric Minick
& Chinmay Gaikwad

目录

序言

软件行业正处于一个关键时刻。系统日益复杂，用户需求呈指数级增长，而无论是财务、声誉还是运营方面的失败成本都达到了前所未有的高度。然而，尽管经历了数十年的发展，许多团队仍然受制于过时的实践：手动部署、被动救火，以及因自身复杂性而崩溃的工具链。本书旨在弥合我们当前状态与未来目标之间的鸿沟。它是一份指引图，帮助您从脆弱、高风险的交付模式转向 AI 驱动的自主交付 —— 一个软件能自我部署、系统能自我修复、创新速度超越风险的未来。

本书读者

本书为以下读者撰写：

- 寻求用智能自动化取代繁重劳动的工程师和 DevOps 实践者
- 负责将 DevOps 成熟度与业务成果（如速度、韧性和成本控制）对齐的技术领导者
- 希望了解 AI 原生交付如何加速价值实现的产品经理和创新者
- 任何关注软件未来的人，从 CTO 到学生，准备重新思考部署、测试和可观察性领域的一切可能性

本书创作缘由

作为受过专业训练的软件工程师，我们多年来一直致力于研究软件开发和交付的演进。然而，OpenAI 于 2022 年末推出 ChatGPT，对我们而言标志着一个转折点。与业界许多人一样，我们不仅将生成式 AI 视为一个编码助手，更将其视为重塑整个交付流水线的催化剂。

在接下来的三年里，我们对 AI（从代码生成到代理工作流）将如何重塑部署、测试和治理进行了假设、测试和验证。

我们撰写本书是因为软件交付的利害关系已经改变。微服务、云原生架构和 AI 生成代码的兴起，使得传统的 DevOps 1.0 实践变得力不从心。团队如今在一个流水线中要同

时处理十余种工具，与“依赖地狱”作斗争，并面临类似 SolarWinds 式的供应链攻击威胁，同时还要努力满足消费者科技巨头所塑造的用户期望。

现有资源大多侧重于历史性的 DevOps 概念，或者抽象地推测 AI。本书旨在将这些点连接起来。基于 25 年来从敏捷的早期成功到 Kubernetes 编排革命的经验教训，我们将技术严谨性与前瞻性洞察相结合。现代工具印证了这种转变。我们展示了 AI 不仅在自动化任务，还在重塑协作、治理和创新。

当然，变化的步伐永不停歇。代理式 AI、自运营系统和新框架每月都在涌现。尽管我们力求本书能够适应未来，但我们也承认某些细节会随之演变。然而，核心原则不会改变：自动化繁重工作、优先考虑韧性，以及使交付与业务价值对齐。

本书导读

本书并非追逐潮流，而是旨在构建在复杂环境中蓬勃发展的系统。每一章都将理论与真实世界的案例相结合，从《凤凰项目》的 DevOps 寓言到 AI 驱动的部署。无论您是逐章阅读还是深入特定章节，读完本书后，您都将掌握改造交付流程和提升团队影响力所需的知识。

第一章，“AI 原生 DevOps 之路”追溯了软件交付从混乱的手动部署到 DevOps 1.0 实践（及其文化转变和自动化工具）的演变，同时强调了 DevOps 2.0 旨在通过 AI 原生能力和集成平台解决的当前挑战，例如微服务复杂性和工具链蔓延。

第二章，“源代码管理”追溯了源代码管理（SCM）从早期系统到 Git 当前主导地位的演变（截至 2022 年，近 95% 的开发者使用 Git），解释了现代 SCM 如何解决代码冲突和版本跟踪问题，同时提供了关于分支策略、GitOps、AI 集成以及您组织实施考虑的实用指导。

第三章，“持续集成的构建与部署前测试步骤”探讨了持续集成（CI）从其历史根源到现代 AI 增强实践的演变，详细阐述了构建自动化、智能缓存和策略性测试方法如何协同工作，以加速软件交付，同时在整个部署前流水线中保持质量和安全性。

第四章，“部署到测试环境”引导您完成 CI 与生产部署之间的关键阶段，探讨如何在不同环境间建立一致的部署流程，利用基础设施即代码（IaC）提升可靠性，实施 GitOps 工作流，优化测试策略（包括新兴的 AI 驱动方法），并自动化升级决策 —— 所有这些都是为了在开发与实际使用之间建立无缝衔接，同时保持速度和稳定性。

第五章，“保护应用程序和软件供应链安全”审视了不断演进的软件供应链安全格局，详细阐述了组织如何通过左移实践、软件构件供应链级别（SLSA）框架、软件物料清

单（SBOM）和 AI 增强型安全工具来保护其应用程序，同时培养一种将安全性整合到整个软件开发生命周期（SDLC）中的协作式 DevSecOps 文化。

第六章，“混沌工程与服务可靠性”探讨了混沌工程作为构建弹性系统的系统化方法，展示了如何将受控故障实验（从简单的延迟测试到复杂的基础设施扰动）与服务级别目标（SLOs）、错误预算以及持续集成和持续交付（CI/CD）流水线相结合，从而创建一种持续韧性的文化，将不可预测的故障转化为可预期、可管理的事件。

第七章，“部署到生产环境”通过一个真实案例研究的视角，深入探讨了生产部署的关键挑战，提供了一个涵盖现代部署治理、渐进式交付策略和 AI 增强验证技术的综合框架，这些技术共同将高风险部署转变为受控、可观察且可回滚的流程，从而保护您的应用程序和业务。

第八章，“特性管理与实验”阐述了特性管理和实验如何作为现代软件交付的基石，展示了特性标志如何实现主干开发、团队解耦和渐进式交付，同时 AI 增强型实验如何将产品决策从主观争论转变为数据驱动的洞察，从而最大化业务价值。

第九章，“AI 与自动化在云成本管理中的应用”审视了复杂的云成本管理世界，追溯其演变为 FinOps 实践的过程，审视了多云挑战，并展示了 AI 驱动的解决方案如何优化资源分配、强制执行治理策略，并将成本效率与业务目标和环境可持续性目标对齐。

第十章，“现代 DevOps 的平台工程方法”探讨了平台工程如何通过创建集成的、自助式平台来解决开发者认知负荷危机，这些平台提供“铺设好的道路”和标准化模板，使组织能够在平衡开发者生产力与治理要求的同时，将平台视为产品，将开发者视为其客户 —— 所有这些都通过一个金融服务组织的实际案例进行了说明，该组织仅用 6 名平台工程师就服务了 1400 名开发者，从而变革了其交付能力。

O’ Reilly 在线学习

40 多年来，O’ Reilly Media 一直致力于提供技术和商业培训、知识和洞察，帮助企业取得成功。

我们独特的专家和创新者网络通过书籍、文章和在线学习平台分享他们的知识和专业经验。O’ Reilly 的在线学习平台为您提供按需访问的直播培训课程、深入学习路径、交互式编码环境，以及来自 O’ Reilly 和 200 多家其他出版商的海量文本和视频内容。欲了解更多信息，请访问 *https://oreilly.com*。

联系我们

关于本书的评论和问题，请联系出版商：

O’ Reilly Media, Inc. 141 Stony Circle, Suite 195 Santa Rosa, CA 95401
800-889-8969 (美国或加拿大境内) 707-827-7019 (国际或本地) 707-829-0104 (传真)
support@oreilly.com *https://oreilly.com/about/contact.html*

我们为本书设有一个网页，其中列出了勘误、示例和任何补充信息。您可以在此页面访问：*https://oreil.ly/ai-native-software-delivery*。

获取有关我们书籍和课程的新闻与信息，请访问 *https://oreilly.com*。

在领英上关注我们：*https://linkedin.com/company/oreilly-media*。

在 YouTube 上观看我们：*https://youtube.com/oreillymedia*。

致谢

我们对许多个人深表感谢，他们的专业知识、鼓励和坚定不移的支持将本书从一个大胆的想法变成了现实。

特别感谢我们优秀的同事们提供了专业知识：Matthew Schillerstrom、David Karow、Mridhula Venkat、Dan Gordan、Sean Roth、Harold Bell 和 Patrick Wolf。

我们还要感谢本书的技术审阅者 —— Charles Humble、Julian Setiawan、Sagar Gandhi 和 Laura Uzcategui —— 对本书技术方面提供的宝贵反馈。

还要感谢 O’ Reilly 团队在整个撰写过程中给予我们的支持：采购编辑 Louise Corrigan、开发编辑 Jeff Bleiel、制作编辑 Elizabeth Faerm 和执行编辑 Lisa LaRew。

也要感谢 Kristy Saunders。

第 1 章：通往 AI 原生 DevOps 之路

大多数软件开发团队都能讲述部署失败的惨痛经历。正是这些故事，促使我们走上了现代化交付实践的道路。

这里有一个例子：经过数周或数月的特性开发、大量的重构以及漫长的测试和稳定阶段后，一个团队准备进行部署。开发人员、运维团队成员、一群经理，甚至可能还有一些高管，聚集在“作战室”中。在此之前，开发和运维之间的协作微乎其微。然而，现在这两个团队作为一支统一的团队协同工作。他们开始逐一核对一份冗长的手动步骤清单或操作手册。

然而，即使详尽的清单也不能保证部署万无一失。鉴于本次发布中改动的数量，部署可能复杂且风险高。正如我们将在接下来的章节中看到的，依赖管理充满挑战，“依赖地狱”可能真实存在。因此，团队可能会发现生产环境中缺少了一个关键依赖项。团队可能会发现安装了不兼容的库版本，或者某个关键设置配置错误，或者迁移步骤失败或被遗忘，或者更改导致对合作服务的请求失败。

任何数量的失误都可能使本已复杂的部署脱轨。紧张气氛会加剧，救火行动随即展开，时间一分一秒地过去。团队希望在部署窗口内完成部署和随后的手动冒烟测试。如果部署不可挽回地失败且无法挽救，团队则希望回滚到上一版本不会导致意外困难，从而延长停机时间和复杂性。当部署最终完成时，筋疲力尽的团队才得以撤离。通常，团队需要在流量恢复后的“关键护理期”内保持警惕。随后可能会有几天或几周的稳定期，在此期间，开发团队可能会暂停所有特性开发，专注于热修复或补丁。

正如这个故事所示，高强度、高风险的部署对开发和运维团队都造成了巨大的消耗。这些大型生产部署，以及随后的稳定化工作周期，使团队无法继续构建增加业务价值的功能。

相比之下，现代软件交付简化并加速了将软件从开发人员的计算机交付给最终用户的整个过程。部署频繁、低风险、低戏剧性，且高度自动化。但我们正在进入一个超越自动化的新时代。下一个前沿是 AI 原生软件交付。

AI 原生交付将 AI 融入软件交付生命周期的每一个层面，使智能代理能够做出决策、优化工作流程并实时适应。这些代理 —— 从代码和 DevOps 到安全和测试 —— 自主协作、

强制合规、自我修复基础设施，并利用强化学习持续优化软件交付管道。这一转变标志着从被动治理到主动治理、从孤立工具到统一生态系统、从静态自动化到动态自治的转变。

随着 AI 生成代码、编排管道并减少手动繁琐工作，开发速度加快。系统变得更具弹性和安全性，AI 能先发制人地识别问题并自主解决。同时，通过智能优化，云成本得以降低，协作规模得以扩大，因为 AI 驱动的代理以机器般的速度处理跨团队协调和决策。

在本章中，我们将描述过去 25 年软件交付是如何演进的。我们将定义 DevOps 并描述 DevOps 实践如何实现现代软件交付。我们将探讨 DevOps 现状面临的诸多挑战。最后，本章将概述现代软件交付、DevOps 实践和 AI 原生方法如何演进以应对这些挑战。

开发 + 运维 = DevOps

“DevOps”这个术语通常归功于 Patrick Debois，他在 2009 年将“开发”（development）和“运维”（operations）这两个词组合起来，命名了他组织的一次会议，旨在探讨如何打破开发和运维团队之间的传统壁垒，以更快地交付软件。造成这些壁垒的两个主要因素是：

> **沟通和协作不畅**
>
> 开发人员通常专注于编写代码和功能，然后将完成的产品基本上“扔过一道墙”给运维团队。运维团队随后承担在生产环境中部署、维护和故障排除代码的责任。

> **优先级冲突**
>
> 开发团队优先考虑快速开发和新功能的快速发布，而运维团队则专注于系统稳定性、安全性和防止停机。尽管他们的优先级不同，但这些团队本质上是相互关联和相互依赖的。无论你的代码或基础设施多么令人印象深刻，除非它被部署并在生产中运行以服务于你的业务目标，否则它没有真正的价值。

这种目标不匹配，有时被称为“核心慢性冲突”，可能在问题出现时导致摩擦和相互指责。

作为回应，DevOps 原则鼓励在每个阶段进行沟通。它们鼓励运维在开发早期参与，并在代码部署后长期与开发人员保持持续合作。

DevOps 简史

日益复杂的软件团队、新的软件方法论和新工具为 DevOps 铺平了道路。在本节中，我们将探讨这些因素。

千禧年后的敏捷

21 世纪初，组织对如何提高软件交付效率的新思路产生了浓厚兴趣并乐于接受。基于精益制造思想的新“敏捷”方法论开始流行。这些方法论反对强调大量前期规划和严格线性独立阶段序列的“瀑布”式软件交付模式。相比之下，敏捷提倡短开发周期和频繁发布，以高度响应变化。许多并行努力将新的敏捷实践标准化。1995 年的一篇论文将 Scrum 实践形式化。Kent Beck 在他的 1999 年著作《极限编程解析》（Addison-Wesley）中描述了一套用于软件开发的敏捷实践。2001 年，Beck 和其他有影响力的敏捷流程倡导者在《敏捷宣言》中阐述了类似的主题，该宣言提倡适应性和响应性而非僵化地遵守计划。DevOps 借用了宣言第一条原则中的“持续交付”名称：“我们的最高优先级是通过早期和持续交付有价值的软件来满足客户。”

Jeffrey Fredrick 观察到，Ken Schwaber 的 Scrum 系列书籍从 2001 年到 2007 年的进展，可以作为衡量敏捷成熟度及其组织影响力的一个晴雨表。在此期间，Scrum 凭借其清晰的结构、规范的角色以及跨团队适应性，迅速成为主导的敏捷实践。2001 年，Agile Software Development with Scrum (Pearson) 向刚开始探索敏捷方法的开发人员和小型团队介绍了这个框架。到 2004 年，Agile Project Management with Scrum (Addison-Wesley) 解决了实际实施挑战，标志着敏捷在更广阔的 IT 领域中得到日益普及。到 2007 年，The Enterprise and Scrum (Microsoft Press) 承认了将敏捷实践从单个团队扩展到整个组织的需求日益增长。这些书籍反映并帮助塑造了敏捷从边缘理念到企业必然趋势的历程。

持续集成与持续交付

在接下来的十年里，技术组织日益受到敏捷思维的影响。一个结果是采用了持续集成和持续交付 (CI/CD) 实践。

《敏捷软件开发宣言》催生了持续集成实践，它实现了敏捷的一个关键宗旨：频繁交付可工作的软件。开发人员将其代码更改合并到共享仓库中。通过持续集成，每次合并都会触发自动构建和测试过程。这个自动化系统能够迅速发现错误和冲突，让团队在开发周期的早期进行修复。持续集成鼓励更小、更频繁的更新，从而实现更快的交付、减少集成问题并拥有更健康的 codebase。

持续交付是持续集成的自然延伸。CD 自动化了将已通过集成构建和测试的代码准备好发布到生产环境的过程。这包括打包、配置和将软件部署到预发布环境等步骤。CD 使团队能够快速可靠地推送新功能、错误修复和更新，确保可部署的软件始终可用。

在每个开发周期结束时交付“潜在可发布产品”是另一个关键的敏捷实践。潜在可发布仅仅意味着可靠、经过测试、已打包且可以部署到生产环境的软件。（实际上，许多采用 CD 的组织只在内部进行交付，并继续不频繁地部署到生产环境。持续交付并不等同于持续部署。）

早期 DevOps 的里程碑

敏捷方法论倾向于关注软件交付生命周期的规划和执行部分，而早期 DevOps 则侧重于交付和运维。在 DevOps 兴起之后的几年里，这场运动获得了显著的势头。一个关键的里程碑发生在 2009 年，首届 DevOpsDays 大会举行。这次活动汇集了专业人士，分享他们在 DevOps 实践方面的经验和见解。

另一个重要的发展是 2010 年 Gene Kim、Kevin Behr 和 George Spafford 合著的《凤凰项目》（IT Revolution Press）出版。这部小说描述了一个虚构 IT 组织所面临的挑战，以及 DevOps 原则和实践的采用如何使其性能发生了戏剧性的转变。它以一种技术和非技术受众都能产生共鸣的方式阐述了 DevOps 的案例。第二年，另一部有影响力的出版物《DevOps 手册》由 Gene Kim、Jez Humble、Patrick Debois 和 John Willis 发布。这本实用指南通过提供一个实施 DevOps 的综合框架，帮助许多组织开始了他们的 DevOps 之旅。

2013 年，Kim 和 Humble 发布的初始 Puppet Labs（现为 Puppet）“DevOps 现状”报告引起了关注。该报告不仅关注技术指标；它强调了 DevOps 采用的业务效益，表明实施该方法的组织可以比同行快 30 倍地发布代码，同时失败率降低 50%。这直接将 DevOps 实践与领导者关心的业务成果联系起来。Nicole Forsgren、Jez Humble 和 Gene Kim 合著的《加速：精益软件与 DevOps 科学》（IT Revolution）一书更详细地探讨了这一主题。

2013 年平台即服务（PaaS）和 Docker 的引入标志着另一个关键时刻，因为这些技术简化了应用程序的部署和管理，使得 DevOps 实践可以在更大规模上实现。在此之前，基础设施和应用程序管理的复杂性使得 DevOps 的广泛采用充满挑战。2014 年 AWS Lambda 的推出通过开创大规模事件驱动函数执行进一步改变了格局，允许开发人员专注于编写代码而无需担心底层基础设施。同时，同样于 2014 年引入的 Kubernetes 提供了一个强大的框架，用于大规模编排容器化应用程序，确保部署的可靠性、高效性和可扩展性。

到了本世纪后半叶，机器学习 (ML) 技术开始渗透到 DevOps 工具链中，尤其是在应用程序性能监控 (APM) 和测试领域。测试工具会利用 ML 来优化测试执行和检测用户界面的变化。同时，Datadog 和 New Relic 等 APM 工具很早就将自己品牌化为“AI Ops”，因为它们使用 ML 来识别问题信号。到 2018 年，Harness 将 ML 应用于持续交付，以检测问题信号，使系统能够识别部署何时导致问题并触发必要的回滚。这些技术共同为现代 DevOps 奠定了基础，通过提供必要的工具和框架来高效管理复杂的软件系统，为 AI 原生 DevOps 铺平了道路。

DevOps 1.0

DevOps 已从一个松散的、小众的概念发展成为一套完善的思想，我们可以称之为“DevOps 1.0”。其属性包括：

文化转型

认识到文化转变对于协调软件开发和运维团队的重要性

自动化实践

实施持续集成和持续交付等实践，以简化软件交付

自动化工具

利用特定工具自动化软件交付管道的各个阶段，包括代码提交、测试、部署、供应和生产监控

DevOps 1.0 实践的早期采用者获得了立竿见影的成功。在 2010 年代初，许多工程团队每季度发布一次软件，投入数周精力进行手动测试、协调和生产部署。这些发布流程缓慢、容易出错，并需要下班时间安排以最大限度地降低风险。随着组织开始采用早期 DevOps 原则 —— 使开发和运维团队更紧密地协作并自动化交付管道的关键部分 —— 他们实现了更快的发布周期、更高的可靠性和更少的手动工作。对于许多组织来说，这种转变使他们能够从季度发布转向双周甚至每周发布，为更迭代的开发和更快的价值实现奠定了基础。

DevOps 1.0 的挑战

DevOps 1.0 提供了有价值的概念、实践和工具。然而，由于以下原因，今天的公司在充分实现 DevOps 的效益方面面临新的挑战：

- 软件趋势引入了需要 DevOps 适应的复杂性
- DevOps 1.0 工具集要么功能不足，要么对于许多组织来说变得过于复杂

以下章节将详细探讨这些挑战。

云原生和微服务架构的采用。新的架构模式涉及数十个部署到独立容器的离散微服务。DevOps 1.0 管道无法满足这些新架构的要求。

在过去的十年中，微服务和云原生架构已成为现代软件开发的实际标准，其驱动力是软件系统对更高可扩展性、灵活性和敏捷性的需求。这些架构为 DevOps 团队带来了重大的新要求。微服务的采用导致要部署的服务激增，每个服务都有自己的依赖项和配置。协调部署和在这些分布式服务中保持一致性变得越来越具有挑战性。

容器（云原生系统的一个关键特性）和无服务器架构的使用需要新的部署和管理策略，并增加了另一层复杂性。DevOps 团队现在必须处理跨数十甚至数百个瞬时容器或无服务器函数的部署，这需要强大的编排工具、用于构建和管理容器生命周期的自动化流程，以及对这些新兴技术的深入理解。自动化容器的整个生命周期 —— 从构建镜像、推送到仓库，到以最小停机时间推出更新 —— 对于高效的容器管理至关重要。

开源软件的兴起。开源软件（OSS）已成为现代软件开发中无处不在的一部分。虽然 OSS 提供了许多好处，但它为 DevOps 团队带来了新的挑战。管理依赖项、确保与不同版本的兼容性以及维护跨多个 OSS 组件的安全补丁可能是一项艰巨的任务。此外，团队必须仔细审查代码，并确保其符合组织的安全性合规标准。

数字化体验和企业消费化的重要性。在这个数字化颠覆的时代，Marc Andreessen 关于软件正在吞噬世界的预言被证明越来越准确。公司提供的数字化体验正成为客户的主要接触点，塑造着他们体验品牌的方式。此外，企业技术的消费化意味着员工期望获得与面向客户的应用程序相同的无缝体验和持续更新。这些期望给 DevOps 团队带来了压力，要求他们提供更频繁的发布、保持高可用性并支持实验以推动快速创新。

DevOps 1.0 工具集力不从心

自 2009 年首届 DevOpsDays 大会以来，我们对工具的需求已经发生了变化。交付节奏加快，而监管负担却增加了。以制品仓库为例：它们最初是作为本地缓存引入以加速构建的，现在它们对于在多种语言中保护软件供应链至关重要。为了简化部署，我们进行了容器化，而我们的构建时间变得更长，使得我们的持续集成构建不再那么“持续”。我们从一套配置管理工具转向了更新的、云原生的声明式工具。

但我们仍然需要测试、保护和管理那些基础设施的更改。与此同时，新工具层出不时 —— 每个都承诺改进，但也需要与所有其他工具连接起来。对于许多团队来说，当前的堆栈正在崩溃。

管道很快变得非常复杂。组织平均管理着 10 种或更多不同的工具来部署软件。例如，

一个部署 Rails、Sidekiq 和 NodeJS 应用的自动化管道可能包含以下工具：

- GitHub actions 用于运行 CI
- 用于检测 Sidekiq、Rails 和 Puma 并将应用程序指标推送到 Prometheus 的库
- Docker 镜像构建和 Kubernetes
- Artifactory 用于存储镜像和 Helm chart
- ArgoCD 用于 Kubernetes 上的 GitOps 部署
- Helm 用于管理部署和升级
- Terraform 用于管理 Amazon Web Services (AWS) 基础设施、角色、权限等
- New Relic 用于异常捕获和监控
- Kube-state-metrics 用于收集容器指标
- Prometheus 用于存储指标
- Grafana 用于使 Prometheus 指标易于消费

这个工具集的集成和管理对于资源有限的团队来说可能构成相当大的挑战。让我们来看看 DIY 方法的一些挑战。

广泛使用的开源工具通常不是最优的。DevOps 的 DIY 方法通常会导致效率较低的管道。一些开源工具缺乏能够减少开发人员工作量并缩短生产时间的特性。例如，维护 Jenkins 的正常运行时间和扩展性需要大量资源。漫长的测试时间可能导致构建缓慢。最后，重用管道的模式是复制/粘贴，导致“管道蔓延”，这可能难以维护且成本高昂。第三章将更详细地讨论这些问题。

DIY 管道导致重复和浪费的工作。团队通常必须实现“管道”来将工具和系统连接起来。这导致大量的重复造轮子。例如，Jenkins 和 ArgoCD 是常用的 CI/CD 工具。这些工具为自动化软件开发和部署过程提供了强大的功能，但它们需要团队从头开始构建基本结构，如基于角色的访问控制（RBAC）、审计日志和通知。

实施和维护这是一项本可以用于为客户创造价值的工作。

自动化通常不完整，需要手动步骤。一个团队在大部分部署过程中使用自动化脚本，但需要手动干预来配置环境变量，这可能导致部署不一致，如果不是所有团队成员都遵循相同的程序的话。不完整的自动化可能导致监控和反馈回路的空白，因为手动步骤可能不会触发自动化警报或指标收集。手动步骤引入了人为错误的风险，这可能导致停机或安全漏洞。因此，DevOps 中不完整的自动化可能导致效率低下、错误和可伸缩性问题。

治理是事后才考虑的。如果没有前期治理，团队可能会忽视合规要求（例如符合通用数

据保护条例 [GDPR] 标准），导致问题后期被发现时，需要昂贵的返工或面临罚款。如果安全措施不一致或作为事后诸葛来应用，应用程序将容易受到攻击。如果没有明确的治理政策，云服务或基础设施等资源可能会被过度配置或未充分利用，导致成本浪费（我们将在第九章中讨论此话题）。如果缺乏监督，团队可能会使用不同的工具、流程和标准，导致集成挑战和效率低下。

DevOps 2.0

DevOps 1.0 显著加速了许多公司的软件交付过程。然而，它的复杂性、遗留的空白以及所需的投入，都为改进留下了空间。这就是我们称之为 DevOps 2.0 的愿景 —— 它由更简单的开发人员体验、具有易于管理所有运动部件的端到端自动化，以及增强整个管道的 AI 能力所定义。这一演进将重点从工具和流程转移到它们所服务的人员和成果。

DevOps 2.0 的流程和工具以强大的新功能增强了开发人员体验。开发人员通过自动化开发和交付工具链的设置，在几分钟内即可启动新项目和服务。开箱即用的集成使团队能够轻松启动并连接仓库、敏捷项目和管道。为了进一步简化流程，模板封装了组织的最佳实践，确保一致性并消除创建新服务时的工作管理开销。团队专注于构建应用程序，而不是繁琐的基础设施设置。AI 代理执行日益复杂的 DevOps 任务，例如自动诊断和解决基础设施和管道问题，优化资源分配，并根据观察到的性能模式提出架构改进建议。

DevOps 2.0 工具通过更具凝聚力、紧密集成的工具集，解开 DevOps 1.0 解决方案的复杂性。基本结构（RBAC、审计日志）被集成。内置了对各种部署策略和实验方法的支持，实现了团队所需的频繁发布和快速迭代。新工具可扩展以支持跨多个环境的大规模部署，包括本地、云和混合设置。DevOps 2.0 工具将提供安全的管道和策略强制执行，以最大限度地降低开源采用和 AI 生成代码固有的风险。

最后，AI 正被融入到 DevOps 2.0 的工具和流程中，贯穿整个软件交付管道。Agent 控制协议（ACP）、模型上下文协议（MCP）和 Agent-to-Agent 协议等新兴协议正在帮助实现 AI 模型与更广泛的工具、系统和数据生态系统之间的无缝交互。这些协议定义了 AI 代理与工具交互、安全访问数据并在护栏内执行任务的标准化方式 —— 从而实现更动态和自主的工作流程。

在现代 DevOps 环境中，这些协议充当了 AI 能力与操作基础设施之间的桥梁，使 AI 不仅仅是观察和建议；它们赋予 AI 采取有意义行动的能力，同时保持可审计性和合规性。随着 DevOps 2.0 拥抱日益智能的自动化，这些协议为安全、可扩展、高效的 AI 驱动操作提供了基础，从而极大地提升了开发人员的工作流程。想象一下，工具可以生成代码、注释、测试和基础设施脚本，或者使用自然语言搜索提取相关代码片段。此外，ML 通过仅执行相关测试来加速测试周期。

利用 AI，这些工具提供个性化的入职指导，检测漏洞并提供修复建议或启动修复，甚至帮助编写和理解策略。通过分析可观测性遥测数据来识别何时需要回滚，从而提高了部署的可靠性。AI 分析功能实验以了解变更的影响。这种 AI 驱动的软件开发生命周期（SDLC）转型正在提高生产力、改善质量、降低风险，并增强整体开发人员体验。

随着开发人员能够借助 AI 编码助手越来越快地编写代码，企业快速安全地将更改交付到生产环境并了解这些更改是否有利的能力，将成为创新的限制因素。要做好这一点，既需要做好 DevOps 的基础工作，也需要在交付的每个阶段融入前沿 AI。

总结

现代软件交付强调快速发布、无缝体验和持续创新，这推动了传统 DevOps 实践的变革需求。尽管 DevOps 1.0 通过 CI/CD 和初步的跨团队协作奠定了基础，但其对由不同解决方案构建的复杂工具链的依赖造成了障碍。这些挑战源于应用程序（微服务、容器）日益增长的架构复杂性、开源组件的激增以及管理日益多样化工具集的需求。DevOps 2.0 旨在通过简化开发人员体验、提供更集成和智能的工具集，并将 AI 原生融入到管道中来解决这些问题。这种演进有望带来更高的效率、增强的质量，并将重点放在交付价值而非仅仅管理工具上。

此外，AI 原生软件交付用自主代理（例如代码、DevOps、安全）取代静态自动化，以实现自我优化系统和主动统一的生态系统。它通过自主代码生成、上下文管道创建、预测性故障解决和实时决策，加速开发速度、增强可靠性、确保合规性、降低成本并促进可伸缩的协作。尽管这具有变革性，但组织必须解决 AI 治理、数据隐私和技能差距，才能充分利用其优势。

在第二章、第三章和第四章中，我们将涵盖 DevOps 自动化的骨干。这包括用于有效版本控制的源代码管理、使用持续集成进行高效开发的构建和测试，以及使用持续交付系统在内部进行部署以实现无缝软件发布。我们将探讨 DevOps 1.0 的方法以及 DevOps 2.0 带来的机遇。

第 2 章：源代码管理

想象一下这样一个场景：你和你的团队正在协作一个复杂的软件项目。多位成员贡献智慧，进行修订和增强。如果没有清晰的系统来管理变更，你们可能会覆盖彼此的工作，并且无法追踪谁在何时做了什么更改，以及为何进行这些更改。如果没有清晰的系统来标记一系列变更，一旦出现问题，你将无法回溯到团队代码的某个稳定旧版本。如果没有明确的工作流和结构化的访问控制，任何人都可以随时更改任何内容，且缺乏监督。如果没有这些控制，当你需要重新构建某个特定版本时，团队将无法确定哪些代码文件被用于构建该版本。

接下来，再想象一下，几个团队已在一个新应用程序上工作了数月，现在即将部署到生产环境。各种开发和质量保证（QA）环境都进行了临时性修复和调整，但这些更改并未可靠地反映在生产环境中。重要的生产设置未在 QA 环境中重复，并且开发环境之间差异巨大。鉴于所需环境日益复杂，搭建新环境已成为一个耗时且容易出错的瓶颈，这会带来沮丧和延迟。

这些情况都是导致功能失调和精力浪费的根源。源代码管理（SCM）实践正是为了解决这些问题而创建的。SCM 的核心在于跟踪和管理代码以及配置等关键资源随时间推移所发生的变化。

如今，人工智能（AI）正在改变我们处理 SCM 的方式。AI 可以自动检测高风险更改、建议代码或配置改进，甚至通过理解修改背后的意图来帮助解决合并冲突。它能够识别跨环境的不一致性、推荐修正方案并优化部署工作流。AI 驱动的工具不仅帮助团队管理复杂性，还在赋能更快速、更安全、更具韧性的开发周期。随着软件交付变得更加分布式和动态，AI 正在成为使 SCM 更智能、更主动、更高效的重要伙伴。

源代码管理简介

在团队内部协调变更的问题可以追溯到编程的早期，SCM 实践的历史与计算机编程的演进紧密相连。在本节中，我们将探讨 SCM 如何演变以及 AI 工具在现代 SCM 中扮演的关键角色。

源代码管理简史

在编程早期，程序相对简单，受限于有限的硬件，代码管理也十分初级。随着 CPU 变得强大和复杂，计算和代码也变得更加复杂。代码仓库，即提供基本 SCM 功能的集中存储，最早出现在 20 世纪 70 年代，与高级语言和结构化编程方法的兴起同步。像 Source Code Control System (SCCS) 这样的工具提供了基本的版本跟踪功能，允许开发人员回滚到以前的版本并查看变更历史。这些早期系统反映了向更有组织性的程序开发转变的趋势。

20 世纪 70 年代，随着更结构化的软件工程团队的出现，SCM 进一步发展。像 1982 年引入的 Revision Control System (RCS) 和 1986 年引入的 Concurrent Versions System (CVS) 等工具增加了对协作至关重要的功能，包括分支。这使得更复杂的项目管理和协作文化成为可能。

20 世纪 90 年代初，IBM Rational ClearCase 作为一种商业 SCM 解决方案出现。它强调强大的配置管理和流程定制，使其适用于复杂的软件开发环境。由 CollabNet 开发的 Subversion (SVN) 是另一个获得普及的集中式代码仓库。SVN 1.0 于 2004 年发布，旨在解决 CVS 的缺点并提供缺失的功能。

分布式版本控制与 Git

21 世纪初，敏捷方法论和开源的兴起对软件开发提出了新的要求。快速发布意味着团队需要对日益复杂的代码库拥有更大的灵活性和控制力。团队本身也发生了变化，变得更大且通常分散在不同地理位置。Git 由 Linux 内核的创建者 Linus Torvalds 于 2005 年创建。他需要一个强大而高效的系统来管理 Linux 项目的庞大代码库，而现有选项都无法满足要求。

版本控制系统 (VCS) 是跟踪文件随时间变化的核心技术，构成了任何 SCM 方法的基础。与大多数早期代码仓库不同，Git 是一个分布式 VCS。在集中式 VCS 中，每个人都从存储在中央服务器（仓库）中的代码库的单个副本进行工作。每个开发人员都有自己的本地副本（工作副本），他们可以修改它。当开发人员进行更改并提交时，这些更改会立即上传到中央仓库，使其对其他人可见。要查看其他人的最新更改，开发人员只需从中央仓库更新他们的本地副本即可。图 2-1 展示了一个集中式 VCS。

分布式系统采取了不同的方法。在这里，每个开发人员在自己的本地机器上都拥有代码库的完整副本（包括仓库和工作副本）。开发人员所做的更改只存在于他们的本地副本中，直到他们明确地与团队共享这些更改。这通过将更改“推送”到中央仓库来完成。同样，要查看其他开发人员所做的更新，用户需要从中央仓库下载（“fetch”）这些更改到他们的本地副本中。图 2-2 展示了一个 Git 分布式 VCS。

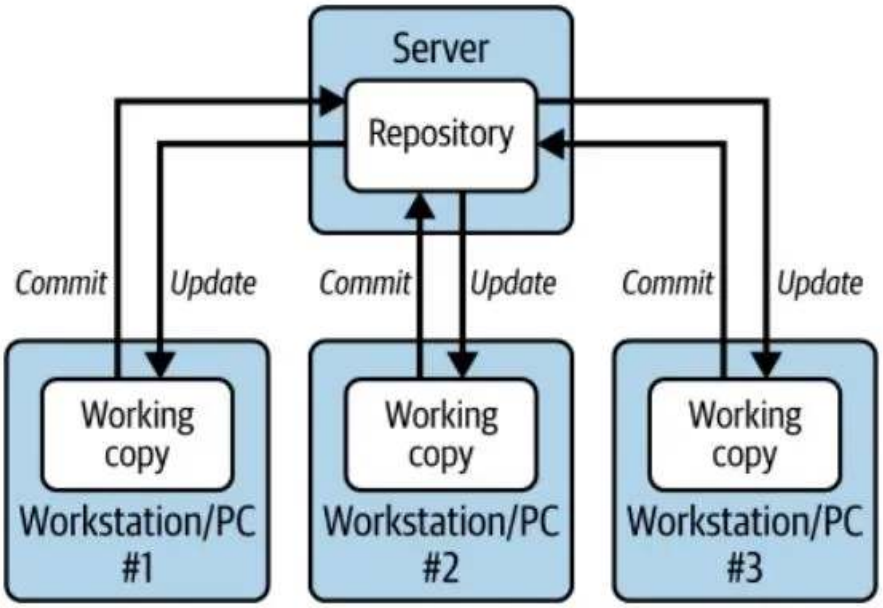

图 2-1. 集中式版本控制

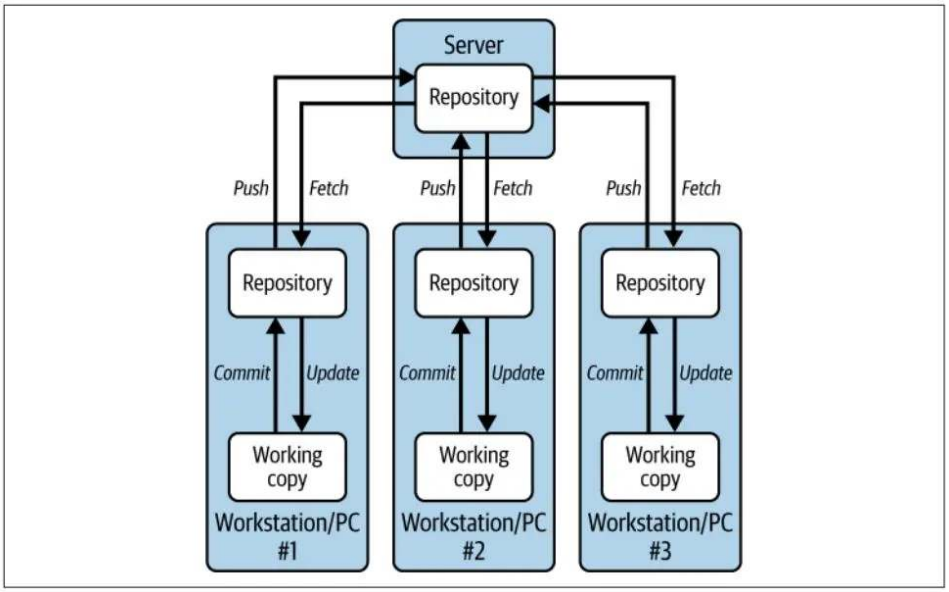

图 2-2. Git 分布式版本控制

Git 对速度的专注、其分布式特性以及强大的分支功能使其在许多方面成为颠覆性的创新：

分布式促进离线工作

Git 的去中心化方法促进了高效和独立的工作，因为开发人员可以在没有中央服务器的情况下在本地进行更改。这也使得开发人员能够离线工作。

灵活的分支与合并

Git 的分支系统非常灵活。开发人员可以创建隔离的分支来开发新功能或修复错误，而不会影响主代码库。将这些分支合并回主代码库是一个流畅高效的过程。这使得开发人员能够更自由地进行实验和迭代。

轻量且高效，适用于大型代码库

Git 擅长高效处理大型代码库。它只存储代码版本之间的差异，这使得它比传统的 SCM 系统更快，并且所需的存储空间更少。

非线性历史有助于组织

与某些强制执行线性历史的 SCM 系统不同，Git 允许开发人员通过诸如变基（rebasing）之类的功能来重写历史。这种灵活性有助于维护一个干净整洁的代码库。

首批广泛使用的 Git 托管仓库在几年后出现。GitHub，如今最受欢迎的平台，于 2008 年推出。这些平台建立在 Git 的强大功能之上，提供了用户友好的网页界面、代码库的云存储和协作功能。这种结合将 Git 从一个强大但技术性的工具转变为一个可访问且社交化的软件开发平台，使其成为现代软件开发工作流的基石。

尽管传统的集中式仓库仍有其历史足迹，并用于具有非常特定需求的环境中，但 Git 现在是主要选择。Stack Overflow 2022 年的一项调查发现，94% 的受访者使用了 Git，而使用任何源代码控制的人中有 98% 使用 Git。因此，我们将重点关注 Git 仓库的变体。

Git 分支扩展

2010 年，Gitflow 分支约定出现，它利用分支为开发、功能创建和发布准备提供了清晰的分离。图 2-3 展示了一个 Gitflow 工作流。

在 Gitflow 工作流中：

1. 主代码库位于名为“main”的分支上。此分支通常被认为是稳定的，并且只应包含生产就绪的代码。
2. 创建一个新的“develop”分支，它作为所有开发工作的持续集成（CI）分支。

3. 功能开发发生在从 develop 分支派生出的隔离分支（feature/release 分支）上。开发人员在这些功能分支上开发新功能和修复错误。一旦功能完成并经过彻底测试，它就会合并回 develop 分支。
4. develop 分支充当所有已完成功能的集成点。它代表即将发布的版本，并不断通过合并的功能分支进行更新。
5. 当需要发布时，会从“develop”创建一个发布分支。此分支上可以进行错误修复和微小调整。一旦最终确定，发布分支会合并回“main”以创建官方发布。同时在“main”中创建一个相应的标签来标记发布版本。

拉取请求（Pull Request，有时缩写为 PR）是 Git 版本控制中用于代码审查和集成的一项核心协作功能，并广泛用于 Gitflow 和其他分支模型。拉取请求为开发人员提供了一种结构化的方式来提议更改代码库，并在将其合并到主分支之前获得其他人的审查。

Gitflow 强调计划发布和独立的发布分支，这受到了新的 Git 分支模型的挑战。受持续集成和持续交付日益普及的推动，这些模型优先考虑更快的部署和更频繁的更新。主干开发（Trunk-based development）完全摒弃了专用开发分支的概念。相反，功能在经过严格测试后会持续直接集成到主分支（通常称为“trunk”或“main”）中。图 2-4 展示了这种模式。

这种简化的方法允许更快的反馈循环和更快的部署，与现代 DevOps 实践非常吻合。拉取请求在这些工作流中仍然至关重要，通过代码审查确保代码质量，然后再将更改合并到主分支。

GitOps 与源代码管理

我们已经看到代码仓库如何与编程和软件开发实践共同演进，解决了我们所设想的问题，使团队能够有效地协作源代码。但是部署问题呢？我们如何高效、系统地生产所需的环境，又如何简化代码到生产环境的部署？

这就是 GitOps 的用武之地。DevOps 将开发（Dev）和运维（Ops）结合起来，强调自动化在消除人为错误和确保环境一致性方面的重要性。这转化为更快的部署、更高的可靠性和更低的风险。GitOps 是指自动化配置基础设施的过程，尤其是在现代容器优先的云基础设施中。GitOps 强调使用代码仓库（通常是 Git）作为系统所需状态的单一事实来源，并利用自动化持续协调实际状态与所需状态。存储到我们仓库的资源可以包括：

基础设施配置

定义环境所需组件的文件、虚拟机（VM）的类型和数量、存储配置、网络设置和安全策略。这可以包括声明式和命令式配置以及部署脚本。

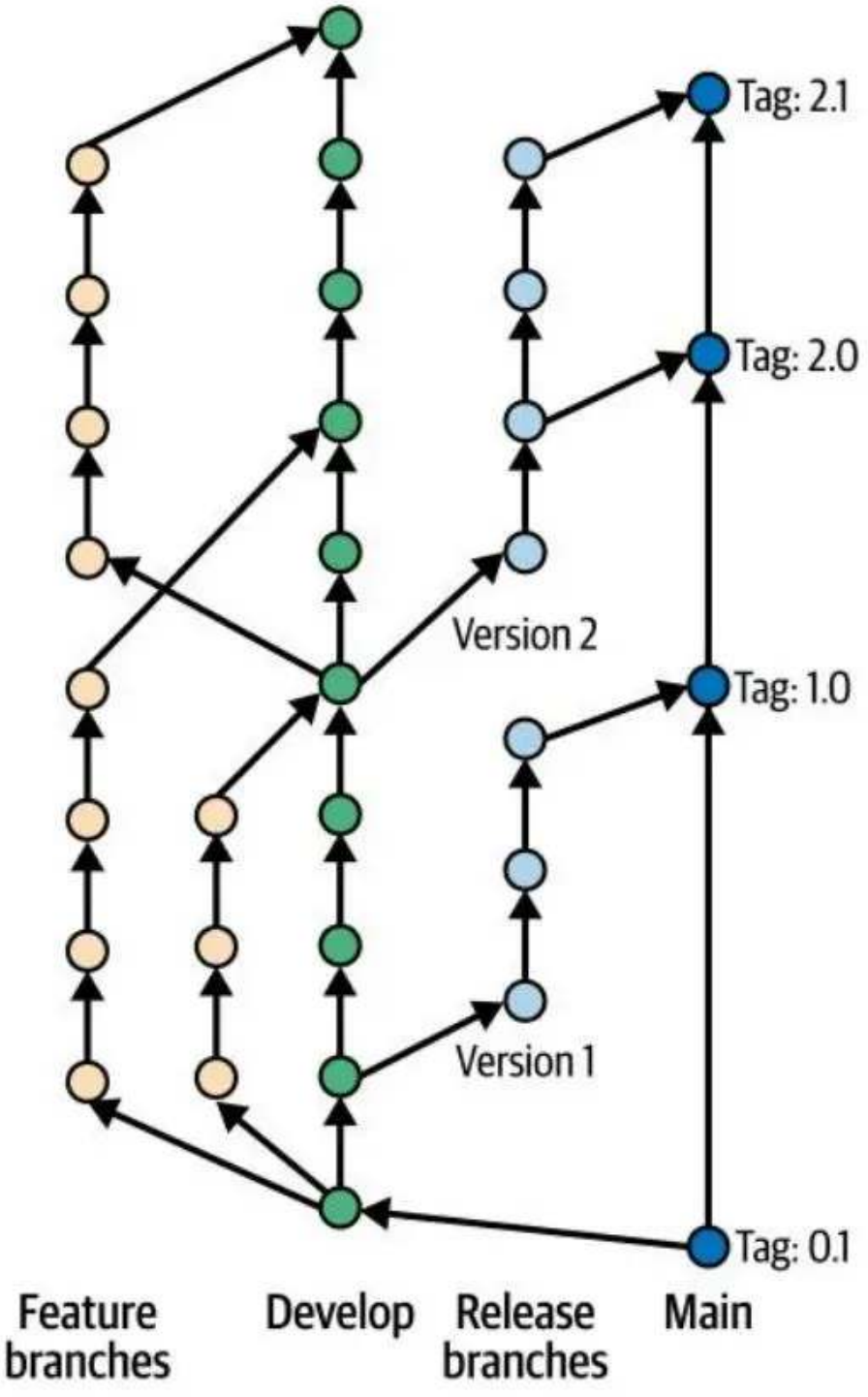

图 2-3. Gitflow 工作流

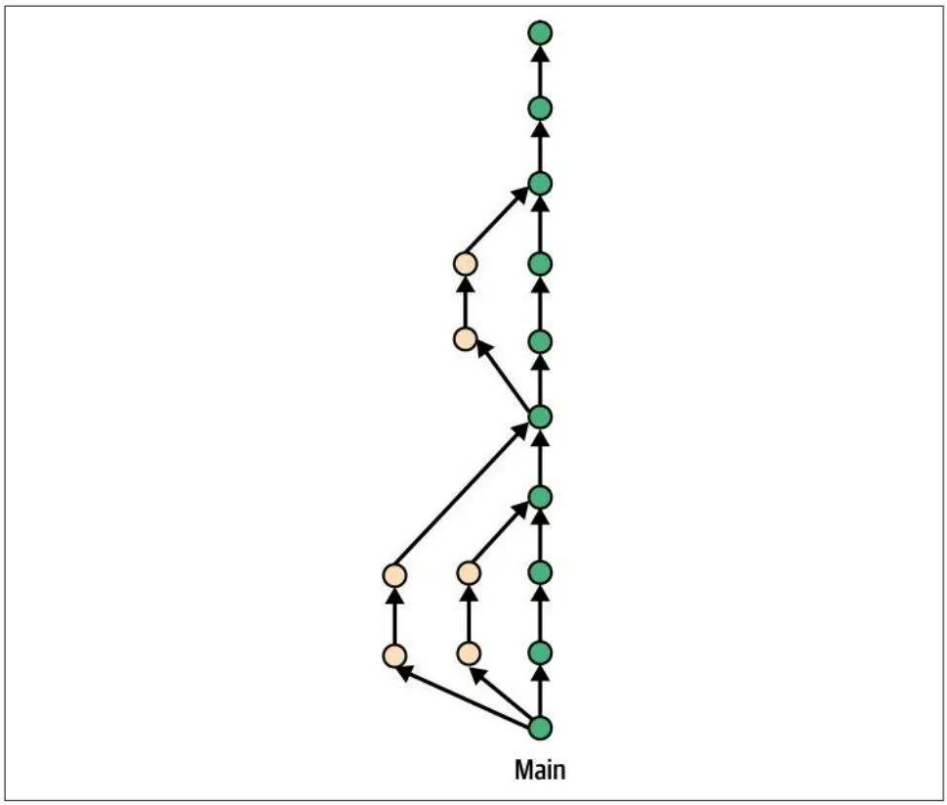

图 2-4. 主干开发

环境变量

这些对于存储不应直接嵌入代码中的敏感信息（如密码或 API 密钥）至关重要。基础设施即代码（IaC）工具通常具有安全管理和引用环境变量的机制。

额外资源

根据环境的复杂性，仓库还可能存储其他资源，例如用于应用程序部署的容器镜像（通过 git-lfs）。

利用我们的仓库作为单一事实来源，我们可以利用其强大的功能。我们获得详细的版本跟踪和变更历史记录，并且可以通过 Git 工作流来管理基础设施更新，这些工作流通过拉取请求（Pull Request）等方式促进协作和监督。管理良好的基础设施自动化意味着更快的部署、更少的错误以及每次需要创建新环境时都具有可靠的环境。我们将在第 4 章中了解更多关于使用 GitOps 进行部署的内容。

单体仓库与远程缓存

我们在第 1 章中提到了微服务的重要性。两种提高基于微服务系统生产力的关键实践是使用单体仓库（monorepos）和远程缓存。

单体仓库（monorepo）是一个单一版本控制的代码仓库，用于存储多个项目或服务的代码。在微服务背景下，这种方法简化了协作，简化了依赖管理，实现了跨服务的原子性更新，并减少了版本冲突。

远程缓存是指将构建产物（如编译后的代码或测试结果）存储在远程服务器上。像 Nx 这样的工具使用此技术显著加快开发工作流，允许团队重用以前生成的输出，而不是从头开始重建，从而减少冗余计算。

单体仓库和远程缓存共同支持更快、更高效的 CI/CD 流水线，并有助于提高整体系统性能。然而，随着项目规模的扩大，单体仓库可能会引入复杂性，并且如果未深思熟虑地实施，远程缓存可能会引发厂商锁定（vendor lock-in）的担忧。

AI 在源代码管理中的应用

AI 工具彻底改变了开发人员的编码方式。GitHub Copilot、Cursor、Harness AI Code Agent 等编码助手/代理充当智能结对程序员，根据项目上下文提供实时代码建议。这些工具可以预测并建议整个代码行或代码块，显著加快开发过程。

除了代码补全，AI 助手还可以：

- 自动生成样板代码结构
- 建议不同的实现方法
- 提供代码解释和文档
- 协助调试和优化

AI 原生软件交付始于 AI 原生 SCM。AI 与 SCM 的集成超越了代码补全。在 SCM 内部，AI 可以分析仓库模式，在代码到达生产环境之前识别潜在的错误，并根据在类似项目中观察到的最佳实践提出架构改进建议。这种前瞻性方法显著减少了技术债并从开发的早期阶段提高了代码质量。我们将在本章稍后探讨其中一些主题。

在接下来的几节中，我们将介绍 SCM 系统如何适应交付流水线。在此理解的基础上，我们将讨论在选择适合团队的 SCM 时需要考虑的因素。最后，我们将探讨现代代码仓库的特点，包括 AI 的作用，这些特点可以简化你的整个软件开发流水线。

源代码管理在交付流水线中的作用

核心代码仓库是交付流水线中的关键组成部分，它锚定了整个流水线过程。它作为代码的单一事实来源，确保一致性和可靠性，并且是开发人员持续交互、启动集成和交付活动的实体。

图 2-5 描绘了代码仓库与持续集成和交付的关系。

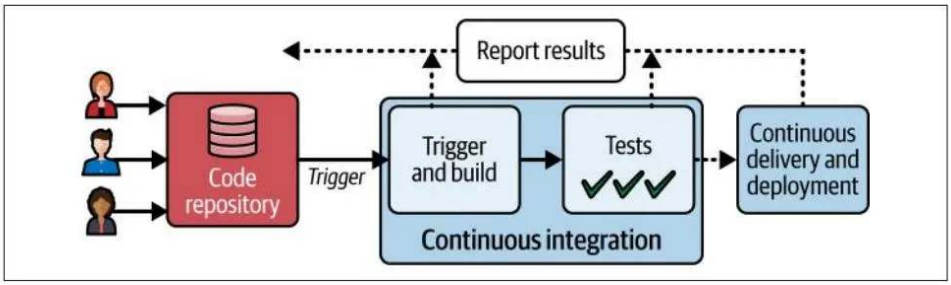

图 2-5. 开发人员对代码仓库的操作会触发 CI/CD 流水线

让我们来看看典型流水线的三个主要部分：

代码仓库

开发人员对代码仓库进行操作，提交更改并打开和关闭拉取请求。

持续集成

持续集成由代码仓库内的特定操作触发。这些触发器可以自定义，包括代码提交、拉取请求的打开或关闭，或团队根据其特定需求和实践确定的其他相关操作。CI 为开发人员提供关于代码更改的快速反馈。通过自动化构建和测试，CI 充当早期预警系统，提醒开发人员潜在的错误、集成问题，甚至代码风格违规。这种即时反馈使开发人员能够迅速解决问题，防止它们演变成更大、更昂贵的后续问题。有了 CI，你的代码库将保持在持续可部署的状态，为交付流水线中的下一步做好准备。

持续交付与部署

持续交付和部署步骤自动化了基础设施的配置以及新代码版本到一个或多个预生产环境的部署。通常会针对在预生产环境中运行的应用程序执行各种类型的测试。我们将在第 4 章中讨论这些步骤。最后，自动或手动决策会控制软件到生产环境的最终部署。我们将在第 8 章中详细讨论这些步骤。通过频繁部署更小的更改，CD 简化了交付过程，降低了发布风险，并增强了快速响应用户反馈的能力。

许多代码仓库内置了敏感信息检测功能。敏感信息可以包括以下内容：

API 密钥

用于验证和授权访问各种 Web 服务和 API 的唯一标识符

访问令牌

授予应用程序或资源特定访问权限的临时凭据

OAuth 令牌

用于委托授权的令牌，允许一个应用程序代表用户访问资源

私钥

在非对称加密中用于解密消息或验证数字签名的密钥

用户名和密码

用于系统和服务基本身份验证的凭据

数据库连接字符串

建立与数据库连接所需的详细信息，通常包括主机名、用户名和密码等敏感信息

云服务连接字符串

用于连接到 Azure Storage 或 AWS S3 等云服务的字符串，可能包含访问密钥和其他敏感信息

某些代码仓库会在开发人员尝试提交或合并包含检测到的敏感信息的代码时阻止或警告。CI 过程可以在敏感信息检测中发挥作用，阻止它们到达生产环境。理想的方法是同时利用两者以实现全面的安全性。

代码仓库考量

鉴于 SCM 对软件开发的重要性，选择代码仓库是团队将做出的首批决定之一。“我们将把源代码放在哪里?”是团队在启动项目之前就需要回答的问题。

首先，一个代码仓库必须支持对其团队至关重要的基本操作和开发人员工作流：

- 支持分布式离线工作的仓库创建、导入和克隆
- 分支、合并以及定义满足团队特定需求的分支规则（例如，限制特定用户创建/删除分支）
- 创建、审查和合并拉取请求，以及根据团队所需的治理定义拉取请求策略（例如，要求所有更改都与拉取请求相关联、禁止直接提交或设置所需审阅者批准的最小数量）
- 创建和修改标签，并定义标签策略（例如，强制标签名称遵守特定的模式，如语义化版本控制）

虽然实现细节可能有所不同，但这些都是预期的仓库功能。

在创建交付流水线时，团队通常首先选择仓库；因为这是一个可能对实现产生深远影响的选择，所以确保你的代码仓库能够无缝集成到更广泛的生态系统中至关重要。你的代码仓库应该在一个能够提高团队生产力而非增加工作量的生态系统中运作。此外，解决方案应该具有成本效益，并提供组织所需的透明度。

全面的集成

一个设计良好的 DevOps 生态系统的特点是易于使用的工具和与交付流水线所需的功能和服务的全面集成。这与零散的方法形成对比，在零散的方法中，开发人员需要手动集成许多不同的工具，这可能导致难以排查的问题和安全风险。它也与过于复杂的单一平台解决方案形成对比，这些解决方案通常存在功能冗余，难以配置。

配置即代码（configuration-as-code）是简化集成的一个例子。这种实践允许对交付流水线的更新直接在仓库中进行版本控制和跟踪，就像你的项目代码一样。你可以通过强制执行要求通过拉取请求和审批进行更改的工作流来进一步增强协作和治理，从而与标准开发实践保持一致。

另一个功能示例与安全/漏洞扫描有关。在拉取请求的上下文中显示检测到的漏洞和建议的修复有助于开发人员快速理解并解决任何检测到的问题。

AI 驱动的功能

过去几年，使用大型语言模型提高开发人员效率的编码助手或代理呈爆炸式增长。这些编码助手有助于代码自动补全、生成代码建议、理解代码功能以及许多其他用例。当 AI 助手与代码仓库集成以访问完整的代码库作为上下文 —— 而不仅仅是孤立的代码片段 —— 它们可以生成更准确和相关的建议。

MCP 在此发挥关键作用，它提供了一种通用的、标准化的方式来连接 AI 模型和代码助手与各种数据源，包括 Harness Code Repository、GitHub 和 Git 等仓库。这消除了对自定义集成的需求，减少了开发工作并提高了效率。

生成式 AI（GenAI）在代码仓库中的另一个强大应用是语义搜索 —— 使用自然语言搜索整个代码库的能力。像 Sourcegraph 的 Cody 和 Harness Code Repository 这样的工具使开发人员能够提出诸如“身份验证是如何实现的？代码在哪里?”之类的问题，而不是依赖于基于关键字的搜索，如“登录”或“验证”。此功能对于新团队成员的入职特别有价值，可以帮助他们快速理解复杂的代码库，而无需深入了解项目特定的术语。

在代码审查方面，像 DeepCode 和 Codacy 这样的工具使用机器学习算法来审查代码更改，自动检测潜在的错误、代码异味以及对编码标准的遵守情况，比手动审查更高效。AI 在 SCM 中的其他用例还包括：通过在代码提交之前自动扫描漏洞和合规性问题

来增强安全性，并推荐这些问题的修复方案；总结拉取请求；以及使用 SCM 作为数据源之一生成软件交付流水线。

需要注意的是，对于 AI 系统而言，结果在很大程度上取决于用于训练 AI 模型的数据。因此，例如，“好”的代码将带来好的代码建议和审查，而“坏”的代码将带来坏的代码建议和审查。

衡量 AI 的影响同样重要，以验证使用 AI 是否确实对开发人员产生了积极影响。Harness Software Engineering Insights 等工具可以帮助衡量使用不同编码助手的开发人员的生产力，并将其与不使用任何编码助手的开发人员进行比较。

AI 驱动的 SCM 通过生成快速可靠的代码（尤其是在训练良好的情况下）来加速上市时间，通过识别问题（包括安全漏洞）从源头提高代码质量，并通过提升代码审查的质量和效率来增强团队协作。

通过开源实现效率和透明度

你的 DevOps 工具是否开源是一个重要的考量。开源解决方案对于预算受限的组织来说可能具有成本效益，它们提供的透明度也具有优势。

专有解决方案通常可以声称提供可靠的正常运行时间和专门的客户支持团队来解决你遇到的任何技术问题。然而，企业用户通常需要支付订阅费，这对于小型团队来说可能是一个重要的成本因素。开源代码库可以免费使用，使其成为预算有限的团队的理想选择。开源性质允许透明度和社区驱动的开发。开发人员可以访问源代码，从而能够根据特定需求定制平台。但是，他们通常必须依靠社区进行故障排除和支持。虽然有价值，但开源可能无法提供与商业提供商相同水平的保证协助。此外，虽然开源促进了透明度，但也意味着潜在的漏洞是公开可见的。

像 Harness.io 和 GitLab 这样的开放核心（Open core）解决方案提供了一个中间地带。它们提供免费的、功能受限的版本，类似于开源。

最后，如果出于监管要求或为了确保业务连续性，可以将开源软件（OSS）托管（escrow）起来。这提供了保证，即在工具提供商倒闭的情况下，你仍然可以访问构建、测试和监控应用程序以及重新创建开发、测试和生产环境所需的工具。

平台化方法

传统的、零散的 DevOps 工具链常常造成数据孤岛，阻碍了对整个软件开发生命周期（SDLC）的可见性。然而，单一的 DevOps 平台提供了一个引人注目的解决方案，通过提供端到端的可见性。例如，它能够跟踪每一个变更，从代码仓库中的首次提交到生产

服务器上的最终部署。这种整体视图有助于识别瓶颈，在开发周期的早期发现潜在问题，并衡量 DevOps 实践的整体有效性。此外，全面的审计追踪提供了所有活动的清晰记录，简化了故障排除并确保符合安全法规。

统一平台还简化了治理并释放了智能自动化的潜力。在不同工具之间管理治理策略可能既繁琐又容易出错。单一平台允许你在整个开发流水线中一致地定义和执行策略。这确保代码符合编码标准、安全最佳实践和内部准则。例如，你可以通过实施以下策略来简化治理：在提交代码之前、在 CI 过程期间以及在 CD 过程期间，使用（你组织）批准的安全扫描器扫描代码。通过统一平台，这可以很容易地作为模板实现并重复使用。

此外，通过对部署上下文（包括基础设施和配置细节）的全面理解，该平台可以提供智能代码建议，优化性能和效率。想象一个由 AI 驱动的助手，它根据服务将如何部署来推荐代码调整，这可能节省开发时间并提高代码质量。

访问控制示例

当团队组建交付工具链时，通常会从单个点解决方案开始。然而，这种零散的方法可能导致显著的运维开销。在本节中，我们将以 RBAC 为例，看看一个内聚的交付流水线如何简化操作并赋能开发团队。

大多数协作工具都以某种形式使用基于角色的功能访问。代码仓库将支持内置角色，或包含内置角色并允许用户定义自定义角色。例如，GitHub 定义了“读取（Read）”、“分类（Triage）”、“写入（Write）”、“维护（Maintain）”和“管理（Admin）”等角色。这些角色对应于不同的访问级别；“读取”角色推荐给非代码贡献者，而“管理”角色则专为需要完全访问项目（包括敏感和破坏性操作）的用户设计。

这些系统使用 RBAC，这是一种管理系统中资源访问的方法，其核心围绕三个要素：用户、角色和权限：

- 用户代表需要访问的个人或账户。
- 角色是定义好的权限集合，授予对系统中特定资源或操作的访问权限。
- 权限是控制的基本单位，定义用户可以执行的操作（如读取、编辑或删除数据）。

用户不直接被分配权限。相反，他们被分配一个或多个角色。一旦用户被分配了角色，他们就继承了与该角色相关联的所有权限。这种方法通过消除为每个用户单独分配权限的需要来简化访问管理。相反，权限在角色级别定义，并根据用户被分配的角色授予访问权限。图 2-6 说明了用户如何被分配到角色，以及与角色相关联的权限集。

使用基于角色的访问是一种常见的模式，它通过强制执行最小权限原则（即仅授予用户执行其工作职能所需的权限）来减少管理工作并增强安全性。基于角色的访问还有助于

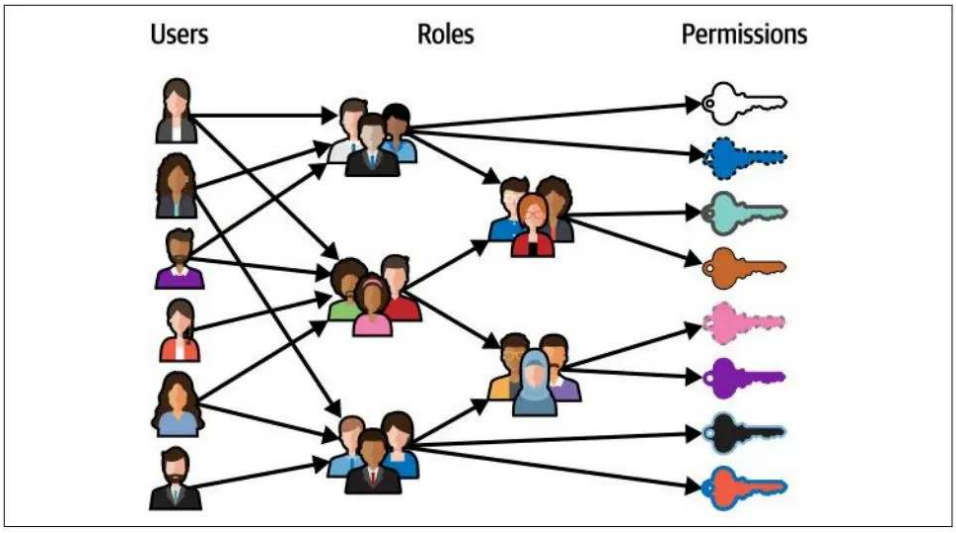

图 2-6. 用户被分配到角色；权限与角色相关联

合规性，因为它提供了谁可以访问系统中哪些内容的清晰文档。

定义角色，平台化方法

想象一个 DevOps 生态系统，它由 Git 仓库、Jenkins、用于管理 AWS 基础设施的 Terraform、用于配置管理的 Ansible 以及用于捕获性能指标的 Datadog 组成。在这样一个由多个不同工具构建的系统中，你可能会发现需要在每个系统中定义相似的角色，并重复添加这些角色。为新开发人员配置权限可能需要耗费时间的多个步骤。让我们看看一体化平台如何使用平台化方法处理 RBAC。

例如，Harness 平台具有三级层次结构。这三个级别（或范围）分别是账户（Account）、组织（Organization，简称 Org）和项目（Project）：

- 账户是最高级别的实体。它可以对整个平台进行控制并拥有可见性。
- 组织是一个控制单元，来自同一业务部门的人员和项目可以在一个独立的层次结构中进行组织。一个组织可以有多个项目。
- 项目是协作的基本单元，用户被分组在一起以处理相同的任务。

资源组（Resource Groups）是 RBAC 的一个组件，它定义了用户可以访问的对象。对象是任何 Harness 资源，包括项目、流水线、连接器、敏感信息、代理、环境、用户等等。当你将资源组分配给用户时，资源组中定义的访问权限将授予目标用户。资源组可以在任何范围定义。

角色同样可以在每个范围定义。角色与资源组一起应用，以创建完整的权限和访问集合。例如，你可以将“流水线执行者（Pipeline Executor）”角色分配给只允许访问特

定流水线而非项目中所有流水线的资源组。

总结

在本章中，我们介绍了 SCM，它是现代软件开发的基石。SCM 解决了团队协作和随时间管理代码库变更的挑战。它使团队能够有效地协作并管理代码变更。

SCM 对于 DevOps 和 CI/CD 工作流至关重要，随着 AI 原生 SCM 系统的出现，其作用正在扩大。这些智能系统可以生成、审查、分析和优化代码，改变团队编写和管理软件的方式。通过自动化日常任务、提高准确性并提供洞察力，AI 驱动的 SCM 系统加速了开发并提高了交付效率。

我们还讨论了选择正确代码仓库的重要性以及统一 DevOps 平台在实现内聚工作流和更强治理方面的优势。有了坚实的 SCM 基础，第 3 章将深入探讨持续集成如何自动化构建和单元测试，以确保代码质量和开发速度。

第 3 章：持续集成的构建和预部署测试步骤

简而言之，我们现代的软件交付实践提供了一种结构，帮助我们规划、编写、构建、测试和部署软件。在第二章中，我们探讨了 SCM 系统如何帮助跟踪和管理代码编写过程中的变更。

在本章中，我们将关注持续集成。图 3-1 展示了一个 CI/CD 流水线，我们将简要介绍，并在第四章和第八章中再次回顾。

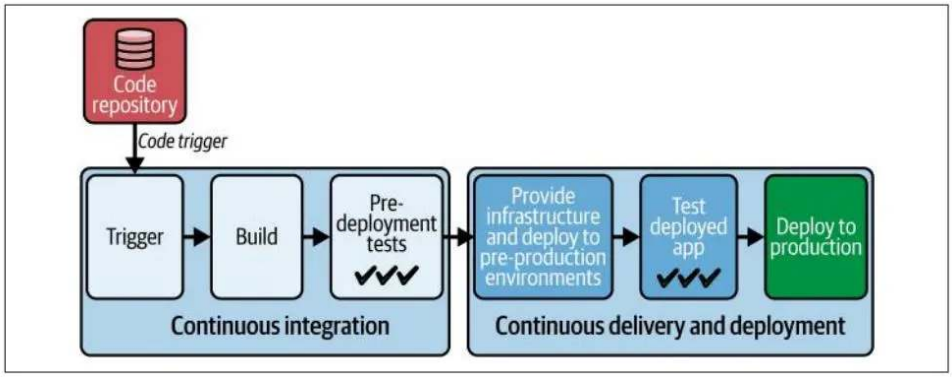

图 3-1. 一个 CI/CD 流水线

我们将深入探讨持续集成流水线，重点关注构建过程和预部署测试（静态扫描、单元测试和集成测试）。我们将演示 AI 原生方法如何通过生成式 AI (GenAI)、智能体 AI (Agentic AI) 和开放标准（如 MCP 实现）来加速 CI。这些技术在构建、缓存和测试阶段中实现了自动化流程、预测性优化、标准化上下文管理和智能测试策略。

除了关键的持续集成步骤外，我们还将回顾持续集成工具，并讨论选择工具时需要考虑的因素。您将深入了解如何提高构建流水线的效率、质量和安全性。

软件构建与测试简史

这是一个耳熟能详的故事。1947 年，在哈佛 Mark II 计算机上工作的一个工程师团队发现一只飞蛾卡在继电器中，导致机器出现故障。他们移走了飞蛾，并将其用胶带贴在日志本上，附注“首次发现实际错误案例”，从而巩固了“Bug”与软件错误的关联。早期软件开发中的测试，准确地描述了在机器中寻找 Bug。开发者独立编写代码并进行集成。测试通常是手动且临时的。当发现错误时，团队的重点是找出 Bug，清除机器中的“飞蛾”。Bug 通常在生产环境中发现，导致延误和不可靠的软件。

随着软件开发的演进，测试变得更加正式和严谨，重点放在尝试“破坏”软件以发现缺陷。正式的测试方法和标准也开始出现，例如 IEEE 829 软件和系统测试文档标准（1983 年）。

结构化软件开发与瀑布式方法

瀑布式方法引入了一种结构化的软件开发方法，其中测试成为一个独立的阶段。在需求收集阶段定义的验收标准，概述了软件必须满足的条件。然后，在开发结束时开发并执行测试用例以验证这些标准。缺陷会被记录并解决，直到软件满足所有要求。然而，这种正式的方法通常导致编码和测试之间存在相当大的延迟，使得早期问题检测和解决具有挑战性，最终导致新产品和新功能的上市时间变慢。

敏捷与测试驱动开发

在第一章中，我们讨论了敏捷方法在软件开发中的兴起，其动机是瀑布式开发的低效率和局限性。敏捷方法更为灵活和响应迅速的开发模型强调频繁反馈和迭代开发，这需要新的测试方法来跟上快速的开发周期。这导致了新的测试方法。

极限编程 (XP) 是由 Kent Beck、Ward Cunningham 和 Ron Jeffries 开发的一种特定敏捷方法，由一系列最佳实践定义。其中一个基本的 XP 实践是测试驱动开发 (TDD)。在 TDD 中，您在编写相关代码之前先编写测试。Beck 颇具影响力的著作《解析极限编程》（Addison-Wesley 出版），于 1999 年首次出版，将 TDD 推广给了广大受众，而 JUnit（用于 Java）和 NUnit（用于 .NET）等早期工具为开发者提供了框架，以便在编写相应代码之前轻松编写这些类型的测试。

在编写代码之前编写测试，鼓励开发者深入思考预期的代码行为，从而带来更好的设计和更少的缺陷。虽然这个概念以前就存在，但 TDD 先编写失败测试，然后编写代码使其通过的特定方法，与敏捷方法关注短周期和频繁交付可用软件的理念非常契合。这种实践重新定义了“完成”的概念：当代码可用时，功能并不算完成，而是当自动化测试

完成并通过时才算完成。

在 TDD 期间创建的自动化测试提供了一个安全网，让开发者能够放心地重构代码，因为他们知道任何回归问题都会被测试迅速捕获。这使得更快的迭代和更频繁的发布成为可能，进而允许从客户和利益相关者那里获得更快的反馈。测试本身也作为一种文档形式，清晰地阐明了系统的预期行为。

持续集成登场

正如我们在第一章中介绍的，CI 是将来自多个贡献者的代码变更集成到共享代码库中，并频繁触发自动化构建和测试，以确保软件保持可用状态的实践。这与 TDD 相辅相成。

CI 的根源可追溯到 1990 年代。Grady Booch 于 1991 年首次提出了“持续集成”这个术语，但真正将其付诸实践的是 Kent Beck 和 Ron Jeffries，他们在 1997 年的一个项目中合作时。他们的目标是解决代码合并不频繁导致的“集成地狱”问题，即冲突和错误会堆积如山，变得越来越难以解决。

早期的 CI 系统通常是定制化的，并针对特定项目进行调整。一个值得注意的例子是 CruiseControl，由 ThoughtWorks 于 2001 年创建。它是首批开源 CI 服务器之一，允许团队在每次代码提交时自动构建和测试软件。然而，它缺乏用户友好的界面和灵活的任务调度，导致 Kohsuke Kawaguchi 于 2005 年开发了 Hudson。Hudson 因其易用性和强大功能而迅速普及。

2011 年，与 Oracle 的一次争议导致 Hudson 分叉为 Jenkins，自那时起，Jenkins 已成为不仅用于持续集成，还用于持续交付和部署的最广泛使用的工具之一。Jenkins 的受欢迎程度可归因于其灵活性、可扩展性以及庞大的插件生态系统，使其能够与各种工具集成并适应不同的工作流程。

今日持续集成

持续集成已演变为现代软件开发中的一项基础实践，CI/CD 系统是任何交付流水线的支柱。通过代码变更的持续集成，团队已开始依赖以下优势：

减少集成问题

CI 通过确保开发者频繁合并代码变更来消除令人恐惧的“集成地狱”，最大程度地减少冲突，并使其更容易解决。

更快的反馈

CI 的自动化构建和测试过程为开发者提供了对其代码变更的快速反馈，使他们能够迅

速捕获和修复错误，从而维护稳定且可部署的代码库。

提高效率和可靠性

通过自动化构建和测试过程，CI 消除了人工错误和不一致性，从而实现更可靠和可预测的构建。

提高透明度

CI 仪表板和通知提供构建和测试状态的实时可见性，让团队中的每个人都能跟踪进度、识别潜在问题并更有效地协作。

加速发布

通过简化和自动化构建、测试和集成过程，CI 能够实现更快、更频繁的发布，使企业能够更迅速地响应客户反馈和市场变化。

在“CI/CD 流水线中的持续集成”中，我们将探讨 CI 在交付流水线中的功能，并探索 CI 工具的图景。

CI/CD 流水线中的持续集成

在第二章中，我们介绍了 CI/CD 流水线，重点关注代码库和代码集成之间的关系。让我们回到这个流水线，并关注持续集成，即构建步骤以及执行预部署测试类型（包括静态分析、单元测试和集成测试）的步骤。

图 3-2 中的流水线显示了一个典型的 CI 过程。

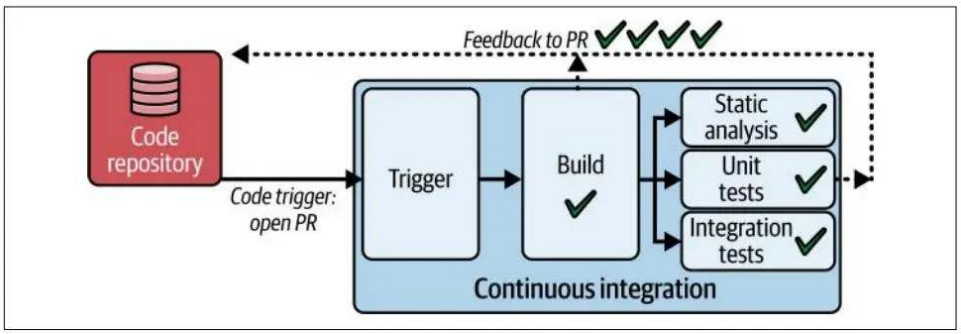

图 3-2. 由打开 Git PR 触发的 CI 流水线

此示例在开发者打开拉取请求时触发。此流水线的目的是在变更**合并到主分支之前**验证 PR 中提出的变更。让我们来看一下这些步骤：

1. 代码触发

开发者或 AI 智能体在托管代码库（例如 GitHub、GitLab、Bitbucket）上打开拉取请求，从而触发流水线。

2. 检出

流水线从 PR 中指定的分支检出源代码。

3. 构建

代码被编译（如果需要）并构建成可执行或可部署的制品。

4. 静态分析

Linter 和代码分析器等工具扫描代码，查找风格违规、潜在 Bug 和安全问题。

5. 单元测试

执行自动化测试，验证单个代码单元的功能。

6. 集成测试

可以运行相对快速的测试，以验证代码不同组件之间的交互。

7. 反馈

流水线向开发者提供 PR 状态（成功/失败）和发现的任何问题的反馈。此反馈直接显示在托管代码库上的 PR 中。

这个流水线检测并通知开发者其代码中的任何问题。构建步骤确定代码变更是否破坏了构建。测试步骤回答以下问题：此代码是否实现了预期功能？此代码是否包含安全漏洞、不安全操作、潜在 Bug、不良实践、已弃用功能，甚至是不一致的格式？

代码流水线通过在拉取请求打开或更新时检测问题和运行快速测试，为开发者提供了近乎实时的反馈。它回答了关于代码功能、安全性和质量的关键问题。然后，开发者可以迅速解决问题，完善 PR，或者在所有检查通过时放心地合并它，从而加速开发并确保代码库的健壮性。

（在第四章中，我们将探讨当 PR 合并时触发的补充性 CI 流水线。此流水线将新代码部署到测试环境并执行运行时间更长的测试套件。）

请注意，虽然我们的示例流水线使用代码变更触发器，但 CI/CD 系统通常提供其他触发选项，例如计划触发和手动触发，以提供更大的灵活性。

关键的构建步骤

构建步骤涉及将代码打包成可部署制品。可部署制品的示例包括容器镜像（用于在 Kubernetes/无服务器环境中部署）、特定语言包（例如 JAR、npm、NuGet 等）以及移动应用程序包（例如 APK 或 IPA）等。例如，用编译型语言（如 C++）编写的代码首先被编译，然后链接以创建机器码。解释型语言通常需要一个构建步骤来将代码打包成中间格式，例如 Java 归档 (JAR) 文件，以便在运行时编译。其他解释型语言，包括 JavaScript，可以被转译或压缩以优化执行。

根据代码类型，此步骤或系列步骤依赖于构建自动化工具、任务运行器或构建脚本。

构建自动化工具协调整个构建过程。流行的自动化工具示例包括：

Make 和 CMake

Make 是最古老、最基本的构建工具之一。它使用 Makefile 来定义文件之间的依赖关系以及构建它们所需的命令。CMake 是一种较新的跨平台构建系统生成器，可以生成 Makefile、Visual Studio 项目和其他构建脚本。它广泛用于 C 和 C++ 项目。

Ant

一种早期的基于 Java 的构建工具，使用 XML 描述构建过程。它以其灵活性和跨平台兼容性而闻名。

Maven

另一种流行的 Java 构建工具，它不仅限于编译。它管理依赖、构建、测试和打包项目。

Gradle

一种较新的构建工具，结合了 Ant 和 Maven 的优点。它使用基于 Groovy 的 DSL 来定义构建，并提供更灵活、更简洁的语法。

Bazel

由 Google 开发，Bazel 是一种功能强大的构建系统，专为大型项目设计。它以其速度、可伸缩性和对多种语言的支持而闻名。

MSBuild

一个构建自动化平台，通常与 .NET 框架和 C#、Visual Basic .NET 和 F# 等语言一起使用。

Cargo

Cargo 是 Rust 编程语言的包管理器，用于构建、编译和管理 Rust 项目。

任务运行器自动化开发工作流中的重复性任务，例如压缩、合并和转译。JavaScript 广泛使用的任务运行器包括：

npm 脚本

作为 Node 包管理器 (npm) 的一部分，npm 脚本是定义在 package.json 文件中的简单脚本，可以自动化常见任务，例如启动开发服务器、运行测试和进行生产构建。

Gulp

一种流式构建系统，使用 JavaScript 代码定义任务。它以其文件处理速度和效率而闻名。

Grunt

另一种用于 JavaScript 项目的任务运行器，Grunt 使用配置文件定义任务。它以其庞大的插件生态系统而闻名。

Webpack

一个主要用于 JavaScript 应用程序的模块打包器。它可以将 JavaScript、CSS 和其他资产打包成优化的文件以供生产使用。

Rollup

另一个模块打包器，以其专注于生成比 Webpack 更小、更高效的捆绑包而闻名。

最后，构建脚本是自定义脚本（通常用 Bash、Python 或其他脚本语言编写），它们定义构建项目所需的具体步骤和命令。它们可以与构建自动化工具或任务运行器结合使用。

通过静态分析优先保障质量和安全

在代码构建完成后，我们立即运行静态分析工具，其中可能包括 Linter。Linter 是一种特定类型的静态分析工具，用于检查编码风格（例如，确保一致的格式和命名模式）；对于 JavaScript 等解释型语言，Linter 检查拼写错误、缺少分号或不正确的语言用法。这些工具在不执行源代码的情况下检查源代码，类似于发布前校对文档。它们有助于在开发过程的早期识别潜在问题。静态代码分析涵盖了一系列技术，用于评估代码的：

潜在 Bug

识别常见的编程错误，例如空指针解引用、资源泄露或逻辑缺陷。

安全漏洞

检测可能导致 SQL 注入、跨站脚本 (XSS) 或其他攻击的不安全编码实践。

代码异味

标记可维护性问题，例如重复代码、过度复杂或未使用的变量，并提出重构建议。

遵循标准

强制执行编码指南，有时还包括特定于语言或项目的最佳实践，以确保一致性和可读性。

通过将这些静态分析工具集成到开发过程的早期阶段，我们不仅能确保代码质量，还能实施一种称为“左移安全”的最佳实践。“左移安全”是指在开发的最早期阶段实施安全实践的策略。我们将在第五章深入探讨左移安全，并探索 AI 如何帮助快速修复安全问题。

自动化测试：尽早测试，频繁测试

自动化测试是 CI/CD 流水线的基础。在我们的示例流水线运行静态分析检查后，它会针对新代码执行单元测试和集成测试。让我们看看这些测试类型：

单元测试

这些测试验证最小的独立代码块（单元），例如函数或方法，以验证它们在隔离环境中按预期运行。想象一个简单的天气应用程序，它从外部 API 获取天气数据，对其进行处理，然后显示给用户。单元测试可能会测试处理原始天气数据的函数，验证它们是否正确地将数据转换为所需的格式。这些测试仅验证转换逻辑。

集成测试

这些测试侧重于验证软件模块之间的交互，确保正确的通信和数据交换。集成测试相对快速，通常在单元测试之后进行，并且像单元测试一样，有助于及早发现问题。同一个天气应用程序的集成测试可能会关注数据获取和处理模块如何交互。这些测试可以验证应用程序是否正确地从 API 检索和处理天气数据，包括错误场景，使用部分模拟来模拟真实世界的 API 响应。与隔离组件的单元测试不同，集成测试评估多个组件如何协同工作。在流水线早期使用的集成测试（例如在我们的示例流水线中）应避免慢速操作，例如访问数据库、文件系统或其他外部系统。

单元测试和集成测试框架数量众多，因语言而异，例如：

Java

JUnit 5 和 TestNG 是用于单元测试的框架。Mockito 和 Spring 用于 Java 集成测试。

JavaScript

Jest 和 Mocha 广泛用于 JavaScript 单元测试。Jest 也支持集成测试。

Python

pyTest 和 pyUnit (UnitTest) 既可用于单元测试，也可用于集成测试。

.NET

NUnit 和 xUnit 用于 .NET 单元测试，而 Moq 和 NSubstitute 通常用于集成测试。

Ruby

RSpec 支持 Ruby 的单元测试和集成测试。

iOS 的 XCTest 和 Android 的 Espresso 是移动单元测试和集成测试的标准。

单元测试和集成测试是第一道防线，它会提醒开发者代码中潜在的 Bug 或回归问题。这些快速的自动化检查只是我们测试策略的开始。在第四章中，我们将探讨当 PR 关闭并合并时触发的后续流水线。

彻底测试每个代码单元，包括所有可能的场景，会产生大量但至关重要的测试套件 —— 即使对于看似简单的代码也是如此。然而，由于单元测试是隔离的，不依赖外部资源，因此它们执行迅速。

我们的流水线优先考虑这些快速的单元测试作为基础，然后是验证不同组件如何协同工作的集成测试，最后是数量较少但全面的端到端测试，它们模拟真实世界的使用情况。

在“测试金字塔”中，我们将探讨测试金字塔框架，它说明了如何平衡不同测试类型以实现最佳软件质量。

测试金字塔

测试金字塔提供了一个模型，用于战略性地构建我们的测试，根据它们的范围和速度来优先排序不同类型。虽然测试金字塔有时会以特定测试类型来描绘每个层，但我们更倾向于将层概念化为包含广泛测试类别，如图 3-3 所示。

在金字塔的底部是预部署测试，包括单元测试、集成测试和静态扫描等类型。这些测试量小且执行迅速。

集成测试可以指一系列测试策略。不与数据库和网络服务等外部系统交互的集成测试速度快，并包含在此级别。宽阔的金字塔底部反映了这些类型的测试套件应该庞大，并且理想情况下应覆盖整个代码库。测试应旨在为开发者提供快速反馈。

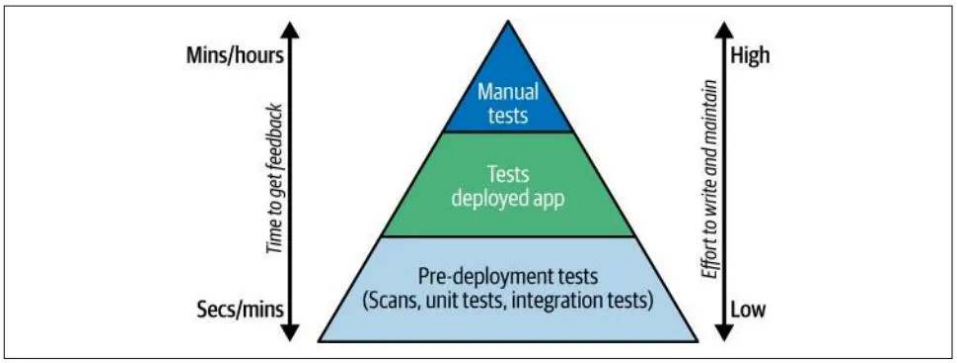

图 3-3. 大量的快速测试构成了测试金字塔的底部；较小量的慢速测试构成了更高层

向上移动金字塔，我们将中间层描绘为包括我们在预生产测试环境中对已部署代码执行的任何类型的测试。通常，这些测试比上面提到的测试慢，但可以提供关于系统整体如何运作的宝贵洞察。

在金字塔的顶端，我们发现手动测试。这些测试速度慢且劳动密集，并且在代码经过多层自动化测试验证后进行。

采用金字塔方法使团队能够在测试工作中平衡速度、成本和有效性。通过专注于小而快的测试的坚实基础，并辅以针对已部署代码的战略性测试，我们可以在最大程度地减少所需时间和资源的同时实现全面的测试覆盖。

强大的测试策略是精简流水线的关键，可加速高质量发布的交付。在“持续集成工具”中，我们将考虑 CI 工具的选择如何优先考虑这一因素。

持续集成工具

有效的 CI 流程对于现代开发团队至关重要。在本节中，我们将探讨传统的 CI 工具以及现代工具的特点。

一家大型全国性零售商 —— 我们的客户 —— 在预计数字需求激增的情况下，发现自己正处于十字路口。其传统 CI/CD 工具，包括 Jenkins，分散在客户端网页、移动端和后端服务团队中，导致构建时间过长，每年给公司造成高达 50 万美元的开发者空闲时间损失。这些工具不仅抑制了创新，还带来了重大的安全风险，每年花费 80 万美元用于维护和自定义脚本更是加剧了这一问题。这项巨额投资使资源从增强客户体验方面转移。面对日益增长的挑战和不断攀升的成本，该零售商寻求一个统一的 CI/CD 平台，以简化操作、加速创新并强化安全性。

该公司日益增加的挑战揭示了 Jenkins 固有的局限性，尤其是在组织规模化和数字需求加剧时。让我们来看看其中一些局限性。

Jenkins 考量

Jenkins 将持续集成引入主流，功不可没。作为一个开源自动化服务器，Jenkins 利用庞大的插件生态系统来扩展其功能和特性，并赋予用户无限定制流水线的能力。Jenkins 插件市场是一个集中式存储库，用户可以在其中查找和安装数千个社区开发的插件。Jenkins 社区庞大，其文档丰富。它是一个适用于各种开发环境的适应性解决方案。

虽然 Jenkins 因其专用插件（例如，大型机）对于传统系统仍然很有价值，但现代 CI 流水线需要更多功能。如今的开发环境需要能够提供速度、安全性、协作工作流以及与多个云提供商技术、Kubernetes 编排和容器化应用程序原生集成的 CI 工具。以下各节探讨了使 Jenkins 不太适合这些现代要求的具体挑战。

插件复杂性

Jenkins 的灵活性和庞大的插件生态系统常常导致复杂且碎片化的架构，从而阻碍可维护性并增加开发者的负担。对 Groovy 脚本进行流水线定制的依赖可能会使故障排除和更新变得繁琐，特别是随着流水线数量和复杂性的增加。

此外，现代 CI/CD 解决方案通常采用“流水线即代码”范式，使用 YAML 等声明式语言来定义流水线。这种方法通常被认为比 Jenkins 的脚本化方法更直接且更易于维护。基于 YAML 的流水线通常比 Groovy 脚本更具可读性且更易于维护（可能存在例外），Groovy 脚本随着流水线规模和复杂性的增长会变得复杂且更难调试。将流水线定义为代码，可以将其与应用程序代码一起存储在版本控制系统 (VCS) 中。这确保流水线变更被跟踪、审查和审计，从而促进团队成员之间更好的协作。因此，“流水线即代码”方法可以实现更好的版本控制、协作和更简单的故障排除。

最后，管理众多插件（每个插件都有自己的配置）会引入维护开销。团队成员发现自己将宝贵的时间花在解决插件冲突、更新依赖项和破译神秘错误消息等繁琐任务上。这会分散对创新和核心开发的关注，从而减缓创新和功能交付。

可伸缩性挑战

Jenkins 的架构主要为单服务器设置设计，随着作业、流水线和用户数量的增加，其扩展效率常常会遇到困难。这可能导致性能瓶颈、构建时间变慢和整体系统不稳定。虽然 Jenkins 提供分布式构建和集群选项，但设置和维护这些解决方案可能很复杂且资源密集，需要专业知识和大量开销。因此，水平扩展 Jenkins 以满足大型组织或高吞吐量 CI/CD 工作流的需求通常成为一项重大挑战。

安全隐患

虽然 Jenkins 插件提供了可扩展性，但它们也引入了潜在的漏洞。每个插件都有自己的代码库和依赖项，从而扩大了 Jenkins 实例的攻击面。监控这些插件是否存在漏洞并确保及时更新成为管理员持续的开销。此外，配置 Jenkins 安全性，包括用户权限、访问控制和网络配置，可能非常复杂。错误配置可能会使系统暴露于未经授权的访问或恶意活动。插件生态系统的动态性质以及错误配置的可能性意味着您必须警惕监控风险并积极缓解 Jenkins 环境中的风险。

资源使用和效率问题

Jenkins 的资源消耗可能是一个显著的缺点，特别是当作业和插件的数量增加时。基于 Java 的架构（JVM 的运行时要求、垃圾收集行为和框架抽象）通常会导致高内存使用，管理大量并发构建可能会对 CPU 和磁盘资源造成压力。这可能导致构建时间变慢、基础设施成本增加和潜在的性能问题。在更大的环境中，水平扩展 Jenkins 可能会变得复杂且资源密集，需要额外的硬件和仔细的配置。

此外，在 CI 流水线中构建 Docker 镜像可能会迅速变得资源密集且成本高昂，尤其是在处理大型代码库或频繁提交触发大量并行构建时。每个镜像都需要计算资源、存储空间和网络带宽 —— 这些成本在不同环境和分支中都会成倍增加。同样，虽然全面的可观测性提供了宝贵的系统洞察，但实施过度日志记录可能会产生其自身的问题：存储成本飙升，信噪比降低，以及处理开销增加。在全面覆盖和资源效率之间找到正确的平衡仍然是一个关键挑战。

超越 Jenkins

由于 Jenkins 的局限性，像我们国家零售商这样的公司常常会超越它，并寻求提供以下功能的现代全托管解决方案：

内置、完全支持的构建块

现代 CI/CD 工具提供丰富的内置、完全支持的构建块库，可简化流水线设置。这消除了对社区维护插件的依赖，确保了可靠性和稳定性。然而，鉴于定制需求，大多数解决方案仍支持通过自定义插件进行扩展。这使团队能够自动化独特的工作流，并根据其特定需求定制 CI/CD 环境。

声明式流水线定义

现代 CI/CD 工具使用 YAML 等声明式代码简化流水线定义，使其比 Jenkins 的 Groovy 脚本更易于访问和维护。这加速了设置并最大程度地减少了与手动脚本编写相关的错误。

原生支持容器化和编排

Jenkins 早于 Docker 和 Kubernetes 的广泛采用，虽然 Jenkins 流水线可以使用插件来处理和编排容器，但缺乏原生支持通常会导致繁琐的配置。相比之下，新工具无缝地集成了容器化和编排功能，简化了应用程序在容器化环境中的部署和管理。

在接下来的部分中，我们将探讨 Jenkins 之后的新工具提供的其他现代特性。在我们关注这些特性之前，让我们思考一个在考虑 CI/CD 工具时最基本的问题：是自行托管和管理工具，还是选择全托管解决方案。这个决定将影响从开发速度和成本效益到维护要求的一切。鉴于移动端的重要性，选择能够处理移动应用程序构建和部署复杂性的 CI/CD 设置至关重要，我们将探讨移动应用程序开发特有的考量因素。

托管选项

组织有三种主要的 CI/CD 系统构建基础设施选择：本地自托管、云端自托管和供应商托管（云）。每种选择都有其独特的优缺点，应仔细考虑：

本地自托管解决方案

在本地自托管 CI/CD 系统，您可以完全控制和拥有其基础设施和数据。这种方法允许最大程度的定制，能够根据特定的安全协议和组织需求进行调整。此外，一些组织可能更喜欢与本地解决方案相关的一次性付费模式。然而，这种方法也有几个缺点。它需要对硬件和软件进行大量前期投资，以及时间和精力进行维护和更新。对持续维护的需求和潜在的可伸缩性挑战可能会耗费资源，特别是对于小型组织而言。

云端自托管解决方案

云端自管理模型在控制和可伸缩性之间取得了平衡。组织可以控制其 CI/CD 软件，同时利用云的灵活性和可伸缩性。这种方法减少了对物理硬件的需求，并且与本地解决方案相比简化了扩展。

云托管应用程序在称为 Hypervisor 的虚拟化环境中运行，在考虑云托管时，您选择的 Hypervisor 类型将影响简单性和性能。需要了解的两种 Hypervisor 类型是：

类型 1 裸机 Hypervisor

这些 Hypervisor 直接在硬件上运行，提供卓越的性能和隔离性，但需要专用硬件。

类型 2 嵌入式 Hypervisor

这些 Hypervisor 在操作系统之上运行，提供更简单的设置和灵活性，但性能可能较低。

裸机可能更适合高要求、高安全性的设置，而嵌入式则适用于需求不那么密集且预算有限的情况。

任何云托管工具集都将需要持续维护和更新，您的组织仍将负责管理云基础设施。这可能导致与本地解决方案类似的挑战，尽管前期成本可能会降低。

全托管、供应商托管解决方案

供应商托管的 CI/CD 解决方案提供完全托管的服务，其中供应商处理基础设施、维护和更新。您的组织专注于开发而不是基础设施管理。这些解决方案高度可伸缩、易于使用，并且通常遵循按需付费模型，使其具有成本效益。然而，它们可能比自托管选项提供更少的定制，并可能限制您的组织根据特定需求定制系统的能力。此外，这种方法可能会出现数据安全和潜在的供应商锁定问题。

移动应用开发特有挑战

拥有健壮高效的 CI/CD 解决方案对于跟上移动用户期望的快速发布周期和高质量应用程序至关重要。移动端开发带来了独特的挑战：您的流程和 CI/CD 工具必须能够管理设备碎片化和频繁的移动操作系统更新。

在选择自托管和全托管 CI/CD 解决方案时，请考虑自托管解决方案虽然提供控制和定制，但可能导致物理硬件限制等挑战。此外，您的团队将负责构建环境的持续维护和更新。这些复杂性可能导致意想不到的成本。Xcode 等 iOS 开发工具的频繁发布周期需要定期更新硬件，这对于任何团队来说都可能是大量的时间和资源消耗。

另一方面，全托管 CI/CD 解决方案通过提供构建环境的自动更新和可预测的成本来缓解这些痛点。这使您的团队能够专注于构建功能和改进应用程序，而不是管理基础设施。此外，专门为移动开发优化的全托管 CI/CD 解决方案提供移动端专用集成和功能，可简化开发过程。其中许多平台为您全面管理移动开发中的挑战，例如设备碎片化和操作系统更新。

加速软件构建的现代特性

回到我们的零售商：它研究了更新的选项，并决定放弃 Jenkins 以及与其配合使用的零散插件和工具。该公司选择了一个统一的平台，简化了其工具集，同时提供了所需的伸缩性和成本节约。它能够将服务、客户端网页和移动团队的 CI/CD 流程整合到这一个平台上。新平台消除了对大量脚本编写的需求，节省了开发人员的时间，并使他们能够专注于创新。它还利用 AI/ML 进行测试，进一步节省了成本并大大加快了构建速度。此外，统一平台通过在流水线早期支持安全测试来提高安全性，从而能够更快地检测和修复漏洞。新平台的效率、安全性和可靠性使该零售商能够轻松应对其数字增长。

在接下来的部分中，我们将探讨现代系统中能够实现更快、更经济高效且更安全的流水线的特性。

通过缓存加速构建

现代构建环境是短暂的，通过提供隔离、经济高效且可伸缩的设置来增强敏捷性，从而加速开发周期，同时保持 CI/CD 流水线各个阶段的一致性。然而，短暂环境每次都需要从头开始设置整个构建过程，包括下载依赖项、编译代码和生成制品。这非常耗时。

缓存是 CI/CD 中用于存储和重用构建制品、依赖项、Docker 层和中间结果的技术。这通过避免冗余操作并仅专注于构建已更改的部分来显著缩短构建时间，这不仅加速了开发周期，还节省了计算资源和能源。现代 CI/CD 系统智能地管理此缓存过程，无需人工干预即可优化构建。缓存可以在不同阶段进行 —— 缓存软件依赖项、缓存 Docker 层以及缓存 Bazel、Gradle 和 Maven 等工具的构建输出。

利用 AI 简化构建、缓存和测试

一个 AI 原生 CI 解决方案将无缝集成生成式 AI (GenAI)、智能体 AI (Agentic AI) 和 MCP，以增强软件构建、所需组件缓存以及每次构建的测试。让我们更详细地了解这些增强功能。

构建阶段增强。生成式 AI 可以自动化重复任务的样板代码创建（例如，Dockerfile 模板、CI 配置文件），减少人工工作量。它还可以分析历史构建数据，预测依赖冲突并建议最佳版本，从而最大程度地减少构建失败。生成式 AI 的另一个有趣的用例是根据项目结构生成优化的 CI 流水线 YAML 配置，减少试错式设置。

智能体 AI 可以检测构建失败（例如，缺少依赖项），然后自动使用纠正后的配置重试并记录根本原因。它还可以根据工作负载需求动态扩展构建资源（例如，云实例），平衡速度和成本，并且可以动态地将单体构建拆分为可并行任务，从而缩短执行时间。

MCP 可以标准化分布式团队的环境变量、构建标志和工具链版本，确保一致性并通过 MCP 的集中式缓存，在相关项目之间共享预构建制品（例如，编译的库），从而避免冗余构建。

缓存阶段增强。生成式 AI 可以使缓存技术变得更智能。它可以根据代码变更预测需要哪些依赖项（例如，node_modules、.m2 制品），并在构建开始前对其进行预缓存。机器学习模型可以通过分析代码差异模式来识别过期缓存，确保只保留相关制品。智能体 AI 可以实时标记并清除损坏的缓存（例如，损坏的制品），从而防止构建失败。

在可伸缩基础设施中使用 MCP 具有许多优势，包括通过标准化 API 实现跨 CI 流水线的安全、低延迟缓存共享，以及通过在 CI 运行之间缓存中间构建输出（例如，Docker 层）来减少冗余数据传输。MCP 可以通过标准化 API 实现并行 CI 作业之间的安全缓存共享，从而消除单体仓库架构中的冗余构建。

测试阶段增强。考虑这样一种情况：开发人员在一个大型应用程序中修改了一个很少使

用的组件中的一行代码。我们拥有大量健壮的单元测试套件，代码覆盖率很高；这些是我们测试策略的基础，是测试金字塔的底部。然而，当代码变化很小时，执行整个测试套件会导致漫长、资源密集且效率极低的测试周期。

现代工具可以通过 AI 工具来缓解这些问题，这些工具智能地选择并执行仅与修改代码直接相关的测试。这种方法显著减少了测试所需的时间和资源，从而带来了更快的反馈循环和更高效的开发过程。

Harness 测试智能 (TI) 是这种方法的一个示例。让我们看看 TI 的底层工作原理。三个组件协同工作以实现 Harness TI：

TI 服务

此服务使用 AI 并理解您的代码库、Git 提交和单元测试，并利用这些数据动态构建一个图，映射代码方法与其对应单元测试之间的关系。此图会持续更新以反映代码库中的变更。

一个测试运行器代理

此组件与服务通信并执行测试。

一个测试步骤

这是您添加到 CI 流水线中以将 TI 集成到您的工作流中的步骤。

TI 工作流始于开发者发起拉取请求并触发流水线。TI 服务分析代码变更并将其与图进行比较，以识别需要执行的测试。它不仅考虑代码修改，还考虑测试本身的任何变更或新增。这确保了代码库的所有相关方面都得到彻底测试，同时避免了冗余的测试运行。

因此，通过专注于受影响的测试，智能测试方法可以显著减少测试时间，尤其是在具有大量测试套件的大型项目中。这转化为更快的构建和更快的开发者反馈，使他们能够更快地识别和解决问题。

AI 驱动的构建和测试洞察

现代 CI/CD 工具还利用生成式 AI 来自动化繁琐的任务，并在出现问题时提供洞察。例如，工具可以自动生成您的流水线，分析代码中潜在的问题，并实时排查构建和部署失败。如果 CI 构建失败，生成式 AI 可以分析日志文件，查明错误，甚至建议潜在的修复方案。这节省了您的时间，减少了停机时间，并加速了软件交付过程。

智能体 AI 也可以用于根据组织的黄金标准来提出优化现有流水线的建议。由于组织往往是优化现有流水线而非创建新流水线，因此此功能将极其有价值。

生成式 AI 的另一个绝佳用例是编写意图驱动测试。测试，尤其是 UI 测试，如果 UI 发生

变化，可能会非常手动且不稳定。通过使用生成式 AI，开发人员和质量保证工程师只需说明测试的意图，让生成式 AI 自动生成步骤。我们将在第四章详细讨论意图驱动测试。

最后，AI 还可以用于以道德和负责任的方式生成测试数据。一些示例包括：在使用生产数据进行模型训练时确保符合 GDPR 和其他法规，在整个数据生成过程中维护数据隐私和安全，以及使用适当的算法生成合成数据。

通过企业可观测性统一 CI/CD 指标

一个现代的 CI/CD 解决方案应该是一个团队合作者，与您企业生态系统中的其他关键平台协同工作，特别是您组织赖以理解系统行为、识别性能瓶颈以及主动检测和解决问题（在它们影响用户或业务运营之前）的可观测性平台。可观测性平台包括包含 Logstash 和 Kibana 的流行开源平台 Elastic，以及知名的商业选项 Datadog 和 Splunk。

现代持续集成工具通过实现 OpenTelemetry（一个开源框架）向这些平台提供遥测数据。这带来了 CI/CD 指标，从而实现可观测性和仪表板，帮助您了解正在发生的事情并提高构建性能和可靠性。

现代 CI/CD 对单体仓库的支持

当管理跨多个代码库的复杂代码时，版本控制和依赖管理变得非常具有挑战性。单体仓库是包含项目或组织所有代码的单个代码库，提供了一种集中管理复杂代码的方法。单个代码库通过保留任何共享库或组件的单个副本来简化依赖管理，并简化了跨不同项目的代码共享和重用。虽然单体仓库增加了合并冲突的风险，并且需要仔细设计以避免紧密耦合的代码，但许多大型公司已成功将其应用于海量代码库，这表明有效管理的单体仓库可以提供一种非常可伸缩的方法。

在采用单体仓库策略时，了解单体仓库对代码库和 CI 工具的独特要求非常重要。在数百名开发者可能贡献到大型单体仓库的情况下，高效管理变更和拉取请求变得至关重要。团队必须能够按子目录定义适当的访问权限，部分是为了确保只有相关的审阅者才能收到每个变更的通知。代码库应支持子目录特定所有权。

单体仓库需要 CI 系统能够对已更改的组件进行选择性构建和测试，并且支持高级依赖管理、缓存和并行执行。Harness CI 等工具通过以下功能支持这些需求：基于路径的触发器（仅当代码库中特定目录发生更改时才运行流水线，例如，针对 serviceA/ 的更改触发服务 A 的流水线），以及稀疏检出（克隆子目录而不是整个代码库）。这优化了资源使用并加速了反馈循环，同时保持了依赖完整性。

总结

持续集成已成为一项不可或缺的实践，它减少了集成问题，提供了更快的反馈，并提高了整体效率。在本章中，我们探讨了现代全托管 CI/CD 工具的特性，并将其与自托管的成本和挑战所带来的权衡进行了对比。我们探讨了优先进行更快、更小的单元测试以获得快速反馈的重要性，然后是较慢的测试类型以实现全面覆盖。我们所探讨的持续集成流水线就例证了这种实践：在打开 PR 的背景下，我们进行构建、完成静态扫描，然后运行快速测试，以确保我们的代码按预期运行并且不引入回归问题。我们还探讨了 AI 原生 CI 工具如何利用生成式 AI、智能体 AI 和 MCP 来增强 CI 的构建、缓存和测试阶段的各种方式。

在第四章中，我们将继续探讨 CI/CD，重点关注部署到测试环境以及执行评估系统性能、弹性以及端到端行为的慢速测试。

第 4 章：部署到测试环境

在第 3 章中，我们探讨了持续集成的基础知识，重点关注 CI/CD 流水线的早期步骤：主要是构建和部署前测试。我们详细讲解了一个在 PR 打开时触发的示例流水线，如 图 4-1 所示。

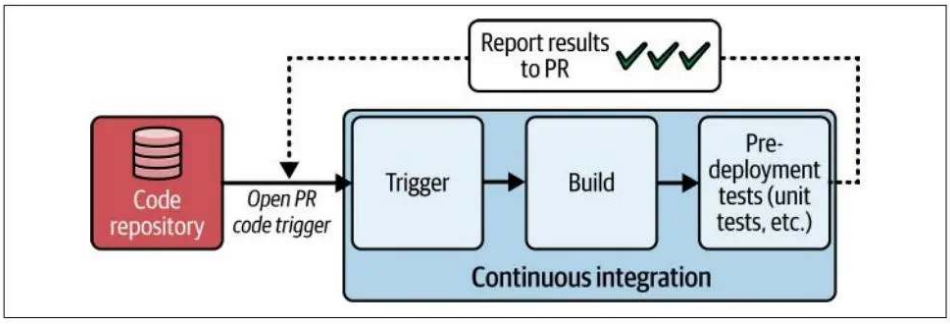

图 4-1. CI 流水线

该流水线构建并打包了代码，进行了静态代码分析，并执行了早期快速测试，包括单元测试和轻量级集成测试，从而向 PR 提供构建和测试反馈。这些步骤确保拉取请求中的代码处于可合并状态，并提供信心，确保合并后的代码能按预期运行，不会引入任何回归。假设 PR 中的代码更改已准备就绪，开发人员即可合并该 PR。

随着新代码的合并，下一步是通过部署到测试环境并运行一系列测试来为生产环境做准备。AI 和 ML 工具正在被整合到部署流程中。这些工具帮助团队做出更好的部署决策，主动识别潜在问题，并简化验证流程。优秀的 AI 实施非但不会增加复杂性，反而能减轻开发人员的认知负荷，同时提高部署的可靠性。

在 CI 步骤和生产发布之间，我们主要关注测试。我们想知道发布是否已为我们的用户做好准备。如果可以安全发布，我们希望快速将其交付给用户，以增强用户体验，并可能提高客户参与度和忠诚度。如果我们的软件有问题，我们需要迅速发现并解决。这种动态是发布有价值更新的障碍，缺陷引入与将其反馈给开发团队之间的时间越长，相关开发人员对工作的记忆就越不清晰。他们将不得不花费更多时间和精力来熟悉这些代码段，从而使修复成本更高。如果开发人员已经深入到下一个任务中，那么该任务也可能会被中断，完成成本也会更高。

当发布准备就绪时，我们将把发布部署到一个或多个环境，以便我们可以在运行中的代码上进行测试。正是在这些预生产环境中，我们弥合了开发与实际使用之间的差距，确保我们的软件不仅功能正常，而且已为实际场景做好准备。

图 4-2 概述了我们的整个交付过程。AI 越来越多地被嵌入到整个流水线中，以加强测试和部署决策。

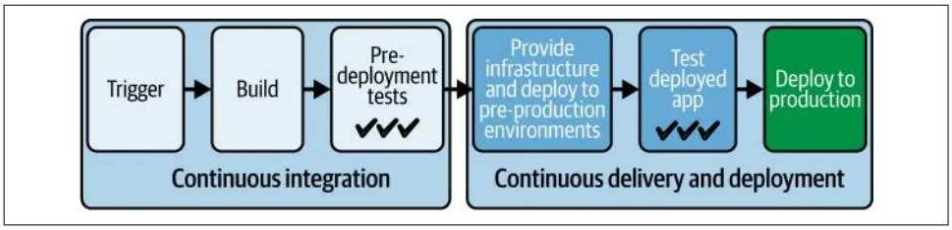

图 4-2. 高层级交付流程

在本章中，我们将探讨预置基础设施、部署到一个或多个预生产环境以及对软件进行测试的步骤。此外，我们还将介绍关键的最佳实践，包括：

- 使用 IaC 创建与生产环境一致但规模更小的低级环境
- 使用“类生产”部署来一致地移动您的应用程序
- 将测试连接到部署流水线
- 选择在何处将 AI 应用于部署以及在何处保持谨慎

流水线中的这个阶段是关键阶段，开发和运维关注点在此交汇。通过理解和实施这些最佳实践，您将能够更好地根据您的特定项目需求（无论项目规模或复杂性如何）确定最佳的测试环境数量和类型。您将了解如何平衡开发速度和运维稳定性，确保您的软件经过彻底测试并已准备好发布。

建立统一的部署流程

当我们继续沿着交付流水线迈向生产环境时，我们需要考虑必要的部署步骤和部署环境。为了实现可预测和可靠的交付流程，我们需要可预测和可靠的部署步骤和环境。

在本节中，我们将介绍从测试到生产环境实现我们所追求的可预测性和可靠性的最佳实践。在第 8 章中，我们将更详细地介绍生产发布和生产环境。

一致地部署到每个环境

自动化是 DevOps 的基础，而我们交付流水线的一项关键功能就是自动化预生产环境的设置以及向这些环境的部署。正如我们需要在发布软件之前对其进行验证一样，我们也需要采取措施来验证我们如何部署软件。

我们通过始终使用相同的方法部署到预生产环境和生产环境来实现这一点。这种一致性测试了我们的部署方法，并最大程度地降低了在将软件部署到生产环境时重复这些步骤可能出现意外问题的风险。

以下最佳实践有助于提供我们所追求的可预测性。

使用一致的工具

开发人员使用简单工具启动自己的轻量级部署流程以部署到测试环境，而运维团队则专注于使用企业级工具进行生产部署，这并不少见。流程之间的这种不一致性导致更改以“有问题才通知”的方式进行沟通，即开发人员更新其流程后，会忘记通知运维团队，直到出现问题。

应避免这种方法，因为它限制了非生产环境中测试的有效性，并导致自动化脚本工作的重复。相反，应为所有部署采用统一的工具集。

鼓励一致性的一种方法是为开发人员提供易于使用的预制模板流水线，称为”黄金流水线”或”铺就之路”。我们将在第 10 章中更详细地探讨这一点。至少，您的开发人员和运维团队需要就一套通用的部署工具达成一致。

使用一致的流水线步骤和部署策略

无论您是使用 CI/CD 工具还是自定义部署脚本，操作序列都应在不同环境之间保持一致。金丝雀发布或蓝绿部署等高级部署策略通常是基于降低生产部署风险而选择的。如果您的生产环境采用了这些策略，请在您的预生产环境中复制它们。同样，如果您在生产环境中使用特性标志来发布单个特性，请在测试环境中使用特性标志来推出这些特性。这种一致性最大程度地降低了部署期间引入差异或疏忽的可能性。

我们将在第 7 章和第 8 章中更彻底地介绍生产部署和这些渐进式部署策略。目前，请注意您使用的步骤和策略应在每个层面进行复制。虽然测试环境可能由于成本或资源限制而规模较小，但部署时应将其视为生产环境。例如，生产环境中的滚动部署可能一次向 10 个目标部署两个节点，而在测试环境中，您可以一次向 3 个目标部署一个节点。这种方法确保您的生产部署步骤和策略在部署到测试环境的每个版本中都经过彻底测试。

在第 7 章中，我们将深入探讨 AI 技术如何验证部署在新环境中没有引起问题。这些相同的方法也应在较低级别环境中使用，以验证其正常工作，并保护我们的测试不会针对

有缺陷的安装运行。

使用参数化处理差异

环境之间不可避免地会存在差异。目标名称、服务 URL 和密码可能不同。与其为每个环境创建唯一的部署脚本，不如利用变量来适应这些差异。这使您能够维护一个单一、可适应的脚本或流水线，可在运行时根据特定环境进行调整。

通过在部署中保持一致性，您将创建健壮且可靠的交付流水线，从而增强团队无缝高效发布软件的能力。

利用 AI 加速流水线创建

在第 3 章中，我们讨论了自动化流水线创建。模板化仍然是一种很好的模式 —— 您希望您的 AI 能够利用您组织的模板并引入适合您的项目和团队的正确调整和变量。无论是由您还是 AI 创建或维护流水线，流水线代码需要管理的越少越好。

利用基础设施即代码实现部署一致性

我们希望拥有一致且可预测的环境，以将发布交付到生产环境。IaC 为我们提供了一种方法，不仅可以实现一致性，还可以像管理代码资源一样精心控制我们的配置。其核心在于，IaC 将基础设施配置视为软件代码。

工程师在本地对 IaC 代码进行更改，并在其开发环境中进行测试。然后，这些更改会像应用程序代码一样提交到版本控制系统（VCS）。通过管理我们的 IaC，我们充分利用了 VCS 和 CI/CD 流水线的这些功能。

代码即服务的特性使得 IaC 成为一个快速受益于大型语言模型的 DevOps 领域。AI 编码助手能够很好地生成和解释 IaC 代码，降低了开发人员和基础设施专业人员采用新 IaC 语言的门槛。对于能够访问环境性能数据或结合云成本功能与 IaC 管理的 DevOps 平台的主要云提供商来说，未来的代码生成工具可能会结合以下基于实时工作负载的运行时优化：

协作与代码审查

版本控制使多个团队成员能够同时处理文件并管理冲突。我们可以定义和强制执行策略，要求对基础设施配置更改进行代码审查。

分支与实验

版本控制允许您创建分支以试验不同的配置，而不会影响主生产环境。

可追溯性与可审计性

版本控制系统（VCS）提供了配置设置更改的完整历史记录。提交信息和更改历史记录帮助您理解系统演变的原因，并且审计追踪对于支持符合安全框架至关重要。

回滚与恢复

如果基础设施配置更改导致问题，您可以快速回滚到以前正常工作的版本，最大程度地减少停机时间并降低对系统的影响。此外，在发生灾难性故障的情况下，您可以使用版本控制的配置将系统恢复到已知的正常工作状态。

自动化测试

交付流水线可以对 IaC 代码运行自动化测试，包括语法检查、安全扫描和合规性测试。然后，这些更改会应用于预演环境进行集成测试，最后，它们会被提升到生产环境，通常采用谨慎的推广策略。

安全性

版本控制有助于强制执行围绕配置更改的安全策略和控制，确保只有经过授权的人员才能进行修改。

考虑一个在科技行业中许多人耳熟能详的场景：一个应用程序在开发环境中运行完美，在预演环境中运行顺畅，但部署到生产环境时却陷入混乱。这种差异通常源于不同环境之间基础设施配置的不一致。通过 IaC 配置定义，您可以确保从开发到生产的每个环境都得到相同的配置。

这种有条不紊的过程确保您的基础设施以受控、可预测的方式演进。它通过消除环境之间意想不到的差异，解决了“在 QA 环境中正常运行”的问题。通过以与对待应用程序代码相同的尊重和严谨对待您的基础设施，您将获得一致性、可靠性和敏捷性。

除了控制和一致性之外，IaC 还提供了多项优势。只需一个命令，您就可以启动与现有基础设施完全相同的全新环境。这不仅使您的流程可重复，而且还充当了准确、实时的文档。由于环境易于创建和销毁，您可以在不使用时将其拆除，从而节省资源并降低成本，并确信它们可以毫不费力地重新创建。

为了有效地实施 IaC，您需要合适的工具，并且有多种流行的选择。Terraform 及其更开放的分支 OpenTofu 采用云无关的方法。如果您完全依赖于特定的云提供商，那么像 AWS CloudFormation 或 Azure Resource Manager 这样的原生工具可能更适合。

通过 GitOps 利用 Git 工作流

GitOps 是一种较新且日益流行的软件部署方法，它建立在代码仓库的能力之上。通过 GitOps 方法，您可以在版本控制的配置中描述所需的基础设施状态。这种描述是声明

性的。GitOps 工具包含一个代理，该代理会定期协调实际环境与 Git 控制配置中描述的所需状态。您无需运行脚本直接部署软件，而是通过更新代码仓库中的配置来启动软件部署。这种方法和 GitOps 工具通常用于 Kubernetes 环境中，以跨机器集群编排容器化应用程序。

通过这种方法，您可以依靠代码仓库来强制执行安全、提供治理，并实施您组织的策略，例如要求通过代码审查和批准进行监督。您的更新是可追溯和可审计的。您可以协作、实验并回滚用于部署软件的配置更新。一旦您进行了更新并合并，GitOps 协调代理就会完成其余工作，获取更新并对目标环境实施更改。

这种方法之所以受到欢迎，是因为管理描述复杂编排云系统的复杂配置非常适合代码仓库的功能。此外，GitOps 解决了环境漂移问题；即，环境在操作上偏离了所需状态。协调代理会自动检测并修复，防止环境中的不一致。

虽然 GitOps 方法功能强大，但在 CI/CD 交付流水线中采用 GitOps 方法进行部署比简单地用脚本推送应用程序更新要复杂。使用 GitOps，您的流水线必须自动化以下步骤：

1. 从您的代码仓库检索配置。
2. 更新配置以引用您的应用程序的最新版本。
3. 将更新后的配置合并回 Git。

然后由 GitOps 协调器接管。

您还可能遇到跨多个集群进行地理复制的应用程序的复杂性。由于许多 GitOps 协调器都针对将应用程序部署到单个集群进行了优化，因此在集群之间维护一致性和同步可能很困难。您可能需要在单一事实来源的需求与某些配置需要针对特定集群进行定制的现实之间取得平衡。商业 GitOps 工具通常在这些更复杂的场景中提供编排和可见性，扩展了开源工具的功能。

尽管存在这些挑战，但在协作、可追溯性和自动化协调方面的优势使 GitOps 成为广泛利用 Kubernetes 的组织的有力选择。

CI/CD 流水线中的持续交付、部署和测试

现在我们已经了解了可预测和可靠的部署步骤与环境的重要性，让我们回到我们的交付流水线。随着新代码的合并，我们现在希望将其部署到一个或多个环境，以便我们可以在运行中的代码上进行测试。图 4-3 显示了一个示例流水线。

在本节中，我们将重点介绍流水线：

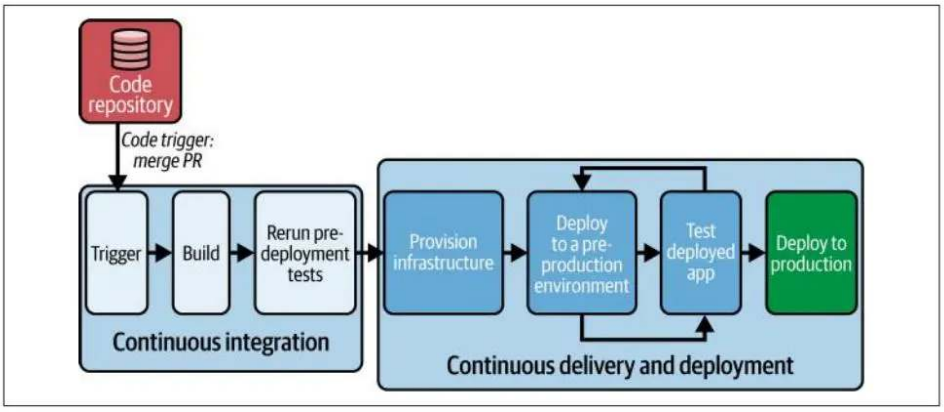

图 4-3. 在预生产环境中测试我们的代码

1. 代码触发器

拉取请求被审查、批准并合并到主分支。在此流水线中，PR 合并触发流水线。

2. 持续集成

流水线重复了我们在上一章中回顾的持续集成步骤，包括检出、构建和执行持续集成测试。

3. 预置基础设施

流水线预置测试所需的预生产环境。

4. 部署到一个或多个预生产环境

流水线将应用程序部署到一个或多个预生产环境。

5. 对已部署的应用程序进行测试

流水线对已部署的软件进行测试。可以运行各种类型的测试，具体取决于软件类型和组织优先级。我们将在下一节中介绍多种不同类型的测试。流水线可以配置为并行或顺序运行多种类型的测试。某些测试可以重复使用相同的预生产环境，而其他测试

可能需要根据测试要求定制的预生产环境。通常，较快的测试优先于较慢的测试。

6. 部署到生产环境

最后一步是部署或推进到生产环境。根据您的交付流程，部署到生产环境的决策可以是自动化的，也可以需要手动批准。我们将在第 7 章中探讨推进策略和部署到生产环境的步骤。

持续交付与持续部署

持续交付（Continuous Delivery）和持续部署（Continuous Deployment）这两个术语经常互换使用。持续交付通常被宽泛地定义为一个自动化软件发布直到生产部署点的过程，在更改上线前需要手动批准。另一方面，持续部署则完全自动化了整个过程，包括部署到生产环境。

混淆产生的原因是流水线会自动部署到中间测试环境。一些人使用“持续交付”来涵盖这些自动化的中间部署，而另一些人则将其保留给不自动部署到任何环境的流程。同样，“持续部署”有时被广泛用于描述任何自动化部署，包括部署到测试环境。

为了避免混淆，我们倾向于广义地使用“持续交付”来指代频繁向用户交付软件的过程。减少手动步骤的数量通常会使这个过程更频繁。当我们讨论特定交付流程中的部署步骤时，我们会包含关于部署环境（中间或生产）和类型（自动化或手动）的详细信息。

测试类型

测试环境对于运行测试至关重要，但测试的选择在很大程度上取决于所开发的应用程序类型、目标用户、软件架构以及预算和时间限制。例如，一般来说，网站的测试优先级与嵌入式软件或 Web API 的测试优先级将大相径庭。在高度受监管行业中的软件服务与必须对大量零售用户直观且引人注目的软件之间的测试优先级也会有所不同。您选择的测试及其频率会显著影响应用程序质量、基础设施成本和整体交付速度。

AI 驱动的测试平台越来越多地使用机器学习（ML）来优化测试策略。这些平台分析历史测试数据、代码更改、应用程序架构和过去的部署问题，以智能地选择和优先处理测试。例如，AI 驱动的测试选择工具可以识别针对每次代码更改要执行的最具影响力的测试，从而显著加快测试周期。Harness、Tricentis SeaLights 和 CloudBees Launchable 等供应商正在使用 AI 和 ML 技术来优化测试选择。

以下是此阶段常见的测试类型：

端到端测试或功能测试

这些测试是最直接的测试类型，它们模拟真实世界的用户场景，并从头到尾验证整个应用程序流程，以确定软件是否按预期运行。这些测试可以是自动化的，也可以是手动执行的。现代团队更多地采用自动化。Selenium 是一个常用的开源测试自动化框架，许多商业工具也以此为基础。机器学习在这些工具中已经存在了一段时间，但我们正越来越多地看到向 AI 优先方法的转变，我们将在稍后深入探讨。

AI 驱动测试

AI 可以自动生成测试用例，识别边缘情况，并从之前的测试运行中学习，从而专注于最可能出现问题的区域。AI 测试很可能补充或成为您的端到端（功能）测试程序的一部分。

API 测试

API 测试是功能测试的一种形式，它验证 API 是否按预期工作。在分布式系统中，服务通过 API 进行交互，因此确保 API 运行良好至关重要。常见的 API 测试框架包括 SoapUI、Postman、Insomnia 和 Swagger。AI 增强的 API 测试超越了简单的验证，能够智能地探索 API 行为和边缘情况。这些系统可以通过分析 API 文档或实际使用模式自动生成 API 测试场景。

用户体验测试

开发人员、测试人员和产品经理可能会评估新功能，以确保它们易于使用且直观。虽然这可能测试与端到端测试相同的系统，但重点在于评估可用性。

用户验收测试

这些测试通常作为最后一道检查，以确保软件满足最终用户的需求、符合要求并按预期运行。用户验收测试可以包括许多其他类型的测试，从端到端到用户体验和性能测试。这些测试从最终用户的角度进行，目的是对软件发布提供最终和正式的验收。

辅助功能测试

这些测试确保我们的软件可供有视力、听力或认知障碍的人使用，以服务我们的用户并符合法律、合同和法规要求。开源辅助功能扫描器包括 Lighthouse 和 Pa11Y。包括 accessiBe 在内的公司也开始提供 AI 增强的测试和修复工具。

本地化测试

本地化测试对于面向全球受众的软件至关重要。它涉及对产品在特定目标区域内的语言准确性、文化适宜性和功能正确性进行全面评估。这包括验证翻译、根据文化敏感性调整视觉效果，并确保软件在本地格式和法规下正常运行。

性能测试

这些测试模拟工作负载，以评估应用程序在不同条件下的速度、响应能力和稳定性。这些测试有助于识别性能瓶颈，并确保应用程序能够处理预期的流量。对于具有季节性高峰的应用程序，此类测试至关重要，以确保发布能够承受高峰需求。Apache JMeter、Gatling 和 Grafana k6 经常用于性能测试。AI 可以利用性能测试数据来推荐要运行的弹性测试。这些由 AI 驱动的性能测试系统现在可以比传统的基于阈值的方法更准确地检测性能异常。这些系统建立基线性能模式，并识别可能预示即将出现问题的细微偏差。更高级的平台甚至可以通过将测试结果与代码更改和架构图关联起来，查明导致性能下降的具体组件或代码更改。

弹性测试

在现代分布式系统中，生产系统由许多组件组成。唯一可以确定的是，总会有某个地方出问题。弹性测试，也称为混沌测试，评估当软件所依赖的服务发生故障时，软件是否能保持可用。我们将在第 6 章中回到弹性测试。

安全测试

这些测试识别应用程序中可能被攻击者利用的漏洞和弱点。它们有助于确保应用程序的安全性和完整性。动态应用程序安全测试（DAST）是一种特定类型的安全测试，它自动化了渗透测试，检查正在运行的应用程序是否存在安全缺陷。DAST 尝试像恶意用户一样攻击您的应用程序。ZAP 是一种常用的免费工具，而 Veracode 和 Checkmarx 的商业产品也很受欢迎。我们将在第 5 章中回到安全测试。

虽然上面概述的测试类型很常用，但重要的是要注意，软件测试没有一刀切的方法，并且术语在不同组织之间可能有所不同。您选择的具体测试以及如何分类它们将取决于您独特的开发流程、应用程序架构和风险承受能力。

基于意图的功能和端到端测试

传统的自动化功能和端到端测试方法通常严重依赖脚本化测试或简单的录制 - 回放方法。虽然最初方便，但这些测试很快变得脆弱且难以维护，只要发生微小的 UI 更改就会失效。这种脆弱性带来了高昂的维护负担，减慢了开发速度，并经常导致团队完全放弃自动化测试或限制其范围。

一种新兴的 AI 优先测试方法，称为基于意图的测试，旨在克服这些挑战。团队不再明确地编写脚本或手动录制每个测试步骤，而是表达其测试场景的意图，描述他们期望的结果，而不是实现它的精确行动序列。AI 原生测试工具随后通过与您的应用程序交互，像人类用户一样动态生成和执行这些测试。

例如，您无需记录电子商务结账流程中精确的点击和表单输入，只需描述目标：“使用信用卡购买产品。”AI 将自动确定通过您的应用程序的最合适路径，智能地与按钮、表单和工作流进行交互。

一个重要的好处是测试弹性的提高 —— 解决了基于 UI 的测试脆弱的挑战。如果 UI 后来发生变化，AI 会适应新的布局或修改后的交互，从而显著减少维护开销。测试自动化工具多年来一直尝试自动修复测试，采用的技术从跟踪 DOM 对象到实施机器学习。转而理解测试背后的意图，并尝试响应 UI 大修重新生成整个脚本，带来了新的可恢复性水平。

这些工具还可能有助于弥补从专业测试人员转向要求开发人员拥有这些测试的转变。这些工具可以推荐与现有测试相关的额外测试和断言，这可能有助于乐观的开发人员记住

检查边缘情况和不良用户行为。

AI 的高级用例包括将使用传统工具（如 Selenium 和 Playwright）编写的测试迁移到基于意图的测试工具中，以及不仅生成和运行单个测试，还生成和运行整个测试用例。

传统测试与“掏空中间层”方法

在传统软件开发中，测试通常是分门别类的，每种类型都有专门的环境。例如，这可以确保手动用户体验测试永远不会受到并发自动化性能测试的影响。然而，这种隔离是有代价的：测试环境的激增成本高昂，并且管理耗时。当单个新发布必须通过众多阶段时，面对加速的发布节奏和日益增长的应用程序复杂性，这种方法变得越来越不可持续。

当您试图加速发布节奏并且您的应用程序变得更加复杂时，跨多个阶段进行测试变得越来越不可持续，每个阶段都需要一个新的环境。图 4-4 说明了这种分阶段的方法。

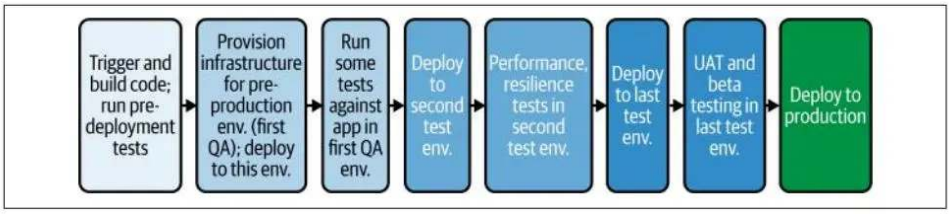

图 4-4. 通过多个预生产环境进行的传统测试

另一方面，一种更现代的测试方法正在挑战这种模式。这种方法有时被称为“掏空中间层”。它不再是在多个环境中进行多个顺序测试，而是减少环境数量，并在其中并发运行测试。这种实践提倡将测试同时“左移”和“右移”。

我们在第 3 章中介绍了左移安全。通过将 SAST、SCA、依赖项扫描和秘密检测移至部署前步骤，我们的示例流水线体现了左移。我们早期就纳入了这些关键测试，使其通过成为代码合并的先决条件。作为合并工作流一部分完成的单元测试和其他早期测试，也代表了左移方法。这有助于更早地发现问题，减少对大量下游测试的需求。

右移方法提倡在实际生产环境中对新版本执行某些类型的测试，这些测试传统上属于后期测试类型。我们不再是从一个或多个预生产环境预置并移动发布，然后使用这些隔离环境进行测试，而是将应用程序直接部署到生产环境并在那里进行验证。例如，负载测试可能难以很好地执行，并且环境可能需要很大。部署到生产环境的一部分，对目标基础设施施加负载，并使用生产可观测性工具测量影响，这可以作为传统负载测试的可行替代方案。图 4-5 阐述了这种方法。

我们可以看到，消除对与生产环境高度相似的预生产环境的需求可以节省成本和维护工

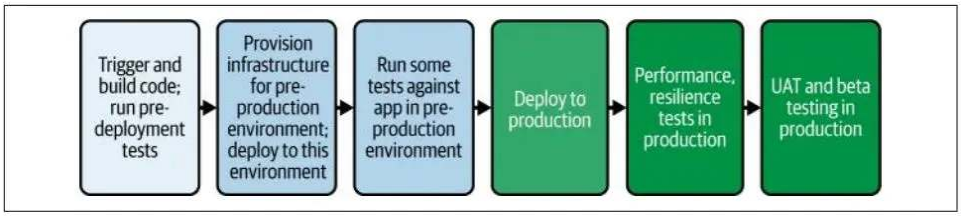

图 4-5. “掏空中间层”的测试方法

作，但如何在生产环境中进行大量测试才能安全呢？右移依赖于新的工具和生产部署实践。凭借先进的流量管理、可观测性工具和容器化技术，许多组织发现这些测试实际上可以在生产环境中执行，且副作用最小。除了显著削减基础设施开支外，这种方法还具有产生更准确结果的优势。我们将在第 7 章中讨论这些新工具和生产部署实践。

掏空中间层优化了测试，是组织为加快交付速度而采取的一种现代策略。通过重新设计我们在环境之间移动软件的方法，我们同样可以加速我们的交付过程。在“环境间的推进”一节中，我们将探讨我们应如何以及为何在环境之间推进发布。

环境间的推进

在上一节中，我们研究了一个典型的交付流程，该流程要求我们的软件经过多个测试阶段，每个测试阶段都在一个单独的预生产环境中进行。在此流程中，我们希望尽可能快且智能地推进我们的发布，这意味着我们的新版本软件应毫无不当延迟地进入下一个环境和阶段。

不同环境之间的版本推进（promotion，即版本在各个交付环境之间的逐步推进/流转，以进入更高阶段的测试或上线）

AI 在此推进过程中开始发挥越来越大的作用，它分析测试结果、性能数据和部署历史记录，以就何时以及如何推进发布做出智能决策。这些系统可以同时评估多个指标，检测可能指示风险的细微模式，并随着时间的推移通过机器学习变得越来越准确。

理想情况下，我们的推进过程很简单：如果一个阶段的测试通过，我们的发布将立即推进到下一个环境，并且该环境已准备好并可用于下一轮测试。推进决策是自动且即时的，仅基于前一个测试阶段是否通过。在实践中，

发布推进，即使是在测试环境之间，在许多交付流程中也成为瓶颈。这可以归因于几个因素：

推进决策由委员会决定

推进决策不是自动化的，需要对测试结果进行小组审查和批准。

推进依赖繁琐的手动步骤

手动干预以触发下一次部署会造成瓶颈。

测试环境数量不足

如果下一个环境正在测试其他版本，则新版本必须等待。

在本节中，我们将探讨解决这些问题的缓解措施。我们将介绍的实践有助于我们将发布从一个预生产环境移动到下一个，也适用于将我们的应用程序推进到生产环境。然而，最终发布到生产环境有一些特殊考虑，我们将在第 7 章中更深入地讨论。

从委员会决策到自动化决策

人类决策，无论是委员会会议还是值得信赖的个人决策，都不可避免地会延迟您的发布从一个阶段推进到下一个阶段。团队成员需要收到提醒，然后花时间分析测试结果，才能做出决定并采取行动。虽然这不一定总是劳动密集型任务，但无疑会减慢速度。

虽然传统自动化依赖简单的通过/失败标准，但 AI 系统提供了更复杂的决策能力。现代 AI 推进引擎可以同时评估数百个指标，超越简单的测试结果，全面分析系统行为。这些系统可能会考虑性能趋势、错误类型、用户影响评估，甚至基于过去的部署模式的代码更改风险级别等因素。通过适当地加权这些因素，AI 可以做出比传统基于规则的方法更细致的决策。

我们的目标是通过自动化发布推进决策来简化这一过程。我们将在第 7 章中详细回顾这一主题。

从手动推进到自动化推进

一旦您自动化了决策过程，构建的实际推进就会变得显著更容易。关键是确保在做出继续决策后立即触发部署到下一个环境，从而消除不必要的等待时间。

如何实现这种自动化取决于您选择的持续交付工具。一些工具提供端到端流水线，带有简单的内置触发器，可实现阶段间的无缝推进。其他工具允许您在当前流水线中将另一个流水线或作业作为步骤调用，提供灵活性，但可能需要更多配置。虽然实现难度各异，但达到这种自动化水平几乎总是可行的。

然而，GitOps 风格的部署在此领域通常提出独特的挑战，正如我们在“通过 GitOps 利用 Git 工作流”中所讨论的。为了执行部署，我们需要自动化对 GitOps 配置的 Git 更改，而不是依赖手动更新。为此，我们通常会直接在我们的 CI/CD 流水线中自动化拉取请求步骤及其批准。我们保留 Git 作为 GitOps 所熟知的单一事实来源，同时自动化我

们发布推进的每个步骤。

例如，想象一个场景，您的流水线已确定一个构建已准备好推进到用户验收测试（UAT）环境。当我们的流水线配置为生成必要的拉取请求、触发任何所需的批准，并在（获得批准后）将更改合并到主分支时，我们的流水线将无缝启动到 UAT 环境的 GitOps 部署。

打破环境瓶颈

在您的交付流程中自动化阶段和环境之间的推进，最后一个挑战是确定您所需的“正确”环境数量。环境过多会因维护其底层基础设施而造成财务负担，而环境过少则会造成瓶颈并延迟发布向交付的进程，因为流程需要等待资源可用。

临时环境为这一困境提供了一个常见的解决方案。这种方法涉及在需要测试时按需创建环境，并在测试完成后迅速将其拆除。在云时代之前，环境创建是一个费力的过程，通常需要数天时间。现在，得益于可编程的云基础设施，环境可以在几分钟内启动和拆除。

基础设施即代码管理（IaCM）工具简化了临时环境。这些专业的 CI/CD 平台使用代码自动化基础设施资源的预置、配置和部署。与专注于应用程序的传统 CI/CD 工具不同，IaCM 工具管理底层基础设施。使用 IaCM 工具，您可以使用声明式代码模板定义所需的基础设施状态，使配置更易于管理、维护和版本控制。

理想情况下，为实现我们“类生产”测试环境的目标，应使用相同的模板创建预生产测试环境和生产环境，仅对变量进行调整。当您的流水线与 IaCM 工具无缝集成时，部署到“测试”阶段会自动触发相应“测试”环境的创建。一旦此环境预置并配置了必要的详细信息，如 IP 地址、密码和其他特定于环境的变量，部署和测试过程即可进行。完成后，IaCM 工具会高效地拆除环境，释放资源。

虽然此策略在一致性、灵活性和成本降低方面提供了显著优势，但需要注意的是，环境创建和销毁过程可能会增加整体测试周期几分钟。因此，对于目标是极快速交付周期（例如以分钟计）的流水线，临时环境可能不是理想的解决方案。然而，对于以小时、天或周为单位的交付周期，临时环境提供了一种强大的方式来打破瓶颈、提高一致性并优化基础设施成本。

总结

在本章中，我们继续探讨交付过程，重点关注持续集成之后的持续交付步骤。这些主要是测试步骤，我们回顾了对验证软件所有方面都至关重要的测试类型。我们讨论了可靠

和可预测的预生产环境对测试的重要性，以及实现这些环境的最佳实践。通过自动化发布在测试阶段之间推进的所有方面，包括推进决策，我们可以显著加速软件的交付。

完成测试后，只剩下最后一步才能将我们最新的软件版本交付给用户：实际部署到生产环境。我们将在第 7 章中回到这一步。在此之前，我们将在接下来的几章中讨论如何加强我们的发布，使其更安全、更具弹性和更可靠。

第 5 章：保护应用程序和软件供应链

在本书中，我们一路探索了从软件配置管理 (SCM) 到持续集成和持续交付的交付过程，并在此过程中探讨了安全工具和实践。我们讨论了现代工具中基于角色的访问控制 (RBAC) 和策略即代码 (PaC) 治理如何帮助保护您的代码仓库和管道，并提到了早期安全测试在持续集成中的作用。我们还研究了动态测试以发现应用程序中的运行时漏洞。这些都只是对安全的浅尝辄止。

在本章中，我们将把安全放在首要位置，在一个网络攻击日益频繁和复杂的世界中，给予它应有的关注。备受瞩目的数据泄露事件屡见报端，全球范围内的法规日益收紧，客户也越来越多地根据供应商的安全态势来评估他们。

随着发布周期从数月缩短到数天，传统的将安全作为生产前最后一道关卡的模式已难以为继。取而代之的是，我们已将安全责任“左移”至开发人员，他们现在必须将安全实践整合到日常工作流程中。那些并非安全专家的开发人员如今承担着前所未有的安全责任。

人工智能有望缓解这种紧张局面。AI 驱动的安全工具正在提高检测准确性，大幅减少浪费开发人员时间的误报，甚至能自动生成修复代码。AI 不仅仅是将安全负担左移，它还帮助分担了这一负担，为开发人员提供了专家级的安全指导，而无需他们自己成为安全专家。

本章将探讨人工智能原生软件交付的演进如何改变我们处理安全问题的方式 —— 它不仅仅是增加更多的工具或流程，而是从根本上改变了我们识别、优先处理和修复安全问题的方式。我们将深入探讨软件供应链安全的重要性，它保护了从初始代码到最终产品中软件构建和交付所涉及的工具、流程和人员。这是一个关键问题，因为现代软件严重依赖于相互连接的组件，每个组件都可能存在被恶意行为者利用的潜在漏洞。

理解供应链问题并学会以安全视角评估您的软件开发生命周期 (SDLC) 将使您能够实施强大的安全措施，更好地保护您的应用程序、数据和组织的声誉。

现代应用程序与网络威胁态势

构建和部署现代软件应用程序严重依赖于分布式且复杂的软件供应链。这些供应链通常包含庞大的代码仓库网络、开源依赖项、第三方组件、制品仓库以及 CI/CD 管道。尽管这种相互连接性促进了创新并加速了我们的开发周期，但它也贯穿始终地引入了安全风险。不断扩大的攻击面以及漏洞在整个供应链中传播的可能性，使得我们的软件供应链成为恶意行为者的主要目标。

在本节中，我们将探讨这些威胁，并了解管理软件供应链的监管合规框架如何演变以应对这些威胁。最后，我们将审视新的合规要求如何影响您的组织。

日益增长的软件供应链攻击威胁

软件供应链涵盖了所有参与创建和交付软件的人员、流程和工具。它跨越软件开发的整个生命周期，从最初的代码创建到部署和维护。这是一个复杂的生态系统，其中每个要素都在最终产品中发挥着关键作用。

软件供应链主要由两个部分组成：应用程序和 DevOps 工具链，如 图 5-1 所示。

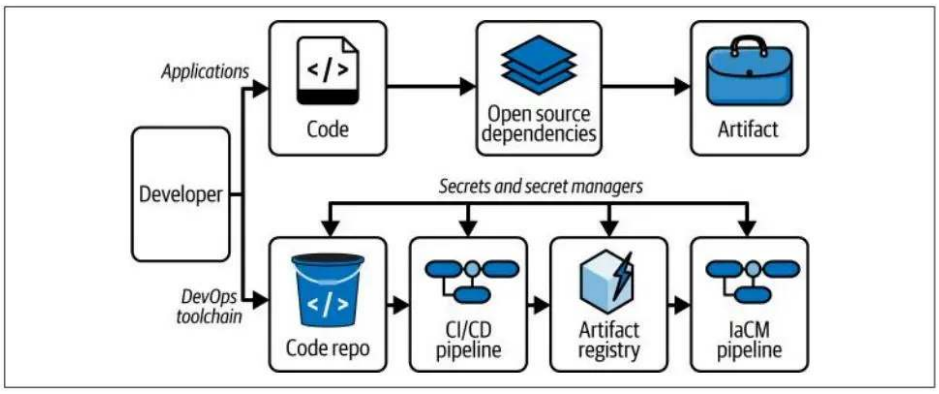

图 5-1. 软件供应链

软件供应链中的应用程序风险

此处的“应用程序”指的是您软件的所有组成部分，包括您的专有源代码；像库、框架和模块这样的开源依赖项；以及在开发过程中产生的软件制品。

根据 Synopsys 发布的《2024 年开源安全与风险分析报告》，96% 的代码库包含开源组件。重要的是要记住，我们的组织有责任保护我们使用的开源组件，就像保护内部开发的代码一样。由于开源使用如此广泛，我们不应惊讶于应用程序中发现的漏洞有 80% 以上来自开源软件 (OSS) 依赖项。2021 年在广泛使用的 Java 日志库 Log4j 中发现的一

个漏洞就是开源引入威胁的一个例子。该漏洞允许攻击者通过简单地向应用程序日志发送一个特殊构造的字符串，从而在受影响的系统上远程执行代码。由于 Log4j 在应用程序和服务中的广泛使用，该漏洞利用极其危险，导致了大规模的抢修和缓解工作。

在广泛使用的 XZ Utils 数据压缩工具中发现后门是另一个例子。XZ Utils 与许多开源项目一样，由资源有限的志愿者维护，难以解决安全问题。一名受信任的贡献者被发现植入了一个后门，该后门允许攻击者获取运行使用该工具构建的软件系统的管理员权限。该工具存在于大多数 Linux 发行版中，幸运的是在广泛部署到生产系统之前被发现。

应用程序供应链中的另一个新兴威胁是利用人工智能编码助手“幻觉”现象。当 AI 模型虚构包名，推荐不存在的库或错误的包标识符时，它们就为攻击者创造了机会。恶意行为者可以监控流行的 AI 编码助手，寻找此类“幻觉”，然后将这些虚构的包名注册到公共仓库中。当开发人员试图使用这些不存在但 AI 推荐的包时，他们会在不知不觉中安装恶意代码。这种“幻觉抢注”攻击向量已在野外被观察到，研究人员发现常见的编码助手经常推荐不存在的包。

软件供应链中的 DevOps 风险

DevOps 工具链包括用于自动化软件构建、测试和部署的工具和流程集合。这涵盖了代码仓库、CI/CD 工具和管道、制品注册表以及其他简化开发过程的工具，例如 GitOps 和 IaCM 工具。

SolarWinds 攻击事件是 DevOps 工具链被攻陷后如何被利用来传播恶意代码的一个鲜明例子。在这起复杂的攻击中，威胁行为者渗透了 SolarWinds Orion 软件构建系统，将恶意代码注入合法的软件更新中。这些被污染的更新随后分发给了一万八千名 SolarWinds 客户，使得攻击者能够广泛访问他们的网络。此事件凸显了攻击者利用 DevOps 管道固有的信任和自动化功能来大规模分发恶意软件的潜力，将一次例行的软件更新变成了毁灭性的网络攻击。

2021 年的 Codecov 供应链攻击是另一起工具链安全漏洞事件，影响了数千个组织。恶意行为者修改了 Codecov Bash Uploader 脚本（客户用于上传代码覆盖率数据的工具）。这一修改使得攻击者能够从 Codecov 客户的持续集成环境中窃取敏感信息，例如令牌、密钥和凭据。该漏洞在两个多月内未被发现，可能暴露了客户持续集成环境中存储的敏感数据。

日益增长的威胁

软件供应链攻击不会消失。Gartner 研究报告预测，到 2025 年，全球 45% 的组织将经历软件供应链攻击。代码中的安全缺陷、第三方库或您管道中的某个工具都可能产生连锁反应，从而危及整个软件产品。保护软件供应链不仅是为了保护单个组件，更是为了确保整个开发和交付过程的完整性和安全性。

适用于软件供应链的监管合规框架

鉴于日益增长的威胁，各国政府和监管机构已通过制定法规来应对这些挑战，旨在建立最佳实践、促进透明度，并要求组织采取主动措施来保护其软件供应链。其中一些最重要的合规和监管框架包括：

美国第 14028 号行政命令：《改善国家网络安全》

该行政命令于 2021 年发布，要求联邦机构及其软件供应商加强其软件供应链安全实践。它强调使用安全的软件开发实践、漏洞披露和事件响应。

欧盟《网络和信息安全指令 2》（NIS2 指令）该指令旨在欧盟范围内建立高水平的通用网络安全标准。它包含了关于软件供应链安全的条款，要求组织评估和管理与软件组件及第三方依赖项相关的风险。

NIST SP 800-218：《安全软件开发框架》(SSDF)

这份美国国家标准与技术研究院的出版物为将安全集成到软件开发生命周期 (SDLC) 中提供了指导，包括供应链风险管理。它提供了一个全面的安全软件开发实践框架。

ISO/IEC 27036-2:2023

该标准提供了管理与供应商和供应链相关的信息安全风险的指南。它涵盖了供应商选择、合同管理和绩效监控等多个方面。

支付卡行业数据安全标准 (PCI DSS)

尽管 PCI DSS 不完全专注于软件供应链，但它要求处理支付卡数据的组织实施安全的软件开发实践，其中包括管理供应链风险。

网络韧性法案 (CRA)

这项拟议中的欧盟法规旨在增强数字产品和服务的网络安全。它包括了对漏洞处理、安全更新、软件物料清单 (SBOM) 以及在发现漏洞后 24 小时内报告已活跃利用漏洞的要求。

此外，质量体系法规 (QSR) (21 CFR Part 820) 和通用数据保护条例 (GDPR) 是间接影响软件供应链问题的软件实践监管框架。QSR 强制要求严格的控制和流程，以确保医疗设备的安全性和有效性，其中包括软件组件。这要求制造商对集成到其设备中的软件进行验证和控制。同样，GDPR 对个人数据保护的严格要求，使得组织必须实施强大的技术和组织措施，这可能扩展到软件及其供应链的安全，特别是当它处理个人数据时。

这些框架和法规有助于构建一个更安全、更具韧性的软件生态系统，使企业和消费者都受益。然而，日益增加的复杂性可能会影响开发团队。理解这些要求并将其整合到您的

流程中对于成功合规至关重要。

通过左移保护现代应用程序

面对动机强烈的黑客，我们传统的”等到最后”的安全方法已远远不够。这些措施不仅无法提供我们所需的保护，而且传统的安全测试还会减缓软件交付速度。为了保护现代应用程序，组织必须使用专为现代 DevOps 工作流设计的工具和实践。在本节中，我们将探讨组织在实施安全实践时面临的挑战。在 第 3 章 中，我们简要提到了左移安全，即在开发的最早阶段实施安全实践。我们将探讨如何利用这种方法来缓解风险，以及实施左移安全和以开发人员友好的方式管理漏洞的最佳实践。

对开发人员友好的左移安全的需求

您必须在软件开发周期的每个可能阶段积极处理和测试安全问题，而不是等到最后才测试应用程序的安全性。这种方法不仅通过避免后期对软件代码进行大量返工来节省时间和精力，而且还提高了最终产品的整体安全性和效率。图 5-2 对比了左移安全方法与传统应用程序安全方法。

重要的是要指出，有效的左移安全不仅仅是在交付过程的早期执行安全测试。虽然这可能有助于开发人员避免在数天或数周后返回代码时产生的上下文切换成本，但它最终并未节省工作量。真正有效的实施需要选择能够与您的 CI/CD 管道无缝集成的安全工具。这些工具不仅应识别漏洞，还应根据严重程度对其进行优先级排序，并提供可操作的见解。您选择的工具应标准化并去除重复的发现，以帮助开发人员避免警报疲劳，并专注于最关键的风险。这种集成方法确保安全在开发过程中处于绝对核心地位。

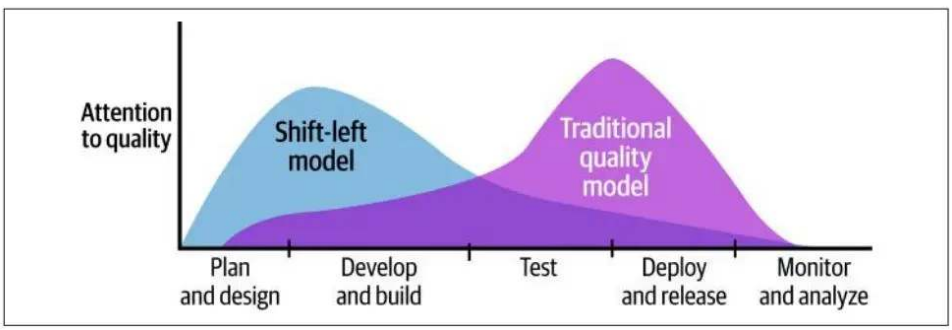

图 5-2. 左移方法与传统测试方法的对比

应用程序安全扫描器

有许多可用的扫描器和工具用于安全测试和分析，其中许多现在都增强了 AI 能力。让我们来看看这些扫描器和工具最常见的类别：

软件成分分析 (SCA)

这种类型的扫描器通过分析软件物料清单 (SBOM) 来识别第三方组件和依赖项中的漏洞，以检测库和框架中的已知漏洞。我们将在本章后面探讨 SBOM。SCA 工具在漏洞可被利用或被利用的可能性方面具有显著的机器学习能力。Snyk 是 SCA 扫描器的一个流行示例。

静态应用程序安全测试 (SAST)

SAST 工具通过扫描代码中指示漏洞的模式（例如 SQL 注入、XSS 和缓冲区溢出）来分析源代码中的潜在漏洞，而无需执行应用程序。AI 正在增强 SAST，以减少误报的发生，从而节省工程师的时间。SonarQube、Checkmarx 和 Fortify 都是 SAST 工具的例子。

容器扫描

这种类型的扫描通过分析容器镜像的内容来识别容器镜像及其依赖项中的漏洞和配置错误。

秘密检测扫描

这种类型的扫描检测代码仓库和配置文件中的敏感信息，例如 API 密钥、密码和令牌。借助 AI，秘密检测工具在检测混淆的秘密以及区分实际凭据和测试数据方面表现更佳，从而减少了误报和相关的手动工作。

动态应用程序安全测试 (DAST)

这种测试方法通过模拟外部攻击来分析运行中的应用程序以识别漏洞。它像真实用户一样与应用程序交互，检测注入缺陷、认证问题和配置错误等问题，而无需访问源代码。AI 增强的 DAST 工具根据应用程序行为而非固定模式生成测试用例。它们试图自动验证其发现，以解决误报问题。

基础设施即代码 (IaC) 扫描

这种类型的扫描分析 IaC 文件，以在部署前识别安全漏洞、错误配置和合规性问题。

这些类型的扫描器与左移方法一致，被集成到软件开发管道的早期阶段。秘密扫描是一种推荐的安全实践，它自动识别并提醒用户代码仓库和其他数据源中的敏感信息。

这从一开始就防止了敏感信息被纳入代码库。SCA 工具也通常在代码提交后、构建之

前，被集成到管道的早期阶段。SAST 扫描可以是构建阶段的一部分。容器扫描通常在容器镜像构建之后、部署之前集成。

通过在您的开发管道中整合 SCA、SAST、容器扫描、秘密扫描、DAST 和 IaC 扫描，您可以有效地实施左移安全，并主动保护您的应用程序免受漏洞侵害。

您的测试工具识别出的每一个问题都必须进行分类。审查问题、判断其是否真实，然后进行修复，这都会产生代价。误报，即被报告但并非真实存在的问题，是一个严重的问题。它们浪费了审查人员的时间，耗费了其他安全工作和创新的资源。此外，它们通过“狼来了”效应降低了工程师对安全发现的信任，并可能减缓对其他真实问题的响应时间。有鉴于此，在许多扫描工具中，减少误报数量成为 AI 的一个关键优先事项也就不足为奇了。

分类问题因涉及的扫描器数量众多而加剧，这些扫描器可能以不同的方式发现相同的问题。在某些组织中，甚至可能在相同的代码库上使用多个 SAST 工具。在这种环境中，可以使用安全测试编排层来去除重复项并将发现结果规范化为一个单一的、可管理的列表。在 AI 原生环境中，AI/ML 在模式匹配以及减少开发人员繁重工作方面发挥着作用。

所有这些类型的扫描器都将检测到问题，并且需要进行修复。安全工具正越来越多地通过专业的 AI 编码助手提供自动化或半自动化的修复功能，从而为开发人员简化这一过程。

保护软件供应链

在本节中，我们将探讨当今软件供应链固有的常见安全风险。我们将审视与代码仓库、CI/CD 管道、制品仓库、开源依赖项以及支撑您软件开发过程的基础设施相关的风险。人工智能正在通过识别复杂供应链中手动大规模监控不可能发现的模式和异常，从而改变组织检测和响应这些风险的方式。我们将探讨可用于评估您的工具链安全性的各种框架和基准。在本节结束时，您将对潜在威胁以及如何缓解这些威胁有更深入的了解。

现代软件供应链的复杂性为人工智能创造了一个理想的用例。AI 系统可以持续监控跨仓库、构建系统和部署的异常模式。例如，机器学习模型可以检测可能表明开发人员帐户被入侵的异常提交模式，识别表明潜在供应链攻击的可疑包行为，或者发现可能导致安全漏洞的配置漂移。这些 AI 能力在相互连接的组件之间提供了前所未有的可见性和保护。

识别 CI/CD 的十大安全风险

开放全球应用程序安全项目 (OWASP) 是一个致力于提升软件安全的领先组织，它已识别出 CI/CD 的十大安全风险。如下列表所示，威胁范围多样。理解这些风险并实施推荐的缓解策略将有助于您保护和加强您的 CI/CD 生态系统：

不充分的流量控制机制

CI/CD 管道中不充分的流量控制机制可能会被获得管道访问权限的攻击者利用。通过绕过必要的审查和批准，恶意代码或制品可能会被推送到管道中，从而可能到达生产环境，造成严重后果。

身份和访问管理不足

在各种系统中管理众多身份的复杂性，加上账户权限过度宽松的趋势，可能导致泄露。如果任何用户账户被攻陷，攻击者可能会获得广泛的访问权限，甚至可能到达生产环境。

依赖链滥用

依赖链滥用是指利用您的开发和构建系统获取代码依赖项的方式中的漏洞。当这些系统被欺骗性地获取并执行恶意软件包而不是合法软件包时，就可能发生这种情况。攻击者通过发布与内部软件包同名的恶意软件包（依赖混淆）、劫持维护者账户（依赖劫持）或利用拼写错误（错字抢注）来欺骗开发人员下载他们的软件包，从而进行利用。

投毒管道执行

这是一种网络攻击，恶意代码被注入到 CI/CD 管道中，通常通过被攻陷的源代码控制系统。受污染的代码随后可以在管道内执行，可能赋予攻击者与构建作业相同的访问权限和特权。攻击者可以操纵构建配置文件或管道依赖的其他文件，导致凭据窃取、数据泄露或恶意制品部署等行为。

基于管道的访问控制不足

当管道执行节点对资源和系统拥有过度访问权限时，就会产生风险。攻击者可以利用这一点在管道内运行恶意代码，滥用授予管道的权限，在 CI/CD 系统内部或外部横向移动。

凭证管理不当

在凭据广泛用于不同系统和上下文的环境中，凭据管理不当是一个重大风险。例如，意外推送包含凭据的代码、在构建和部署过程中不安全地使用凭据、未轮换的凭据，以及凭据被打印到控制台输出或存储在容器镜像中。

不安全的系统配置

不安全的系统配置是常见的漏洞，原因在于典型工具链中存在众多系统和供应商。诸如过时软件、过度宽松的访问控制或不安全默认设置等错误配置，很容易被攻击者利用，从而获得未经授权的访问、操纵 CI/CD 流程，甚至危及生产环境。

未经治理的第三方服务使用

CI/CD 管道中的第三方服务虽然方便且对开发有价值，但很容易被授予对敏感资源的过度访问权限，从而有效扩大了组织的安全攻击面。这种治理和可见性的缺失使得难以维持适当的访问控制，一旦这些第三方服务中的任何一个遭到入侵，组织就容易受到攻击。

制品完整性验证不当

由于软件交付涉及多个阶段和来源，恶意行为者可能会在不发出警报的情况下篡改制品。如果未能检测到，这些受损制品可能会流经管道并最终部署到生产环境中，执行恶意代码并危及系统。

日志记录和可见性不足

如果没有健全的日志记录，您将对开发管道中发生的恶意活动视而不见，从而难以及时检测和响应攻击。

理解这些风险并实施推荐的缓解策略是构建安全且具有弹性的 CI/CD 生态系统的关键。

识别十大开源软件 (OSS) 风险

OSS 依赖项的使用无处不在，因此组织必须应对其带来的安全和合规风险。我们之前提到了两个例子。第一个是 Log4j 威胁，导致数千个系统受到影响。第二个是 XZ Utils 的例子，虽然发现得早，但它说明了恶意行为者如何通过入侵 OSS 组件来造成严重破坏。

常见漏洞和披露 (CVE) 是组织可用来识别已知安全问题并采取措施缓解这些问题的机制之一。CVE 监控工具可自动扫描您的软件并提醒您潜在风险。虽然勤奋的监控可以帮助您消除所用 OSS 中的已知威胁，但这并不能保证您的 OSS 组件是真正安全的。未维护的组件或过时的依赖项也会带来风险，而且由于 OSS 包会引入数十个依赖项，因此管理起来可能非常复杂。

尽管 CVE 管理有助于应对已知威胁，但还有其他类别的威胁需要应对。OWASP 基金会创建了以下十大列表，以更全面地涵盖您的组织需要防范的 OSS 风险：

已知漏洞

开源组件可能包含公开披露的安全缺陷，通常通过 CVE 或其他渠道。这些漏洞如果能在您的软件中被利用，可能会损害您系统的机密性、完整性或可用性。

合法软件包被入侵

攻击者可能通过劫持账户或利用漏洞，将恶意代码注入到现有项目或分发基础设施中。这可能导致在最终用户或组织系统上执行代码，从而危及机密性、完整性、和可用性。

名称混淆攻击

名称混淆攻击涉及恶意行为者创建与合法组件名称高度相似的组件，旨在欺骗用户安装它们。这些攻击可能导致在用户和组织系统上执行有害代码，从而损害机密性、完整性和可用性。

未维护的软件

由于未维护的 OSS 组件不再积极开发或支持，新漏洞的补丁可能不可用。这种情况可能导致需要自行创建补丁的下游开发人员的工作量增加和解决时间延长。

过时软件

在您的项目中使用过时的软件组件会带来重大挑战。这会使紧急更新变得困难，特别是当您使用的版本中发现漏洞时。旧版本在安全问题方面的测试可能也不如新版本彻底。

未跟踪的依赖项

未跟踪的依赖项可能会在开发人员不知情的情况下引入漏洞。这些依赖项可能由于 SBOM 不完整、SCA 工具功能有限或手动安装方法而被遗漏。

许可风险

开源组件的许可可能与预期用途不兼容、违反法律要求或完全没有许可。在没有许可的情况下使用组件或未能遵守许可条款可能导致法律后果。

不成熟的软件

不成熟的开源项目，缺乏标准版本控制、测试或文档等最佳实践，可能会给您的软件带来操作风险。这种不成熟可能导致意外行为和开发工作量增加，同时伴随漏洞。

未经批准的变更

未经批准的软件组件更改可能导致软件构建的完整性和可重现性受到损害。

过小/过大的依赖项

开源组件在大小和功能上可能差异很大，从而带来安全风险。小型组件提供的功能最少，但由于它们依赖于上游项目，仍然可能引入重大风险。大型组件虽然可能提供更多功能，但由于未使用功能和依赖项，其攻击面可能更大。

在下一节中，我们将探讨一个可以帮助应对这些风险的框架 —— SLSA。

通过软件制品供应链级别 (SLSA) 确保完整性

显然，开源软件 (OSS) 的风险众多。在我们自己的软件中利用 OSS 或任何第三方组件之前，我们必须问：这个软件是谁编写的？它是否是使用我们可以信任的工具和平台构建和发布的？它带来了哪些依赖项？它是否符合对我们重要的监管要求？

软件制品供应链级别 (SLSA，发音为“salsa”) 是一个框架，它提供了一种结构化的方法来回答这些问题。SLSA 旨在增强软件制品在整个软件供应链中的完整性。它增强了软件供应链的安全性，并有助于解决我们已讨论过的 OSS 威胁。

类似于实物证据的保管链，SLSA 强调了在软件制品整个生命周期中跟踪和验证其完整性的重要性。在本节中，我们将深入探讨 SLSA，并提供如何遵守其要求以保护您的软件免受潜在威胁的指导。

SLSA 概述

SLSA 是由开源安全基金会推动的一个开源项目。凭借其对实际实施和可衡量安全改进的关注，SLSA 获得了显著的关注。

SLSA 为 OSS 和供应商提供的软件的提供商和消费者带来了益处。在您的组织内部，您可以使用 SLSA 来帮助保护您的软件开发过程免受内部篡改。这确保您部署到生产环境的代码是您已构建、测试并批准的代码。

对于软件消费者而言，SLSA 提供了验证 OSS 真伪和完整性的机制。包注册表能够使用 SLSA 来保证上传的 OSS 包是基于合法仓库中的源代码构建的。作为 OSS 消费者，从受信任的注册表获取资源可确保您下载的包是有效的。此外，您可以要求您的供应商遵循 SLSA 原则。

验证信誉良好的第三方审计机构的供应商 SLSA 认证可以提供额外的信心。

SLSA 定义了一个分层框架，允许组织逐步增强其软件供应链的安全性。级别代表了对篡改的日益增强的保证和保护。一个没有采取任何保护措施的组织被视为处于 0 级。

SLSA 1 级是基础。1 级要求生成基本的溯源信息。此信息应详细说明构建过程、描述依赖项并提供源代码位置。1 级是组织开始其软件供应链安全之旅的起点。消费者可以使

用此信息来决定与软件相关的风险。

2 级在 1 级的基础上，引入了更强的构建要求。您的构建环境必须是隔离且受控的。该级别还强制要求对制品进行签名以进行完整性验证，从而防止篡改。

最后，3 级要求源代码溯源和构建可重现性。溯源必须可审计，并且其完整性必须得到保证。

表 5-1 总结了 SLSA 1.0 定义的三个级别的要求。

实施方 (Implementer)	**要求 (Requirement)**	**等级说明 (Degree)**	**L1**	**L2**	**L3**
Producer（生产方）	选择合适的构建平台	✓	✓	✓	
	遵循一致的构建流程	✓	✓	✓	
	分发构建来源证明(provenance)	✓	✓	✓	
Build platform（构建平台）	来源证明生成能力存在	Exists（存在）	✓	✓	✓
	来源证明生成被保证为真实	Authentic（真实）		✓	✓
	来源证明不可伪造	Unforgeable（不可伪造）			✓

实施方 (Implementer)	要求 (Requirement)	等级说明 (Degree)	L1	L2	L3
	隔离强度	Hosted（托管）		✓	
	隔离强度	Isolated（完全隔离）			✓

使用 SLSA 确保完整性

以下原则指导了 SLSA 框架的设计决策：

信任少数平台；聚焦于制品

将信任延伸到少数核心平台，例如构建和打包工具，然后自动化验证这些平台生成的制品。例如，您受信任的构建平台为其构建的每个制品生成并签署溯源证明。下游平台随后验证由公钥签署的溯源，以自动确定制品是否符合 SLSA 级别。

将软件追溯到源代码，而非个人

在最终软件制品与其原始源代码之间建立直接且可验证的链接。这种方法与信任对包注册表具有写入权限的个人以及信任代码本身不可变和可分析的性质形成对比。通过建立直接链接，组织可以显著降低恶意代码注入或未经授权修改的风险。

偏好证明而非推断

依赖制品来源的直接证据，而不是基于对中间构建系统或其他系统的了解来推断制品的信任度。SLSA 并非推断完整性，而是要求明确证明制品的来源。这需要制品构建过程的具体证据。

在 SLSA 1.0 中，构建平台是确保制品完整性的核心。构建平台是指负责编译、打包和准备软件以供分发的系统。一个健壮的构建平台对于实现更高的 SLSA 级别至关重要。您选择的系统应支持隔离构建，这意味着每次构建都会创建新的基础设施，并且在构建运行后，基础设施会被删除。此外，系统应强制执行非特权、容器化的持续集成步骤，这些步骤不使用卷挂载。这可以防止在符合 SLSA 规范的情况下访问溯源密钥信息。通过一个强化的构建系统，您可以确保恶意行为者无法篡改您的构建。

除了选择一个能够保证制品完整性的构建平台之外，您的系统还应生成和分发证明（数字签名记录），以表明您的软件符合您所需的 SLSA 构建级别。SLSA 溯源证明是加密签名，提供关于软件制品来源和构建过程的可验证证据。它们充当数字护照，确保制品的完整性和真实性。

考虑一个使用 CI/CD 管道构建的容器镜像。该镜像的 SLSA 溯源证明可能包括以下信息：

构建者

用于构建镜像的 CI/CD 平台（例如，GitHub Actions、GitLab CI/CD）

调用

用于创建镜像的特定构建配置或脚本

材料

构建过程中使用的源代码仓库、依赖项和其他输入

主体

制品本身，由其唯一摘要（哈希值）标识

签名

由受信任实体生成的加密签名，验证证明的真实性和完整性

为了验证 SLSA 溯源证明，组织可以使用 SLSA 验证服务等工具。该服务验证证明的真实性，对照受信任签署者的公钥检查签名，并确保证明符合 SLSA 规范。

为了实现最大的安全性，SLSA 建议由构建平台而非个人开发人员生成溯源。如果您的组织没有使用构建平台，请考虑采用一个支持 SLSA 的平台。对于第三方平台，请检查其兼容性并在需要时请求 SLSA 支持。如果您维护自己的构建平台，请添加 SLSA 溯源生成功能。

同样，软件包生态系统应将 SLSA 溯源与软件包一同分发，将证明嵌入软件包内部或作为单独的元数据文件提供。如果您的组织使用第三方生态系统，请咨询 SLSA 支持并遵循其指南。对于直接分发，请在您的软件包制品中包含 SLSA 溯源。

通过利用 SLSA 溯源证明，您的组织可以对软件制品的真实性和完整性充满信心，并降低供应链攻击的风险。

超越 SLSA 增强供应链安全

尽管 SLSA 为构建完整性提供了一个出色的框架，但它主要侧重于制品溯源和构建系统完整性。为了应对 OWASP CI/CD 十大风险中识别出的所有供应链风险，组织需要额外的安全措施。

全面的供应链安全策略应包括：

持续行为监控

现代交付和安全平台，由人工智能和机器学习提供支持，正在不断改进，以检测跨仓库、构建系统和部署管道的异常活动。这些系统建立正常行为基线，并标记可能表明被入侵的偏差。Datadog CI 和 GitGuardian 等监控和安全工具是当今流行的选择。

高级依赖项分析

除了基本的漏洞扫描，智能分析工具可以评估包行为、代码模式和维护者活动趋势，以在恶意依赖项被公开报告之前识别它们。AI 驱动的系统可以通过传统扫描器无法实现的方式分析代码语义和行为，从而检测到细微的入侵迹象，帮助抵御复杂的供应链攻击，如依赖混淆或错字抢注。

自动化策略强制执行

在整个管道中实施自动化策略防护，以强制执行超出构建完整性范围的安全要求。这些系统可防止过度许可的访问、阻止危险配置，并确保正确的秘密管理 —— 解决 SLSA 未完全涵盖的风险，如 RBAC 不足和凭据管理不当。目前，在交付平台中广泛实施策略仍不均衡。展望未来，这是一种强有力的方法，将受益于 AI，使其在适应不断变化的威胁方面更具适应性，并在 PaC 场景中通过 AI 代码生成协助快速创建策略。

供应链风险预测

预测分析和 AI 模型分析历史漏洞趋势和新兴威胁情报，以突出供应链中构成更高潜在风险的组件，帮助团队在漏洞成为关键问题之前主动解决它们。通过分析数千个项目和依赖项的模式，这些系统在安全事件发生之前识别您环境中的风险因素，从而实现对脆弱区域的主动加固。

通过将这些 AI 增强的能力与 SLSA 的构建完整性重点相结合，组织可以创建一个深度防御方法，解决供应链的全部风险。这种全面的策略不仅保护构建过程，还保护从开发到部署的整个软件交付管道。

解决 AI 生成的依赖风险

随着组织越来越多地采用 AI 编码助手，一种新的供应链风险已经出现：AI 幻觉抢注。当攻击者注册 AI 工具通过“幻觉”错误建议的包名时，就会发生这种情况，从而为恶意代码注入创造了一个途径。

尽管核心 SLSA 框架为传统供应链攻击提供了显著保护，但使用 AI 编码工具的组织应实施额外的保障措施：

已验证的注册表策略

配置包管理器，使其仅从官方审查的注册表和包含已知良好包的私有仓库拉取。这可以防止开发人员无意中从不受信任的来源安装包，即使 AI 助手建议了它们。

包年龄和流行度检查

实施工具，自动根据最小下载量和已建立的历史指标来验证推荐的软件包。使用量极少的新软件包应触发额外的审查。

AI 置信度验证

当使用提供推荐置信度分数的 AI 编码助手时，实施流程来标记低置信度的包建议，以便针对权威来源进行手动验证。

预安装验证

在您的开发环境中添加自动化检查，在允许将依赖项添加到项目文件之前，验证包在受信任仓库中的存在性和溯源信息。

这些额外的控制措施，结合 SLSA 实践和全面的 SBOM，创建了一种深度防御方法，既能抵御传统供应链攻击，又能防御新兴的 AI 辅助威胁。通过解决 AI 在依赖项选择过程中引入的特定风险，组织可以安全地利用 AI 编码助手，同时保持供应链的完整性。除了 AI 置信度验证外，这些实践中的每一项都对其他基于包的攻击（如错字抢注）有所帮助。

通过软件物料清单 (SBOM) 应对零日漏洞

在第一节中，我们回顾了 Log4j 漏洞利用事件，该事件允许攻击者通过利用日志消息中的特定模式远程执行任意代码，从而导致了大规模的数据泄露、勒索软件攻击以及对关键服务的干扰或中断。这是一个零日漏洞利用的例子，是 insidious 威胁类型之一，因为它利用了软件供应商未知的漏洞，使得攻击者在任何防御措施到位之前就拥有显著优势。在本节中，我们将探讨 SBOM 如何作为对抗此类漏洞的重要工具。SBOM 提供了软件制品中使用的所有组件和依赖项的详细清单。我们将研究 SBOM 的组成和特性，以及它们在 SDLC 中的管理方式。

尽管依赖管理工具和包管理器已经存在多年，用于跟踪和管理软件组件，但自 2018 年以来，SBOM 取得了显著进展。包括美国国家电信和信息管理局 (NTIA) 多方利益相关

者流程在内的协作努力，已经为 SBOM 制定了最佳实践和建议。这项协作努力汇集了行业专家、政府机构和学术界人士，共同定义了 SBOM 的生成、共享和消费标准及指南。

因此，SBOM 已成为一个关键的组成部分。事实上，Linux 基金会最近的研究发现，2022 年有 78% 的组织正在生产或消费 SBOM，比上一年增长了 66%。

在为您的软件创建 SBOM 时，您有两个标准可供选择：

CycloneDX

CycloneDX 项目已成为 SBOM 的领先标准，它提供了一种机器可读的格式来表示软件组件、依赖项及其关系。CycloneDX 已经获得了广泛的采用和来自各种组织的支持。

SPDX

软件包数据交换 (SPDX) 是另一种流行的 SBOM 标准，由 Linux 基金会赞助并在 ISO/IEC 5962 国际标准中编纂。它提供了一种灵活且可扩展的格式来表示软件组件。SPDX 在开源社区中已广泛使用了多年。

SPDX 是一种更成熟、范围更广的格式，不仅包含组件信息，还包括 SBOM 本身的元数据，例如其创建者和创建日期。它特别适合管理开源软件 (OSS) 许可和共享包信息。

CycloneDX 是一种较新的格式，提供了更结构化和机器可读的方法，重点是提供软件组件及其关系的详细信息。CycloneDX 通常因其灵活性和适应性而受到青睐，使其适用于各种用例。您的特定用例可能会决定您采用哪个标准；您为软件供应链安全管理选择的工具和流程应能够支持这两种标准。

无论您选择哪种具体格式，评估 SBOM 质量的因素都是相同的。NTIA 已经开发了一套 SBOM 应该包含的最小元素，以提供关于软件组件及其依赖项的基本信息。确保这些元素的存在将促进在各种工具和平台之间对 SBOM 的有效分析，以及对底层 SPDX 或 CycloneDX 规范的遵守。

通过提供全面的组件清单，SBOM 提供了透明度和可追溯性，这有助于确保符合您组织的安全策略和法律要求。策略即代码 (PaC) 框架可以利用 SBOM 来自动化这种合规性。使用 PaC，您可以使用代码定义安全策略，这些策略可以像代码一样进行管理和版本控制。然后可以将这些策略应用于 SBOM，确保软件组件遵守组织对开源软件 (OSS) 的安全标准。自动化合规性降低了人为错误的风险并提高了效率。

例如，您的组织可以定义一项策略，只允许使用具有宽松许可（例如 MIT、Apache 许可证 2.0）的开源软件 (OSS) 组件，以确保与组织现有软件组合的兼容性并避免潜在的法律问题。

您可以定义一项策略，自动拒绝已知漏洞严重性高于某个阈值的 OSS 组件。或者，您

可以建立评估 OSS 供应商声誉和可信度的标准。这可以包括供应商规模、安全实践和社区参与等因素。

将 SBOM 与策略即代码 (PaC) 结合起来，创建了一个强大的框架，用于管理开源软件 (OSS) 的使用、确保合规性并缓解安全风险。自动化强制执行安全策略可减轻安全团队的负担并提高整体效率。

使用 SBOM 修复依赖问题

在拥有无数依赖项的复杂代码库中，精确定位和修复受影响的制品可能是一项艰巨的任务。遵循以下最佳实践可以帮助您的组织快速应对零日漏洞利用和其他威胁：

保持 SBOM 最新

确保 SBOM 作为您的 CI/CD 流程的一部分自动生成。这确保您始终拥有关于您的软件依赖项的最新信息，涵盖您的组织支持的每个制品。

利用自动化漏洞扫描工具

使用自动化工具根据漏洞数据库扫描您的 SBOM。这些工具可以及时识别您的依赖项中的已知漏洞，从而使您能够优先安排修复工作并应对潜在的安全威胁。

建立健全的补丁管理流程

制定一个明确的流程，用于修补 SBOM 中识别出的漏洞。这包括设定修补优先级、与供应商协调以及在部署前测试补丁。通过维护一个最新且安全的软件供应链，您可以显著降低零日漏洞被利用的风险。

AI 显著增强了这些最佳实践。智能 SBOM 分析系统可以：

预测漏洞影响

AI 模型可以分析您的应用程序架构，以确定脆弱组件是否处于可利用的位置，区分理论漏洞和那些构成即时风险的漏洞。这种上下文分析有助于团队首先关注最关键的问题。

自动化依赖更新

当识别出漏洞时，AI 系统可以自动生成带有相应依赖项更新的拉取请求，测试与您的代码库的兼容性，并管理跨多个仓库的更新过程。这种自动化显著缩短了从漏洞披露到修复的时间。

识别隐藏依赖项

机器学习算法可以检测包清单中可能未明确捕获的未文档化或传递性依赖项，从而提供更完整的实际攻击面视图。

采纳 DevSecOps 原则

在本章中，我们已经看到了软件供应链是多么脆弱。持续以安全的方式交付软件，不仅仅需要对您使用的工具和第三方组件进行仔细审查。它还需要选择支持 SLSA 以及生成 SBOM 和证明的 CI/CD 工具和技术。为了确保持续的安全交付，您的团队必须维护一个安全的平台，进行彻底的漏洞测试，及时优先处理和修复问题，防止不安全代码的发布，遵守法规，并保证您的软件及其所有组件的完整性。

这不能是您团队中某个单一角色或组织内某个单一团队的工作。它需要一种协作方法，将安全集成到整个 SDLC 中。这就是 DevSecOps 方法所倡导的。与传统方法中仅在少数地方添加安全不同，DevSecOps 促进开发、安全和运营团队之间的持续协作，确保从一开始就考虑安全。

在本节中，我们将解释 DevSecOps 原则，并展示采纳这些原则如何帮助您更快地识别和修复漏洞，降低数据泄露风险，并提升软件的整体安全性。

建立协作文化，打破职能孤岛

成功实施 DevSecOps 的第一步也是最关键的一步是建立一种以安全优先为理念的协作文化。当然，这可能是最困难的一步，需要您组织领导团队的全力支持。安全必须成为组织的首要任务，并成为开发人员、运营团队、安全团队和其他人员共同承担的责任。

打破孤岛并建立共同所有权意识的一个简单方法是创建跨职能的 DevSecOps 团队。孤立的团队会限制沟通和知识共享，这可能导致重复工作和不一致的流程。相比之下，跨职能的 DevSecOps 团队能够促进协作和开放沟通。例如，在建立新的安全实践或选择新的安全相关工具时，通过纳入开发、运营和安全角色的视角，您可以更容易地获得成功所需的认同和协同。

此外，跨职能团队有助于防止导致瓶颈和生产力紧张的选择或建议。一个例子是单方面强制执行新的应用程序安全检查，而不考虑其对开发过程的影响。这不仅会增加开发人员的工作量，还会损害组织内部的信任和善意。

除了创建跨职能团队，您还应该在整个组织中识别和支持一些关键的安全倡导者，以帮助推广安全举措并在同事中提高安全意识。利用您的跨职能团队和安全倡导者，通过建立开放透明的沟通渠道来分享想法并传达您的进展，从而促进信息和思想的交流。这可以包括定期会议、团队聊天和知识共享会议。

AI 工具通过提供共享上下文并翻译安全与开发关注点之间的信息，充当安全团队和开发团队之间的协作桥梁。例如，当一个 AI 驱动的安全工具识别出漏洞时，它可以以开发人员友好的术语解释问题，同时提供安全团队所需的安全上下文。这种共同的理解减少了团队之间的摩擦，并有助于建立一种人人说相同语言的安全文化。

最后，投资于安全培训。识别技能差距，并为所有团队成员提供持续的安全培训，使他们掌握识别和缓解安全风险的知识和技能。这不仅提高了整个团队的标准，也表明了您组织对安全的优先承诺。

采纳并强制执行安全编码方法和左移

安全编码实践对于防止漏洞至关重要。OWASP Top 10 和常见缺陷枚举 (CWE) 为识别和解决常见的安全漏洞提供了有益的指导。此外，请确保您的方法能够解决以下常见威胁：

输入验证

始终验证用户输入，以防止恶意数据被注入到您的应用程序中。这有助于防止 SQL 注入、XSS 和其他注入攻击。

输出编码

正确编码输出以防止 XSS 攻击。这确保了用户生成的内容在页面上安全显示，而不允许恶意代码执行。

错误处理

实施健壮的错误处理，以防止信息泄露和潜在漏洞。避免显示可能为攻击者提供有价值信息的敏感错误消息。

会话管理

使用安全的会话管理技术来保护用户数据并防止未经授权的访问。这包括使用强大的会话标识符和实现超时。

认证和授权

实施强大的认证机制，并强制执行适当的授权控制，以限制对敏感资源的访问。

加密

使用安全的加密算法和实践来保护敏感数据。避免弱加密方法并确保正确的密钥管理。

依赖管理

保持依赖项最新并安全管理它们，以避免漏洞。使用依赖项扫描工具等工具来识别和解决第三方库中的已知漏洞。

尽管及时了解最新威胁和安全编码可以防止许多漏洞，但您组织的努力并非万无一失。静态和动态分析工具，以及早期安全测试，可以作为一道防线，捕获代码审查中可能被忽视的问题。这就是左移的意义所在。我们已经介绍了左移如何使您能够在早期阶段识别和解决漏洞。早期修复可以降低漏洞发布到生产代码中的风险。

总结

在现代软件开发中，安全不能是事后才考虑的问题。它必须是整个开发过程中不可或缺的一部分。正如我们在本章中看到的，AI 原生的安全方法改变了：

- 如何发现漏洞，使用智能分析而非仅仅静态规则
- 如何根据实际风险而非通用严重性评级来确定发现结果的优先级
- 如何实施修复，通过自动化指导和代码生成
- 如何通过持续监控和异常检测来保护供应链
- 团队如何协作，在安全与开发之间共享上下文和理解

通过将安全嵌入到从设计到部署的每个阶段，并通过培养一种由 AI 增强的共享责任文化，组织可以构建和交付能够抵御现代威胁的应用程序。

在 第 6 章 中，我们将把注意力转向通过混沌测试来揭示可能未被发现的弱点，从而提高应用程序的韧性。

第 6 章：混沌工程与服务可靠性

复杂的现代系统本质上是脆弱的。即使是看似微小的中断，或单一的薄弱环节，也可能导致问题螺旋式上升，造成灾难性后果。

考虑以下场景：一个著名的电商平台在类似黑色星期五的销售高峰期遭遇了严重的服务中断。随着流量的增加，平台的结账服务变得异常缓慢，最终导致彻底崩溃。数千名顾客无法完成购买，这不仅导致了直接的收入损失，还损害了声誉，侵蚀了信任和品牌忠诚度。事后分析显示，根本原因是结账服务与关键定价数据缓存之间的网络延迟。在高流量的压力下，当缓存响应变慢时，系统的重试机制不堪重负，导致请求级联失败，最终使数据库过载。

像这样的场景以及不断上升的故障成本，催生了服务可靠性作为一门学科的出现，以及混沌工程（有时称为故障注入测试）的实践。混沌工程的目标是帮助我们理解系统在异常（混沌）压力下如何表现。新工具、新技术和新实践的发展推动了这些实践的日益普及。

混沌工程一词可以追溯到 2010 年的 Netflix。当时该公司正在将其基础设施迁移到云端，这引入了新的复杂性，数百个微服务以不可预测的方式相互作用。为了测试其系统的韧性，Netflix 的工程师开发了 Chaos Monkey，这是一款旨在随机终止生产环境中虚拟机实例的工具。这模拟了真实世界的故障，迫使工程师构建能够优雅处理意外中断的系统。

使用“混沌”一词以及一只猴子被放出在生产环境中随机终止软件的形象，确实会让人联想到混乱。鉴于这些先入之见，将混沌工程引入组织可能会遇到阻力。不止一位老板曾疑惑：“我们这里还不够乱吗?”

在本章中，我们将通过将现代混沌工程理解为一种严谨的实验实施方法来反驳这些观念。作为一种方法论，我们利用这种**受控的**中断来测试我们系统的韧性。除了测试我们当前的状况，混沌工程还为我们提供了一种强大的方法，可以系统地提高韧性。

我们进行的实验让我们对软件在压力下的行为有了更深入的理解。这些知识使我们能够设计有针对性的改进。然后我们进行测试，以验证其在实现目标方面的有效性。

服务可靠性与服务韧性

服务可靠性和服务韧性是相关概念。前者是指服务在指定条件下，在特定时间内无故障地执行其预期功能的概率。后者是指服务承受并从硬件故障、软件错误、网络中断乃至网络攻击等中断中恢复的能力。它关乎服务从逆境中恢复的良好程度。

尽管它们是不同的，但它们相互关联。高度可靠的服务不太可能发生故障，但即使是最可靠的系统也可能遇到意想不到的问题。这就是韧性发挥作用的地方。它确保即使发生故障，服务也能快速恢复，并将对用户的中断降至最低。

我们还将探讨如何使用服务级别目标（SLO）来设定我们的韧性目标。我们将研究如何使用错误预算来允许在该目标范围内达到可接受的故障水平。我们将看到混沌工程如何与这些机制协同工作，帮助我们验证即使面对意外中断，我们的系统是否也能在其错误预算内运行并仍能达到我们的目标。

在本章中，我们还将超越静态混沌实验，了解一种更现代和动态的方法，它涉及将混沌工程整合到我们的 CI/CD 流水线中，使我们能够持续评估和改进系统韧性，作为我们日常开发工作流程的一部分。

在本章中，我们将探讨先进的混沌工程工具如何利用 AI/ML 驱动的洞察力来推荐和指导这些实验的执行，从而提高韧性测试的效率和效果，同时降低风险。我们还将看到智能体 AI、生成式 AI 和 MCP 如何通过自动化实验设计、实现动态风险检测和提供智能补救措施来解决混沌工程中关键的可扩展性和精度挑战。这些技术将混沌工程从一种被动的实践转变为一种主动的、自我优化的系统韧性策略。

混沌工程入门

虽然许多混沌工程实验都采用随机性（例如，随机选择一台服务器或服务来关闭），但混沌工程的实践就像实验室科学一样有条不紊。在本节中，我们将深入探讨混沌工程的核心原则，并探讨在进行实验时降低导致服务中断风险的最佳实践。

混沌工程原则

Netflix 定义了一系列核心原则，为探索系统在压力下的行为提供了一个有用的框架。结构化的方法确保您的混沌实验不仅仅是破坏性事件，而是结构化的调查，生成有价值的数据，您可以利用这些数据来推动系统韧性的改进。这些原则是：

定义表征正常系统行为的“稳态”

可观测性是这里的关键。您必须拥有所需的指标，以了解表示系统健康并按预期运行的正常值范围。这可能包括请求延迟、错误率、吞吐量或特定于应用程序的指标。务必考

虑可能影响指标的外部因素，例如一天中的时间、一周中的某天，或可能导致流量激增的营销活动。

将预期转化为假设

根据您对系统架构和依赖关系的理解，针对当引入特定故障时系统**应该**如何表现，提出一个假设。以能够使用所选指标进行客观测试的方式构建您的假设。例如，“如果我们将流量增加 20%，平均响应时间应保持在三秒以下，错误率不应超过 0.5%。”

通过模拟真实世界事件执行实验

使用混沌工程工具自动化故障注入。模拟服务器崩溃或变得不可用、关键第三方服务中断，或用户请求突然激增。

根据假设评估结果

将实验期间系统的行为与您建立的基线和您的假设结果进行比较。指标是否保持在可接受的范围内？系统是否按预期恢复？是否观察到任何意外的副作用？如果系统偏离预期行为，请调查根本原因。根据实验结果，完善您的假设并调整您的系统设计或操作流程以增强韧性。

从小处着手并逐步扩展

模拟故障以故意关闭系统当然会带来风险。我们以这种受控的方式故意承担风险，以验证我们定义的假设。降低风险的一个重要策略是先从小规模实验开始。

为了说明从小处着手并逐步扩展实验，我们来看一个专注于测试电商系统中结账服务的例子。该服务是处理用户购买的关键微服务。预期结果很简单：客户将商品添加到购物车，进行结账，并完成支付。客户期望获得流畅、快速和可靠的体验。

在幕后，这种直接的操作依赖于一系列复杂的流程。结账服务依赖多个 API 和外部服务才能正常运行，包括库存系统、支付网关和缓存层（如 Redis），以快速检索重要的价格、折扣和可用性数据。结账服务从缓存中获取定价数据以实现快速访问。如果缓存缓慢或失败，结账服务仍应通过故障转移到另一个缓存实例，甚至作为备份转移到数据库来提供正确的信息，尽管这可能会更慢。

> 生成式 AI 可以将混沌工程从手动假设测试转化为自适应、自优化的韧性验证系统。这种方法在结账服务等关键电商工作流中尤其有价值，因为它需要在风险缓解与实际故障模拟之间取得平衡。

开发人员通常会配置重试逻辑、超时和熔断器来处理网络问题或故障。让我们分别看看

它们：

重试逻辑

这确保了如果对缓存的请求失败或遇到网络问题，系统将在放弃之前自动重试几次。这有助于处理瞬时故障。例如，系统可能会重试多达三次，每次重试之间延迟 100 毫秒。

超时

超时设置定义了服务在决定尝试失败之前应等待响应多长时间。这可以防止服务在缓存缓慢或无响应时无限期挂起。系统可能会配置为对缓存的每个请求在 200 毫秒后超时。

熔断器

熔断器在一定数量的失败尝试后，会阻止进一步调用故障服务。如果缓存持续失败或速度过慢，熔断器会“跳闸”并将流量路由到备用系统（例如，另一个缓存或数据库）。熔断器可以在设定的一段时间后自动重置，以测试原始服务是否已恢复。例如，熔断器可能配置为在五次连续重试失败后跳闸。

我们将通过引入少量延迟来开始测试结账服务，以确保重试逻辑和超时设置正常运行，然后再逐步升级，引入更严重的问题，最终触发熔断器。如果一切顺利，我们期望系统能够故障转移到备用数据源。以下是我们的步骤。

步骤 1：进行简单的延迟实验

我们首先测试我们的重试逻辑。我们希望确保系统在网络问题（例如高延迟或临时连接丢失）出现时具有韧性。我们的稳态是一个响应迅速的服务，在可接受的时间限制内作出响应。

我们的假设是，如果网络在尝试访问缓存时出现显著延迟，系统应使用其重试逻辑和超时设置来优雅地处理问题，最终触发熔断器以防止服务进一步退化。

我们从小处着手，在结账服务和缓存之间注入少量网络延迟（例如 200 毫秒）。

我们观察重试逻辑是否启动，以及服务是否在可接受的时间限制内处理延迟而未对用户造成影响。我们持续监控系统是否在延迟后继续按预期运行，并从缓存中拉取数据。

步骤 2：测试针对更显著网络问题的韧性

一旦我们用少量延迟测试了重试逻辑，我们就可以增加实验的范围和强度，以模拟更显著的网络问题。这测试了我们的超时设置。我们增加网络延迟（例如从 500 毫秒增加到 1 秒），以查看服务在更重负载或网络拥堵下的行为。我们测试重试逻辑如何处理延长延迟。服务是否重试对缓存的调用，

并且它是否遵守超时设置？如果是，我们将通过在一定数量的重试后导致缓存 API 完全失败来增加问题的严重性。

步骤 3：验证熔断器是否故障转移到备用方案

接下来，我们设置实验条件，使缓存无法访问。在重试后，熔断器机制应该被触发。当熔断器跳闸时，结账服务会故障转移到备用数据源，例如不同数据中心中的另一个缓存实例（在本例中是我们的 Postgres 数据库）。虽然 Postgres 数据库可能比缓存慢，但目标是保持服务运行，尽管性能略有下降。

AI 代理可以通过使用强化学习动态调整故障注入参数来使这个过程变得更简单。例如：

1. 从 200 毫秒延迟开始，然后根据实时性能遥测数据自主扩展到 500 毫秒至 1 秒。
2. 最初将实验影响限制在 0.5% 的交易，仅在验证安全机制后才扩展。
3. 通过历史成功模式分析优化跳闸阈值（例如，五次失败到四次）。

您可以进一步扩展实验，通过在结账服务和 Postgres 数据库之间引入类似的网络问题来测试故障转移的韧性，以查看系统在增加的故障条件下如何继续适应。通过遵循这个过程，我们逐渐增加实验的复杂性，以验证系统的韧性机制，而不是立即跳入重大的中断。

值得注意的是，韧性机制的初始设置通常基于有根据的猜测而非精确数据。这也是通过混沌工程实验进行测试如此重要的另一个原因。

注入网络延迟只是我们可以在混沌实验中扩展的一种条件。我们将在本节后面讨论其他条件。

在类似生产环境的条件中开始

在混沌工程中，另一个重要的最佳实践是在投入生产之前，先在预生产环境中测试实验。这使我们能够在不影响真实用户的情况下安全地进行实验。我们可以快速迭代、调整参数并观察结果，不受生产约束。一旦我们确认系统在这些设置中的韧性，我们就会将实验推广到下一个环境，最终达到生产环境。每次推广都伴随着风险，因此我们必须谨慎行事。环境之间的配置漂移可能导致实验结果不一致。在环境之间移动时，坚持“从小处着手并逐步扩展”的方法至关重要，以防遇到问题。在预生产中彻底审查我们的实验可确保我们的实验设计良好且富有洞察力，而不会产生意外后果。

利用现代工具

我们广泛地探讨了在混沌实验中测试网络延迟的例子。还有许多其他类型的条件值得测试。现代工具（例如 Harness Chaos Engineering、Chaos Monkey 和 LitmusChaos）可以通过提供丰富的预定义实验目录来提供帮助。现代工具通常会提供跨类别和常见故障模式的混沌工程实验，包括：

资源耗尽

CPU 耗尽

强制高 CPU 利用率以模拟进程消耗过多处理能力。

内存耗尽

消耗所有可用内存，测试应用程序如何处理内存压力和潜在的内存不足错误。

磁盘 I/O 耗尽

生成大量磁盘读/写操作以模拟存储瓶颈。

网络带宽耗尽

饱和网络带宽，测试应用程序在网络拥堵下的性能。

网络中断

网络延迟

在服务之间或与外部依赖项之间引入网络通信延迟。

丢包

模拟网络数据包丢失，测试应用程序如何处理不可靠的连接。

网络分区

隔离网络部分以模拟服务之间的连接问题或可用区中断。

DNS 故障

模拟 DNS 解析问题，测试应用程序如何处理 DNS 中断或不正确的响应。

基础设施故障

节点故障

终止或关闭虚拟机或容器以模拟硬件故障。

Pod 故障 (Kubernetes)

杀死或驱逐 Pod，测试 Kubernetes 部署的自愈能力。

可用区中断

模拟整个可用区故障，测试您的灾难恢复计划和多区域部署。

推理层攻击

在 ML 模型服务期间模拟 GPU 内存耗尽。

应用程序级别故障

服务故障

停止或崩溃应用程序内的特定服务，测试容错和服务降级。

功能故障

在特定函数或方法中引入错误或异常，测试错误处理和恢复机制。

数据损坏

损坏数据库或存储系统中的数据，测试数据完整性和恢复过程。

状态管理

时间旅行

操纵系统时钟以模拟时间偏移，测试应用程序如何处理时间敏感操作或计划任务。

状态注入

将特定数据或状态注入到您的应用程序中，以测试其在异常条件下的行为。使用生成式 AI 创建符合模式约束的合理损坏数据条目。

使用 AI 动态生成场景

架构建模

使用 AI 分析服务依赖关系（例如，Redis 缓存 → 支付网关 → 数据库），创建镜像生产环境的故障链。

生成对抗网络

通过让 AI 模型相互对抗来发现未探索的漏洞组合，从而创建新颖的故障模式。

我们尝试的实验类型越多，我们就能越多地了解系统中的弱点以及如何增强其韧性。

更新的工具除了提供目录之外，还可以分析您的系统架构，以建议针对您特定设置的、暴露潜在弱点的实验。例如，对于使用微服务架构构建的软件，混沌工程工具可能会分析网络流量和依赖关系，以识别关键服务并建议专门针对这些服务的实验。现代工具还可能建议在服务之间的 API 调用中注入延迟或错误，以测试通信中断的韧性。

对于使用 Kubernetes 部署的应用程序，该工具可以分析您的 Kubernetes 部署并建议针对特定 Pod、部署或命名空间的实验，以测试副本扩缩、资源限制和健康检查。像 Red Hat 的 Krkn 这样的工具使用 AI 来分析 Kubernetes Pod，以优先处理网络密集型服务进行分区测试。在多区域部署的情况下，现代工具可能会分析您的多区域设置并建议模拟区域故障或网络分区的实验，以测试您的灾难恢复计划以及应用程序故障转移到另一个区域的能力。

从他人经验中学习

密切关注行业范围内的事件，特别是那些影响使用类似技术栈的公司的事件，对于主动的风险缓解至关重要。例如，OpenAI 在 2024 年 12 月 11 日发生的服务中断，就清楚地提醒我们，看似微不足道的部署也可能产生连锁反应。

在此事件中，一个新的遥测服务压垮了该公司的 Kubernetes 控制平面，触发了 DNS 故障，导致其 API、ChatGPT 和 Sora 平台停机数小时。影响是广泛而持久的：数小时内，开发人员和用户无法访问他们所依赖的服务。工程师们在几分钟内识别出根本原因，但面临一个主要障碍 —— 无法访问 Kubernetes 控制平面，回滚或修复部署变得极其困难。

让我们看几个有针对性的混沌工程实验，以了解这些级联故障可能如何被阻止。

实验 1：控制平面过载模拟

首先，我们设计一个实验来测试 Kubernetes API 服务器的韧性。在这个实验中，我们将故意用大量的读/写操作来淹没 Kubernetes API 服务器，以模拟新的遥测服务在生产环境中所做的事情。通过在具有类似生产规模的预演环境中运行此测试，我们可以发现 API 服务器开始失效的精确阈值。这种早期检测将有助于更好地限制负载、改进警报，并可能制定更安全的阶段性发布策略。

实验 2：DNS 故障测试

该实验将涉及在 DNS 解析过程中引入延迟或故障 —— 特别是针对负责服务发现的组件。运行此实验有助于确认即使 DNS 中断，基本服务也能继续运行。我们将发现我们的缓存、回退机制或替代路由策略是否足够。如果不足，我们就会知道在真正的中断发生之前投资于这些领域。

示例 3：紧急访问演练

最后一个实验（或演练）涉及模拟工程师在重负载下被锁定在 Kubernetes API 之外的情况。通过实践紧急访问方法 —— 例如拥有专用的带外通道或专业工具 —— 团队可以在标准控制平面不可访问时快速回滚或禁用有问题的部署。如果事先进行了这项演练，团队就会知道如何在几分钟内准确地移除有故障的遥测服务，从而最大限度地缩短停机时间。

服务级别目标与服务韧性

我们看到了混沌工程如何帮助我们发现弱点并构建更具韧性的系统。但我们如何定义“韧性”？我们如何衡量和跟踪我们的系统是否达到了可靠性目标？这就是 SLO（服务级别目标）和 SLI（服务级别指标）发挥作用的地方。它们共同为定义和衡量服务可靠性提供了框架，为我们提供了明确的目标和跟踪进展的方法。

SLO 是我们为服务可靠性设定的目标。SLI 是我们用来衡量是否达到这些目标的具体指标。SLO 通常表示为必须满足定义的 SLI 标准的时间百分比或请求数量。例如，99.9% 的请求延迟应低于 200

毫秒。SLI 是具体的、可衡量的指标，反映从用户角度看服务的性能。它们量化了可用性、延迟、错误率、吞吐量以及其他相关因素。

本质上，SLI 是您衡量的**内容**，而 SLO 是您为这些测量设定的**目标**。

建立可靠性目标

在建立可靠性目标时，务必使其与总体业务需求保持一致。监控和可观测性解决方案提供了许多 SLI 指标，但重要的是要优先选择那些能准确反映客户应用体验的指标。目标不是跟踪每个单独的服务，而是专注于那些对客户体验至关重要的服务。

常见的 SLI 包括“四大黄金信号”：

请求延迟 处理用户请求所需的时间

吞吐量

每秒处理的请求数量

错误率

失败请求的百分比

饱和度

系统利用率百分比

仔细考虑如何在您的系统中实现这些指标。例如，在测量延迟（响应时间）时，您可以选择跟踪所有交易，或专注于最重要的交易子集，例如登录、支付提交或将商品添加到购物车。再次强调，选择一个能有意义地代表客户体验的指标。

系统可靠性的共同所有权

在 第 1 章中，我们介绍了 DevOps 作为结合了软件开发 (Dev) 和 IT 运维 (Ops) 关注点的实践。在确保系统可靠性方面，共同所有权和协作的重要性无处不在。SLO 是这种共同责任的一个极佳示例。开发、运维和可靠性团队应该协同工作来定义 SLO。这种协作建立了对可接受系统性能的理解，并为每个人设定了共同努力的目标。SLO 进而成为指导决策的指南，以平衡快速开发（速度）与稳定可靠系统之间的需求。

有了这种共同理解，开发人员可以优先考虑维护可靠性的功能，了解他们的工作如何影响整体系统性能。同时，运维团队获得了有效支持应用程序所需的背景信息。如果 SLO 被违反，它将触发活动，鼓励工程团队在发布新功能之前稳定服务。这有助于防止不稳定循环，并确保可靠性始终是首要任务。

以协作方式设计、优先排序和执行混沌工程实验本身就能将团队凝聚在一起。所有团队都受益于从这些实验中获得的洞察力，以及在发现故障时共同解决问题的过程。

现代工具促进了这种协作式的系统可靠性方法。监控平台、事件管理系统和通信工具提供了对系统性能和潜在问题的共享可见性。实时数据和自动化警报使开发和运维团队都能快速响应事件。更重要的是，这些工具培养了一种积极主动解决问题的文化（例如，数据驱动的优先级排序、实时协作分诊等），团队可以在问题影响用户之前识别并解决潜在问题。

错误预算及其在可靠性和创新中的作用

我们已经了解了混沌工程如何帮助我们主动发现系统弱点，以及 SLO 和 SLI 如何为定义可靠性目标和衡量系统是否达到这些目标提供清晰的框架。错误预算 则作为安全网介入。

错误预算代表了服务在仍能满足其 SLO 的前提下，可以承受的最大不可靠性或停机时间。通过在追求快速创新时容忍小的故障，错误预算承认完美是无法实现的，而是帮助我们达到可接受的可靠性水平，以平衡这两个相互竞争的优先级。

让我们通过电商示例来看看它是如何运作的。假设我们设定了一个 SLO：99.9% 的网站登录时间少于 300 毫秒。在一周内，这转化为最大允许的 SLO 违反时间为 10.08 分

钟。这就是我们的错误预算。这会如何影响我们？如果错误预算耗尽，我们将停止或减缓新软件的部署，并专注于稳定系统，直到错误预算得到补充。错误预算的状态不仅影响我们的部署优先级，还影响混沌测试的优先级。

通过监控为混沌测试实验提供信息

密切关注您的 SLI 不仅仅是向您发出即时问题的警报，它还会揭示您系统中潜在的弱点。例如，如果您注意到您的系统不断触及延迟限制，耗尽您的错误预算，那么您的系统可能在高流量场景中难以跟上。这表明了一个值得您重点进行混沌实验的领域。通过模拟那些高流量、高延迟的情况，您可以看到您的系统在压力下的表现，并确保它在高峰使用期间仍能满足其 SLO。

借助现代化工具，您可以根据这些模式自动触发混沌测试，从而实现自动化，无需动手即可持续测试和改进系统的韧性。现代平台可以使用 AI 将 SLI 趋势与混沌测试建议相关联，从而显著提高测试覆盖率。

错误预算在混沌测试实验中的战略性使用

错误预算不仅仅是偶然故障的安全网；它们是管理风险的工具。以我们的电商网站为例，我们将 10.08 分钟的错误预算视为一项需要明智使用的资源。在本节中，我们将探讨如何主动利用此预算进行混沌实验。

根据您的可用错误预算确定混沌实验的优先级

有效的混沌工程需要考虑您的可用错误预算。当您的错误预算充足时，您的运行空间就很长。您可以更自由地进行激进的实验，模拟大规模故障或将关键系统组件推到极限。这可能涉及测试故障转移机制、注入网络延迟，甚至模拟核心服务的完全中断。

随着错误预算的减少，将重点转向风险较小的较小规模实验至关重要。这可能涉及隔离测试单个组件、模拟轻微网络问题或验证最近更改的韧性。以这种方式确定实验的优先级可确保您可以在不危及整体系统稳定性的情况下继续学习和改进。

现代自动化工具可以提供帮助。通过实时分析您的错误预算，这些工具可以根据您的可用“容错空间”推荐适当的实验。这使您能够在主动测试和服务可靠性之间保持平衡，确保您的混沌工程工作既富有洞察力又负责任。

利用 AI 增强的试运行和模拟保护您的错误预算

首先进行模拟是最大限度降低混沌实验期间意外后果风险的另一种策略。当错误预算 dwindling 时，这一点尤为重要。AI 增强的“试运行”混沌实验实践涉及在受控环境中，使用系统模型或副本模拟实验，以在生产中执行之前评估其潜在影响，并使用 AI

修复代理在异常检测阈值超出预定义限制时回滚实验。通过事先识别潜在问题并完善实验参数，团队可以降低导致可能耗尽错误预算并造成重大中断的可能性。

将混沌工程和 SLOs 集成到 CI/CD 流水线中

可靠性问题主要由变化驱动，包括我们应用程序的变化或基础设施的变化。Google DevOps 研究与评估 (DORA) 定义了变更失败率 (CFR) 指标，这为我们提供了挑战的另一个视角。CFR 描述了我们的变更（例如新代码部署或基础设施更新）在生产环境中引入问题并需要热修复或回滚的频率。DORA 2024 年 DevOps 状态报告显示，80% 的受访团队平均 CFR 为其发布量的 20%。事实上，25% 的团队平均 CFR 甚至高达 40%。

此外，我们必须考虑修复每个变更失败所需的时间和成本。故障部署恢复时间指标（取代了类似的平均恢复时间 [MTTR] 指标）侧重于组织从故障中恢复的速度。这让我们了解团队在这方面面临的挑战。虽然许多团队能够在不到一天的时间内修复，但 25% 的团队需要一周到一个月的时间来替换有缺陷的软件。

在前面的章节中，我们研究了防止缺陷进入生产环境的策略。我们在交付流水线的每个阶段进行测试，执行各种类型的测试。我们小心翼翼地管理我们的环境。我们通过 GitOps 等实践结合 IaC 来防止配置漂移。我们还在预生产和生产环境中进行混沌工程测试，以帮助我们发现系统中的弱点。然而，尽管我们尽了最大努力，偶尔需要快速修复的缺陷是不可避免的。这就是持续韧性的用武之地。

正如持续集成和持续交付是关于使用自动化来构建、测试和部署我们的代码一样，持续韧性是关于通过将混沌工程实验添加到我们的 CI/CD 流水线中来自动化我们的韧性实践。这样做意味着我们不仅测试功能，而且积极持续地

评估变更将如何影响我们系统的稳定性。利用 AI 代理进行 DevOps，混沌实验可以智能地集成到 CI/CD 流水线中。

在“扩展您的混沌工程实践”中，我们将探讨如何在现代工具的帮助下，将混沌工程实践集成到我们的交付流水线中，从而扩展它。我们将研究如何优先考虑要添加到流水线中的实验，以及在流水线中保护和管理混沌实验的最佳实践。

扩展您的混沌工程实践

组织开始其混沌工程之旅的方式各不相同。通常，一个或两个团队会采用开源工具并在组织的某个小范围内引入实验。一个组织可能会定期举办混沌工程“游戏日”。这些是全体参与的、有计划的活动，团队故意向系统注入一系列故障，以在受控环境中练习事件响应并识别弱点。这些活动通常不频繁，并且响应是对发现问题的被动反应。

在整个组织范围内大规模实施持续韧性的诀窍在于选择合适的工具。虽然开源和专有解决方案都提供有价值的功能，但组织应仔细评估其需求。某些企业环境可能需要特定功能，如高级安全控制、全面的审计跟踪和 RBAC —— 这些功能在开源解决方案中的可用性和成熟度可能有所不同。

一家领先的金融科技公司对此挑战深有体会，该公司每天处理超过 10 亿笔支付交易。面对高峰需求期间日益增加的交易失败，它寻求一种解决方案来提高其支持 20 多种金融产品的复杂平台的可靠性。

该公司选择现代混沌工程工具，对于克服扩展混沌工程实践的障碍至关重要。它所选择的工具（在本例中为 Harness Chaos Engineering）包含一个广泛的预构建实验库，简化了自动化和编排众多混沌实验的工作。此外，全面的分析和报告功能使该公司能够快速洞察其系统的韧性。

该公司首先将重心放在一个关键服务上，该服务每日处理九百万笔支付请求。它在复杂的底层基础设施中找出了容错目标，为韧性测试的受控推广奠定了基础。通过优先将混沌实验自动化到交付流水线和生产环境中，它解决了交易失败的根本原因，并为持续韧性建立了基础。

通过其自动化的韧性测试平台，该公司能够扩大测试范围，以发现服务恢复中的漏洞，优化应用程序设计模式，并微调配置。结果是显著的：失败交易减少了 16 倍，MTTR 减少到 10 分钟，客户满意度提高了 10 倍。如果没有提供安全性、模板、自动化和编排的现代工具，就不可能在如此短的时间内在整个组织中推广混沌工程并取得这些成果。

将混沌工程实验和 SLOs 添加到您的 CI/CD 流水线

为了巩固您的韧性策略，将 SLO 作为可靠性护栏集成到您的 CI/CD 流水线中。将 SLO 视为赛车上的刹车 —— 对于在追求最高速度时保持控制至关重要。开发团队，就像赛车手一样，努力实现快速部署，但如果没有健全的 SLO，他们可能会因未经检查的变更而使系统崩溃。通过监控关键指标，您可以自动阻止违反这些阈值或耗尽错误预算的部署。这种方法可以在不牺牲稳定性的前提下加速您的开发速度。

在将混沌工程实验添加到 CI/CD 流水线时，请记住两个衡量标准来指导您的进展：韧性得分和韧性覆盖率。韧性得分只是您的服务在 QA 和生产中针对您应用的实验表现如何。韧性覆盖率，类似于代码覆盖率，评估还需要多少实验才能全面评估系统韧性，指导创建实际数量的测试。这些指标共同提供了对韧性的整体视图，使所有团队都能为持续韧性目标做出贡献并衡量进展。

首先添加针对已知韧性条件进行测试的实验，确保您的韧性分数在每次新部署时保持稳定。通过添加测试新条件的实验来缓慢增加您的韧性覆盖率。如果增加韧性覆盖率意味

着韧性分数降低，请确定失败的混沌实验是否需要停止流水线，或者是否可以并行采取行动。

接下来，添加解决目标部署运行平台变化的实验。例如，在升级底层平台（如 Kubernetes）时，将混沌实验纳入您的 CI/CD 流水线，以主动识别潜在的弱点和兼容性问题。这有助于防止潜在问题在未来影响应用程序，并确保平台更新期间的平稳过渡。通过及早发现这些问题，您可以避免代价高昂的事件并保持持续的韧性。

将实验添加到流水线中，以根据以前的生产事件和警报在事件发生时验证部署。最后，添加实验以根据目标基础设施的配置更改验证部署。这是另一种情况，其中之前通过的韧性测试将开始失败，因为目标环境因更高或更低的配置而发生变化。

当您投入创建和微调您的实验时，像对待其他任何软件一样对待它们：版本化它们、测试它们并在存储库中管理它们的生命周期。这确保您的混沌工程实践保持有效并适应系统变化。集中式存储库促进了这些实验的协作和共享，促进了跨团队的一致性和最佳实践。

混沌工程的安全与治理

显然，混沌工程是一种强大的方法，但粗心大意的实验可能会对系统韧性和对混沌工程项目的信任造成严重损害。通过将其与您的安全和治理框架相结合，您可以定义所需的护栏，以确保实验负责任地进行。

就像技术债一样，韧性债也可能在您的生产服务中积累。每一次警报、事件、热修复或权宜之计 —— 例如简单地向问题投入更多资源 —— 都促成了这种债务。这些快速修复往往掩盖了潜在问题，并营造了一种虚假的稳定性，而不是解决根本原因。

现代混沌工程工具可以帮助您建立和执行政策来对抗这种情况。例如，我们可以设定一项政策，规定对于每一个与组件行为异常、网络问题、API 故障或意外负载相关的生产事件，都必须有一个相应的混沌实验。这个实验，集成到您的 CI/CD 流水线中，需要在特定的时间范围内进行验证，比如说，在事件发生后的 60 天内。这样的政策不仅能强制执行解决韧性债的纪律，还能鼓励开发人员和质量保证团队优先修复生产代码，而不是推送可能进一步加剧问题的新功能。

除了使用策略来管理韧性债务之外，您还可以使用安全治理策略来防止未经授权的实验，限制对关键系统的访问，并按环境、时间窗口、人员甚至故障类型限制测试。通过自动化监督并将这些策略集成到 CI/CD 流水线中，您可以提高韧性覆盖率，同时降低风险。

AI 原生混沌工程在服务可靠性中的未来

混沌工程的未来预示着在服务可靠性实践中将变得更加复杂和集成。Harness Chaos Engineering 和 Chaos Monkey 等工具不仅将自动化实验，还将利用 AI/ML 预测其影响，分析系统在压力下的行为，并推荐最佳缓解策略。这种智能自动化将最大限度地降低风险，使团队能够以更高的信心和效率进行更复杂的实验。可观测性和追踪方面的进步将提供对系统动态更深入的洞察，从而更精确地识别漏洞并更快地从中断中恢复。

随着系统日益复杂，分布式架构和微服务变得无处不在，混沌工程将在确保其韧性方面发挥关键作用。即使是基于大型语言模型的多智能体系统也可以通过混沌工程进行增强。通过将混沌测试与 AI 驱动的分析和自动化修复（例如，ChaosEater）相结合，我们将能够更快、更精确地解决潜在故障，最大限度地减少停机时间并保持高水平的服务可靠性。

总结

在本章中，我们探讨了混沌工程作为构建和验证系统韧性的系统方法。我们学习了如何负责任地设计和执行实验，利用 SLO 和错误预算来平衡创新与稳定性。通过将混沌工程集成到 CI/CD 流水线并利用现代工具，组织可以主动识别弱点，从故障中学习，并持续提高韧性。最终，混沌工程使我们能够创建更健壮的系统，以满足当今复杂数字世界的需求。有了这些原则，下一步就是将它们无缝地应用于您的部署流程。让我们探讨如何在生产发布期间确保稳定性。

第 7 章：部署到生产环境

2012 年 8 月 1 日发生的 Knight Capital 事件是一个鲜明的例子，它展示了软件部署出错可能导致灾难性后果。那天，Knight Capital（当时美国最大的交易公司之一）对其自动化交易系统进行了大规模更新。由于自动化程度有限、部署中的人为错误以及糟糕的特性标志管理等多种因素的共同作用，过时的代码被意外重新激活，导致系统迅速下达错误订单到股票市场。

在仅仅 45 分钟内，有缺陷的算法执行了超过 400 万笔交易，给该公司造成了高达 4.6 亿美元的巨额损失。这起事件不仅几乎让 Knight Capital 破产并最终被收购，还导致了重大的市场混乱。它突显了在高风险软件环境中，健全的部署实践、彻底的测试、治理和故障安全机制的关键重要性。

部署到生产环境可能是一项高风险活动。虽然并非所有应用程序都是市场制造型交易平台，但值得更新的应用程序都有依赖它们的用户，并且任何更改都会引入风险。尽管我们可能更希望通过减少部署频率来避免这种风险，但我们面临着更频繁更改的业务需求。此外，随着我们延迟并将在计划发布中积累越来越多的更改，某些类型的风险会增加，这使得持续交付方法更具价值。

在之前的章节中，我们在软件交付过程中探讨了减轻最终部署到生产环境风险的策略。在第二章中，我们讨论了代码审查的重要性。在第三章中，我们研究了如何使用早期扫描和单元测试来快速检测问题。第四章描述了其他类型的测试以强化您的软件，并回顾了部署到测试环境的最佳实践。通过使用一致的工具、管道步骤和部署策略，

并通过对环境差异进行参数化，我们利用部署到测试环境来验证我们部署到生产环境的部署。在第五章中，我们深入探讨了安全性，回顾了帮助保护生产部署的实践。

如今，人工智能正在改变组织如何进行生产部署以防止此类灾难。机器学习系统现在可以分析部署模式，在发布期间检测异常，并以比传统监控更高的精度验证应用程序健康状况。与依赖预定义阈值的基于规则的验证不同，AI 系统可以学习每个应用程序特有的正常行为模式，并检测可能预示新问题的细微偏差。这些能力使团队能够更快地部署，同时又奇迹般地降低了风险 —— 这与传统的速度与安全权衡背道而驰。

在本章中，除了涵盖人工智能的变革性作用外，我们还将探讨生产部署治理的最佳实

践、安全部署到生产环境的策略，并讨论可观测性以验证生产部署的质量。我们将探讨现代 AI 驱动的部署工具如何通过智能验证而不是仅仅被动监控来帮助缓解风险，以及 AI 驱动的系统如何同时评估多个信号以确定部署健康状况，从而捕获可能被人工操作员忽略的问题。

生产部署治理

Knight Capital 事件是一个很好的提醒：部署有缺陷的软件可能导致灾难性的财务损失。您的组织的信任和信誉也面临风险。对组织而言，部署后修复缺陷的成本可能飙升，远远超过在开发阶段解决这些问题的费用。

要充满信心地进行部署，我们需要了解哪些代码发生了变化以及谁进行了这些更改。我们需要验证我们实施的代码审查流程是否已执行，并了解谁进行了这些审查。对于引入的任何依赖项，我们希望了解它们并知道它们符合我们的政策。我们希望知道它们是否已针对任何已知缺陷进行了审查。我们需要确保所有代码更改都已执行我们要求的构建、扫描和测试流程。当然，我们希望确保扫描和测试的结果确实符合我们的通过标准。最后，我们要求有证据表明我们的开发流程本身符合与我们组织相关的框架和要求。

严格的代码审查、彻底而强大的测试实践以及自动化、可重复的部署程序对于避免部署失败至关重要。然而，如果没有适当的治理和控制来确保我们遵守了流程，我们所有的努力都可能变得无效。

人工智能正在开始改变部署治理，尽管大多数应用仍在兴起。当前的 AI 系统主要侧重于分析部署模式以识别风险因素和策略违规，而不是做出自主决策。这些系统可以同时处理比人类更多的部署变量，有助于识别代码更改、部署配置和历史事件之间的细微关联。组织开始利用这些洞察力来完善其治理框架，尽管人工监督对于最终的批准决策仍然至关重要。

> 部署治理就是对软件部署过程进行系统性的监督和控制，以确保我们定义的规则和策略得到执行。从根本上说，治理是为了降低变更的风险。治理包括组织用来确保软件部署以一致、受控、安全和合规的方式进行的策略、流程和工具。治理的挑战在于平衡对敏捷性和创新性的需求与对稳定性和风险管理的需求。

在接下来的几个部分中，我们将讨论部署治理的传统方法和现代方法。我们将研究如何自动化我们治理策略的强制执行，以使我们的交付过程更高效。我们将审查支持我们治理流程的工具和策略，最后，我们将展望部署治理的未来。

传统部署治理方法

传统部署治理方法是为 DevOps 前的世界构建的。在这个世界中，生产环境的更改不频繁、风险高，并且由传统的运维团队执行。决策是集中的，并涉及僵化的流程。

信息技术基础设施库（ITIL）是一种广泛采用的框架，它代表了一种传统方法。ITIL 最初在 1980 年代出现，以应对对标准化 IT 管理实践的需求，从最佳实践的集合演变为一个综合框架。它包括几个与部署治理直接相关的流程和实践。

其中之一是变更管理流程，它定义了一种结构化的方法来管理服务和基础设施的所有变更，包括部署。它规定了对拟议变更的正式文档，包括其目的、范围、影响和风险评估。变更咨询委员会（CAB）或类似机构评估变更。变更请求根据其优点和潜在风险被正式授权或拒绝。如果变更获得批准并执行，

将进行实施后评估，以确保变更达到其目标并识别任何经验教训。

发布管理流程同样正式，它规定了将发布规划、调度和控制到生产环境。它与变更管理流程密切相关，旨在确保部署以受控和透明的方式执行。

CAB 是 ITIL 等方法的一个典型特征。CAB 是一个委员会，由负责正式评估和批准或拒绝拟议软件变更的个人组成。该委员会可能包括一名负责协调变更请求审查和跟踪变更实施的变更经理，以及技术专家、业务利益相关者、安全专家、合规官等。其目的是通过从多个角度彻底评估请求来降低风险。

此外，如果出现任何问题，CAB 还可以确保问责制。虽然一个高效运作的 CAB 将提供预期的监督，但最糟糕的 CAB 由不专心的审查人员组成，他们不进行评估就草草批准审查。或者 CAB 可能名义上有效但效率低下。邮件驱动的审批流程因被忽略的邮件、审批人外出且没有委托以及审查会议被重新安排而减慢。

研究表明，这些传统的 CAB 流程不仅效率低下，实际上对它们旨在确保的稳定性是适得其反的。Forsgren、Humble 和 Kim 在他们的里程碑式研究《加速》一书中，在谈到高性能组织时解释道：“外部审批与交付周期、部署频率和恢复时间呈负相关，与变更失败率无关。”换句话说，像 CAB 这样的外部机构的审批明显减慢了交付速度，而没有提高稳定性。

发生这种情况是因为 CAB 将责任与知识分离；对变更了解最深的人不是做出批准决定的人。虽然这些委员会营造出尽职调查的表象，但它们通常充当合规剧场，当事情出错时，它们提供了一个可以指责的对象，而不是真正防止失败。它们提供的控制错觉甚至可能降低实施变更人员的警惕性，因为“CAB 批准了”成为推卸责任的挡箭牌。

CAB 会议的开销，加上低效和延迟，在应用程序不经常发布时尚可容忍。随着发布频率的增加，CAB 的问题变得越来越明显。

现代部署治理方法

在前面的章节中，我们探讨了如何通过自动化每个阶段的步骤来简化开发过程，从而实现更快、更频繁的软件发布。现代部署治理方法同样侧重于自动化手动步骤，这些手动步骤是不必要的软件发布障碍。

现代方法不再依赖委员会和手动审批来做出部署决策，而是倾向于自动化决策和部署。由于生产部署的风险很高，因此必须非常谨慎地进行。在本节中，我们将探讨如何做到这一点。

除了自动化之外，现代治理方法还利用当前的策略和工具来管理合规性。我们将研究如何使用审计日志来简化合规性，以及 Open Policy Agent (OPA) 等工具来强制执行安全和法规标准。

自动化决策

借助现代 CI/CD 工具，我们可以让我们的管道自主做出部署决策。如果我们能够确保我们的管道能够充分强制执行治理策略以维持我们的标准，我们就可以通过消除或最大限度地减少手动审批来加速软件交付。

考虑以下步骤，以在您的交付过程中自动化部署决策：

1. 明确您的“通过”标准

为推广构建明确标准对于自动化部署过程至关重要，但这可能具有挑战性。我们合作的一家银行将其控制措施记录在一个三英寸厚的活页夹中，其中包含数百页的法规和政策。通常，决策者可能同时依赖客观数据和主观判断。模糊性使得将人类决策转化为一套僵硬的自动化规则变得具有挑战性。例如，决策者可能会推广一个有少量次要测试失败的构建，如果他们认为这些问题风险低且不太可能影响用户。然而，将这种直觉转化为准确评估风险和用户影响的自动化规则可能很复杂。人工智能在将人类决策中更模糊的元素引入完全或大部分自动化流程方面发挥着越来越大的作用。如果以这种方式使用，它应该被要求解释其建议和见解。

2. 使用“质量门”实现复杂标准，尽可能自动化控制

门是 CI/CD 管道中的检查点，用于评估特定标准，以确定构建是否应进入下一阶段。门可以考虑测试结果、基于静态分析结果的代码质量指标、代码覆盖率、编码标准合规性、安全扫描结果和性能指标。其他工具允许您引入一个管道步骤，如果决策为“否”，则该步骤将失败；或者您可以根据您的特定标准设置条件执行。通常，最简单的方法是配置每组测试，使其在不符合您的标准时失败。这样，如果任何步骤失败，整个管道将停止，从而阻止不合格构建的推广。

3. 自动化细微决策时，考虑历史结果

例如，安全计划通常从对新的高优先级问题实行零容忍政策开始，但会容忍现有问题，直到团队解决它们。这需要考虑历史数据，而不仅仅是最新的结果。

4. 最后，将自动化标准化

使用标准化选择或痛苦的手动合规性作为使用标准化工具的激励。我们合作的银行的团队可以选择通过证明发布符合活页夹中详述的所有控制来部署到生产环境，或者使用其标准化的自动化流程和工具。这成为了一个简单的选择。

建立强大的审计追踪以自动化合规性

部署治理和合规性密切相关。有效的治理实践对于实现和维持与各种监管标准和框架的合规性至关重要。

我们在第五章中回顾了几个框架，特别是与安全相关的框架。PCI DSS 是一个广泛适用的例子。它用于确保所有接受、处理、存储或传输信用卡信息的公司都维护一个安全环境。无论您处理的交易规模或数量如何，如果您的组织处理持卡人数据，则您都必须遵守其要求。主要卡品牌（Visa、Mastercard 等）如果无法证明合规性，可能会处以罚款或限制您处理卡支付的能力。

虽然 PCI DSS 主要侧重于保护持卡人数据，但有几个要求直接涉及软件开发和部署过程。这是为了确保处理此数据环境的整体安全性。例如，PCI DSS 要求您通过采取措施（例如在发布到生产环境之前对自定义代码进行审查和解决常见的编码漏洞）来开发和维护安全的系统和应用程序。PCI DSS 还包括

测试要求，规定在任何重大的基础设施或应用程序升级或修改后进行内部和外部渗透测试。

强大而全面的审计追踪对于证明合规性所需的实践至关重要。虽然您的组织可能不受 PCI DSS 要求的约束，但许多其他可能相关的框架将对其开发和部署流程提出类似的要求。

您的源代码控制和 CI/CD 系统在这里发挥着至关重要的作用，它们捕获了交付管道中每个操作的详细信息，从代码提交和构建到测试结果、部署和环境配置，以及相关的用户、时间戳和任何相关的元数据。这包括日志记录用户操作、系统事件、工件跟踪、配置更改和外部集成。通过将这些信息以结构化和可访问的格式存储，CI/CD 工具提供了通用审计追踪，可适应任意数量的安全和法规框架。

支持强大审计追踪的工具使您的组织能够证明合规性，而无需为每个框架维护单独的日志。它还可以让您主动解决潜在的安全或合规问题。

使用策略即代码管理强制执行

策略即代码（PaC）在自动化生产部署的同时保持强大的治理方面发挥着重要作用。PaC 是将安全、合规和操作策略定义和管理为代码的实践，从而实现自动化强制执行。策略以声明式语言定义，可以像任何其他关键代码一样进行管理：在源代码控制中进行版本控制，从而实现跟踪、协作、所需的代码审查和回滚能力。

OPA 是一种流行的开源策略引擎，用于实现 PaC。通过 OPA，每次部署都会根据您定义的策略进行自动评估，确保一致的强制执行，而不会减慢您的交付过程。想象一下，您的部署策略要求所有容器镜像在进入生产环境之前必须经过关键漏洞扫描。使用 OPA，您可以表达此 PaC，并将其集成到您的管道中。现在，每次触发部署时，OPA 会自动扫描镜像，如果镜像干净则允许部署继续，如果发现漏洞则停止部署。这消除了手动安全检查，并确保在没有人为干预的情况下，始终如一地遵守您的安全标准。

OPA 的多功能性超出了安全检查。您可以将各种部署策略编纂成代码，例如强制执行金丝雀部署、要求批准特定更改或验证资源配置。通过自动化这些检查，您可以确信每次部署都符合您组织的标准和法规要求。这不仅加速了您的交付过程，还降低了人为错误和不合规的风险。

保护部署流程

严格控制您的治理机制有助于保护您的部署。开发人员赋能对于现代部署也至关重要。在实践中，您需要在两者之间取得平衡。虽然您希望使开发人员能够适应其部署管道，但您也需要防范潜在风险。恶意行为者可能会篡改或绕过您设置的治理机制，或者他们可能会因人为错误而受到影响。或者，对部署的过度严格控制可能会为高效部署制造另一个障碍。

OPA 也能在这里提供帮助。通过 OPA，您可以对策略更新过程本身应用严格控制，确保对您的治理框架的任何更改都经过仔细审查和符合规定。通过将策略规则集中到 OPA 并将其应用于管道，您可以创建关注点分离。这使得单个开发人员更难规避策略，因为他们需要修改中央 OPA 策略，而这些策略可能受到更严格的访问控制、同行评审和审计追踪。

随着我们越来越依赖 AI 为我们生成管道，OPA 策略既为 AI 提供了我们想要的指导性输入，也提供了保护，确保 AI 的输出符合我们的标准。

保护部署过程的另一个重要控制是实施健壮的 RBAC。如第二章所述，RBAC 允许您精细控制谁有权修改管道和 CI/CD 平台内的敏感配置设置。这确保了只有授权人员才能更改您的部署过程，从而最大限度地降低恶意活动的风险。

通过结合这些方法，您可以集中执行策略，确保您的部署防篡改并得到有效监控。

部署治理的未来趋势

与软件开发的几乎所有领域一样，人工智能和机器学习将推动部署治理的重要未来趋势。例如，预测分析是数据分析的一个分支，它应用机器学习技术分析历史数据以预测未来结果。应用于软件部署时，预测分析可用于识别模式和风险因素以标记潜在问题。供应商正在创建仪表板，例如 Digital.ai 的“变更风险预测”，该仪表板基于团队失败率和测试中发现的缺陷等趋势。如今，大多数这些解决方案都是在广泛数据集中发现的相对简单的相关性。我们有理由期望未来能从模型中获得更多洞察力，特别是那些能够轻松访问更广泛数据集的 DevOps 平台。

您的团队可以在问题影响用户之前主动解决它们。AI 和 ML 可用于实时自动执行治理策略，分析代码更改、配置和部署，以确保符合安全和操作标准。这些进步将使组织能够以更快的速度、更大的信心和更强的弹性交付软件。

协调传统与现代方法

在传统治理方法中，ITIL 将标准变更定义为预批准、低风险且具有明确程序的变更，允许以最少形式授权更快地实施。通过使用现代 DevOps 实践，依赖质量门和现代策略强制执行，我们可以显著降低即使是复杂软件部署的风险。这种控制和可靠性水平允许将这些部署视为标准变更。从本质上讲，现代 DevOps 实践中固有的风险缓解与 ITIL 的标准化、可预测变更管理目标相一致，从而在不损害稳定性或合规性的情况下实现更快、更频繁的部署。

在“生产部署策略”中，我们将探讨如何使用渐进式部署来进一步降低生产部署的风险。

生产部署策略

在第四章中，我们介绍了如何自动化我们的部署流程，现在我们已经了解了如何通过部署治理实践来降低风险。接下来，我们将重点转向实际的软件生产部署业务。在本节中，我们将研究如何通过渐进式部署技术进一步降低风险。即使拥有最强大的治理和最谨慎的渐进式部署，我们的部署仍然可能失败。我们必须准备好回滚策略，因此我们将研究快速恢复的方法。最后，我们将研究工具选择。选择现代工具可以帮助您轻松地进行治理、渐进式部署和回滚。

传统的大爆炸式部署

在研究现代方法之前，我们可以回顾一下传统方法 —— 以及对于有状态应用程序的某些元素仍然可能需要的方法。传统上，我们会将应用程序下线，升级应用程序每个组件的每个实例，然后重新启动应用程序。经过快速验证后，我们会将其重新暴露给用户，并观察一段时间以确保其健康，然后才认为部署成功。如果出现问题，我们会将应用程序重新下线并尽力回滚应用程序 —— 这通常是一项艰巨的挑战。

这种传统方法需要应用程序停机，引入了重大风险，并需要工程师投入大量精力。改进的机会比比皆是。

使用渐进式交付策略

软件部署就像走钢丝 —— 一步走错，后果可能很严重。但渐进式部署策略为您提供了一个安全网。通过逐步推出更改并密切监控其影响，这些策略最大限度地降低了风险，并在出现问题时允许快速纠正。在本节中，我们将介绍多种流行的部署策略，包括滚动更新、蓝绿部署、金丝雀部署以及特性标志的使用。

部署滚动更新

滚动部署是一种非常常见的交付策略，您可以通过逐步将旧版本实例替换为新版本实例来增量更新应用程序或服务。这是以受控方式完成的，确保在更新过程中始终有一定数量的实例可用以处理用户流量。

滚动部署具有明显的优势。它们最大限度地减少了停机时间，因为应用程序在整个更新过程中保持可访问。重要的是，滚动部署降低了风险。通过增量更新实例，可以及早检测并解决新版本的潜在问题，从而限制其影响。这种部署类型可以根据特定的应用程序需求进行定制，允许对更新过程的速度和规模进行无限调整。

然而，实施和管理滚动部署可能比其他部署策略更复杂，特别是对于大规模或分布式系统。还存在潜在的不一致性。在更新过程中，同时运行的两个不同版本的应用程序可能导致数据或用户体验的差异。此外，回滚正在进行的部署可能很复杂，并且需要额外的步骤来保持数据完整性。

实现选项众多。Kubernetes 通过其 Deployment 对象提供对滚动更新的内置支持。新的 Pod（带有更新版本）会逐渐创建，旧的 Pod 在新的 Pod 准备就绪后终止。容器编排平台（例如 Docker Swarm、Nomad）提供类似的滚动更新机制，允许增量替换容器或服务。负载均衡器可用于通过在旧实例可用时逐渐将流量从旧实例转移到新实例来实施滚动更新。在某些情况下，滚动部署可以使用管理更新过程和监控应用程序健康状况的自定义脚本或自动化工具来实施。

虽然滚动部署需要付出努力才能实施，但它们为在应用程序更新期间最大限度地减少停机时间和风险提供了一个有价值的选择。

使用蓝绿部署

蓝绿部署是一种发布策略，涉及维护两个相同的环境，通常称为“蓝色”和“绿色”。在任何给定时间，这些环境中的一个（通常是蓝色）是实时运行的，服务于生产流量。

当您的应用程序新版本准备就绪时，它会部署到非活动环境（绿色）。在绿色环境中进行测试和验证后，流量从蓝色环境切换到绿色环境，使新版本上线。滚动部署是随着时间推移进行更新，不同流量会体验到不同版本的服务，而蓝绿部署通常具有硬切换功能。开关一拨，流量（或至少是新流量）就会立即从旧版本转移到新版本。图 7-1 描绘了部署更新前后的蓝色和绿色环境。

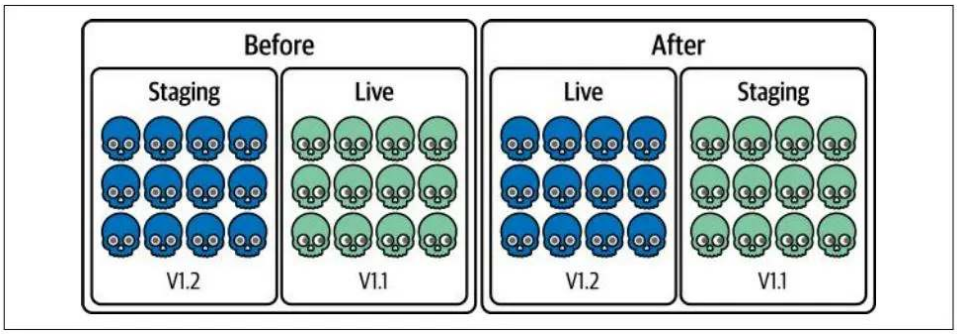

图 7-1. 蓝绿部署涉及运行两个相同的环境，以实现无缝更新和回滚选项（在纸质书中，蓝色显示为深灰色，绿色显示为浅灰色）

以前的实时环境（现在是蓝色）可用于下一次部署，也可以作为需要回滚时的备份，或者被解除。

蓝绿策略具有显著优势：

减少停机时间

流量在环境之间切换，最大限度地减少对用户的任何中断并减少停机时间。

轻松回滚

如果新部署出现问题，流量可以快速切换回以前的版本。

改进测试

新版本可以在上线前在类似生产的环境中进行测试。

主要缺点在于基础设施成本增加，因为维护两个独立、相同的环境可能很昂贵。此外，蓝绿部署可能不适用于具有复杂状态管理或数据库模式更改的应用程序，因为在环境之

间同步数据可能具有挑战性。

更高级的蓝绿模型可以通过集成 IaCM 实践来克服大部分基础设施成本挑战。在稳定生产状态下，只存在一个实例。在部署开始时，部署会触发 IaCM 工具来预置一个新实例，从而使蓝色和绿色都存在。在流程结束时，多余的实例被解除。因此，多余的基础设施只需要在蓝绿部署期间存在。

使用金丝雀发布

金丝雀发布提供了另一种类似于滚动更新的渐进式策略。应用程序的新版本会发布给一小部分用户或服务器。这个“金丝雀”组充当测试平台，允许您在真实世界的生产环境中监控新版本的性能和稳定性，然后将其提供给所有用户。

在典型的金丝雀部署中，只有一小部分流量（例如 5% 到 10%）可能被引导到新部署的版本。新版本的性能、稳定性、错误率会被密切监控并与现有版本进行比较。响应时间、CPU 使用率和错误日志等指标会被分析，以识别任何潜在问题。如果新版本在金丝雀环境中表现良好，则引导到它的流量百分比会逐渐增加，允许更多用户访问它。这个过程持续到新版本完全取代旧版本。如果在金丝雀阶段检测到任何问题或性能下降，部署可以迅速回滚，最大限度地减少对用户的影响。

金丝雀部署可以通过简单的指标阈值实现，但它们越来越多地利用 AI 或 ML 功能来确定新版本是否表现令人满意。传统上，金丝雀部署侧重于性能基准，但我们可以预期，未来它们将越来越多地利用业务指标，即使新版本的应用程序没有崩溃，如果它损害了业务，也会停止发布。

虽然金丝雀部署和滚动更新都旨在实现渐进和受控的软件发布，但它们的侧重点不同。滚动更新解决了最小化服务基础设施的停机时间和服务中断的问题。金丝雀部署则侧重于以指标为指导的决策，决定是逐步增加新版本的流量还是回滚到旧版本。

使用特性标志

特性标志提供了一种以渐进方式部署**特性**的策略。将特性标志想象成代码中隐藏的开关，允许您为特定用户或组打开或关闭特性，而无需部署新代码。这使您可以精细控制谁看到什么，从而实现 A/B 测试和有针对性的发布。特性标志与其他渐进式部署策略类似，它们允许您在真实世界环境中监控性能并收集反馈，并利用这些信息来降低风险。然而，特性标志在不同级别操作；它们控制单个版本内的功能。其他渐进式策略测试一个全新的版本。

特性标志带来的好处不仅限于部署风险缓解，我们将在第八章中再次讨论它们。

回滚

我们已经探讨了几种渐进式部署策略，但您的选择是无数的。变体和混合方法，融合了滚动更新、蓝绿部署、金丝雀发布和特性标志的元素，都是可能性。这些策略的共同点是受控发布，允许您在需要时停止部署并回滚到以前的版本。对于蓝绿部署这样的策略，这是一个简单的提议：您的以前版本随时待命。对于滚动更新或金丝雀部署，回滚过程是关于从带有缺陷软件的节点中移除流量，然后系统地用以前的软件版本替换这些节点。

回滚不仅涉及重新部署先前稳定的软件版本，还包括其相关的配置、依赖项和数据。回滚到先前状态的复杂性可能与部署本身一样复杂，甚至更复杂。某些部署方法将促进可靠的回滚。例如，如果部署是幂等的，意味着它可以重复并实现相同、无损的结果，那么重新部署先前版本将等同于回滚。

测试回滚对于确保您可以无所畏惧地回滚至关重要。仅仅有一个回滚机制是不够的；您需要定期验证其就绪状态。这包括模拟各种故障场景，然后执行回滚程序，以确保其快速可靠地恢复到以前的稳定版本。彻底的回滚测试验证了应用程序、其数据和其依赖项是否正确恢复。根据应用程序及其数据存储机制，回滚可能需要数据恢复或迁移以确保数据一致性。定期测试程序以确保它们在所有场景中按预期工作。

对您的回滚程序充满信心后，您可以根据部署健康状况配置自动触发回滚。验证部署健康状况是下一节将讨论的主题。系统不再依赖手动干预，而是当预定义的阈值被突破时，自动恢复到以前的稳定版本。这不仅减轻了运维团队的负担，还将人为错误排除在外，从而最大限度地减少停机时间。

特定架构的特殊考虑

部署和回滚的复杂性因软件架构而异。单体应用由于其紧密耦合的代码库，通常需要完整的部署和回滚，这会影响整个系统。另一方面，微服务提供了更细粒度的部署和回滚，针对单个服务。然而，这种相互连接意味着必须仔细管理依赖项，以确保服务之间的一致性。分布式单体应用兼具单体架构和微服务的特点，结合了微服务的部署复杂性和单体应用的相互依赖问题。

数据库增加了另一层复杂性。当更新涉及对持久数据结构进行破坏性更改时，需要“扩展和收缩”等策略。该策略涉及在现有数据库字段或表旁边添加新的字段或表，部署更新后的应用程序以利用新结构，并最终逐步淘汰旧字段。该方法实施起来很复杂，但通常需要这样做才能在支持渐进式部署策略和干净回滚时确保数据完整性。

选择合适的工具

有了渐进式部署策略和强大的回滚能力，您就可以充满信心地部署到生产环境。但要真正发挥这些策略的力量，您需要合适的工具。现代部署工具是关键，它们开箱即用地为

渐进式部署策略提供无缝支持。

在选择用于编排软件部署的工具时，除了基本功能之外，还需要考虑给定工具与您的特定需求有多么契合。如果您计划在采用新的持续交付工具的同时过渡到自动化部署决策，那么提前了解所有影响您的推广决策的因素将有助于您选择具有所需治理和门功能的正确工具。此外，请确保部署工具与您的目标环境无缝集成，无论是云、本地服务器还是混合设置。同样重要的是，该工具能否处理您的特定应用程序类型和架构，包括任何复杂的数据库部署或协调的多服务发布。

除了基础设施和架构兼容性之外，部署工具还应包含对您首选渐进式部署策略的开箱即用支持，确保您可以轻松实施金丝雀发布、滚动更新或其他

技术。健壮的回滚机制应作为首要关注点，因为它们允许您在出现意外问题时快速恢复到以前的稳定版本。此外，还要考虑该工具是否与您现有的特性标志管理系统集成，或者是否提供自己的特性标志功能，从而让您对特性发布进行精细控制。

验证生产部署

即使是最勤奋的治理也无法消除对健全实践的需求，以系统地验证您的生产部署。在本节中，我们将探讨可观测性的作用。我们将讨论如何现代化您的验证流程，并研究特定于生产部署验证的测试策略。

部署中的可观测性

验证您的部署从可观测性开始。可观测性简单来说是指通过检查系统的外部输出来理解其内部状态的能力。可观测性让您从知道出了问题到理解为什么出了问题，从而实现更快的故障排除和更有效的根本原因分析。可观测性数据包含三个关键支柱：

指标

这些提供系统性能的定量测量，例如响应时间、错误率和资源利用率。通过跟踪这些指标的趋势和异常，团队可以快速识别潜在问题并评估新部署的影响。

日志

日志提供应用程序及其基础设施内部发生的事件和错误的详细记录。分析日志数据有助于查明问题的根本原因并了解导致问题的事件序列。

追踪

追踪提供请求如何流经系统的可视化表示，突出瓶颈、延迟问题以及不同服务之间的依

赖关系。这有助于识别性能问题并优化应用程序架构。

现代化“战情室”

传统的部署验证通常类似于高风险的战情室场景，工程师们监控仪表盘和日志，随时准备在出现问题的最初迹象时进行手动干预。这个过程高度依赖人工，依赖人类对可观测性数据的解读。它也是被动的，团队通常只在问题影响用户后才匆忙解决。

这种方法不仅压力大、效率低，而且容易遗漏问题并导致响应时间慢。此外，它常常导致验证程序不一致，并且对问题根本原因的可见性有限。

部署验证的现代化涉及自动化这些手动任务和人类决策。自动化系统取代了工程师监控仪表盘和日志的工作，持续分析遥测数据并在检测到异常时触发警报。从被动监控转向主动监控减少了对人工干预的需求并加快了响应时间。

实现这种自动化的诀窍是将您的部署工具与您的可观测性平台集成。集成可以根据所使用的工具采取不同的形式。在一种方法中，您的 CI/CD 工具在部署进行时通知可观测性平台，提供一个可以用于触发回滚的“钩子”。然后可观测性平台分析遥测数据并决定是否启动回滚，调用 CD 工具提供的钩子。

或者，Harness 等 CD 工具可以配置为在部署过程中监视一个或多个可观测性工具是否存在问题迹象。如果检测到问题，CD 工具可以自动触发其自己的回滚机制，停止部署并恢复到以前的稳定版本。部署工具和可观测性工具之间的紧密集成实现了无缝和自动化的验证过程，最大限度地减少了停机时间并确保了更快的反馈循环。

无论哪种情况，业界都不再容忍停机，并寻求在应用程序失败之前检测出问题正在酝酿的迹象。因此，AI/ML 被用于分析多个数据源，以识别预示失败可能性的异常。AI 异常检测已成为现代部署验证的核心组件。与依赖预定义阈值的传统监控不同，这些系统围绕数百个指标构建正常应用程序行为的统计模型，并且可以检测复杂的、多维的异常，而这些异常是无法通过静态规则定义的。这种能力在生产部署后的关键几分钟内特别有价值，因为细微的性能问题可能在升级为影响用户的事件之前被忽视。

部署验证系统将这些 AI 功能集成到自动化验证门中，在整个部署过程中提供持续评估，而不是一次性检查。当检测到异常时，这些系统可以自动暂停渐进式发布，甚至自动触发回滚过程。

测试生产部署

我们在第四章中详细讨论了测试。现在我们回过头来，看看特别适合生产验证的测试策略。验证生产部署需要分层测试方法。

合成测试可以与分阶段或渐进式部署搭配使用。通过在生产环境中模拟典型的用户交互和事务，合成测试运行各种场景以快速捕获问题。这使得团队能够尽早解决问题，无论是通过回滚部署还是通过实施必要的修复。

除了初始部署阶段，生产环境中的持续测试对于确保长期稳定性和性能至关重要。合成测试继续发挥着重要作用，提供对关键用户旅程的持续监控，并识别任何回归或性能下降。第六章中涵盖的混沌工程更进一步，通过故意向系统中注入故障来测试其弹性和恢复能力。

持续测试的另一个重要方面是渐进式特性披露。这涉及逐步向一部分用户发布新特性，允许团队在全面发布之前收集反馈并监控性能。A/B 测试等技术可以比较不同版本的特性，帮助识别最有效的实现。这种受控的特性发布方法最大限度地降低了风险，并允许基于真实用户行为做出数据驱动的决策。通过结合合成测试、混沌工程和渐进式特性披露，组织可以建立一个全面的测试策略，确保其生产部署的持续验证和改进。

总结

随着人工智能继续改变生产部署，部署策略和特性管理之间的联系变得越来越重要。AI 驱动的部署验证系统不仅监控整体应用程序健康状况；它们现在还可以跟踪部署中单个特性的影响，提供粒度见解，为回滚决策和未来的特性发布提供信息。这些系统创建了一个持续的反馈循环，其中部署数据输入特性标志决策，而特性行为为部署策略提供信息。现代平台分析跨部署的特性性能模式，以推荐哪些特性应通过特性标志逐步发布，以及哪些特性可以安全地传统部署。这种智能有助于团队平衡开发速度与运营稳定性，为管理生产环境中的部署和特性创建更复杂的方法。在第八章中，我们将深入探讨特性管理。

第 8 章：功能管理与实验

在第七章中，我们探讨了将软件部署到生产环境的挑战和最佳实践。我们重点关注了降低风险和确保可靠性的策略，研究了渐进式部署策略，并结合了强大的回滚机制。这种方法有助于我们及早发现新软件版本中的问题，从而保障生产系统的完整性。回想一下，我们曾提到功能标志是一种重要的渐进式部署策略；功能标志是一种在软件的单个版本中以渐进方式部署单个功能的机制。在本章中，我们将继续讨论功能标志作为管理功能部署的工具。

我们还将深入探讨功能标志的另一个作用 —— 它们如何推动实验。虽然功能标志对于降低部署风险和实现渐进式交付非常有用，但其影响远不止于此。当与人工智能结合时，它们使您能够运行实验，从而了解用户、优化功能设计理念、验证假设，并做出数据驱动的决策，从而改善产品可用性、用户参与度以及整体业务成果。

功能管理和实验管理密切相关 —— 功能标志是控制特定功能是否启用的基本开关。功能管理系统提供了基础设施，用于控制功能何时、以何种方式、向哪些用户在何种条件下发布，而实验则利用这种控制来衡量每个变体的影响，并帮助团队做出数据驱动的决策。然而，尽管功能管理与功能标志功能强大，但它们也伴随着自身的风险和挑战，我们也将对此进行探讨。

回想一下我们在第七章中对骑士资本事件的讨论。一次错误的软件部署导致在 45 分钟内损失了 4.6 亿美元。该事件发生在骑士资本部署了其交易软件的新版本，该版本重新激活了一段休眠的遗留代码。一个配置错误的功能标志是罪魁祸首。这个旨在控制一段遗留代码激活或非激活的标志，在某些服务器上被错误地启用，而在其他服务器上则没有。这种不一致性触发了过时的逻辑，导致在不到一小时内发生了超过四百万笔错误的交易。

尽管功能标志为帮助团队大规模交付提供了巨大的潜力，但正如骑士事件所示，它们的滥用或管理不当可能会引入重大风险。有效的功能管理需要周密的规划、彻底的测试和强大的治理，以防止此类灾难发生。

人工智能正在通过使实验和实施变得更加易于访问和富有洞察力，从而改变功能管理系统。现代人工智能平台能够以通俗易懂的语言解释统计结果，根据用户模式建议最佳的

发布策略，自动检测异常，甚至生成针对特定实验量身定制的实现代码。人工智能增强减少了开发人员的认知负担，同时使产品团队能够进行更复杂的实验。

在本章中，我们将探讨缺乏人工智能能力的传统自研功能管理解决方案的局限性，并探讨现代人工智能增强系统如何不仅降低风险，还充分发挥功能管理和实验作为交付高质量软件的战略工具的潜力。

现代软件开发中功能管理的好处

想象一下，我们的组织正在实施一个处理小型企业基本在线交易的支付平台。该平台支持支付处理、开票、基本分析以及与电子商务平台的集成。我们持续快速地发布新功能，以进行迭代改进并解决用户反馈。

在本节中，我们将探讨如何利用现代功能管理使我们的组织摆脱传统发布流程的束缚，从而加快支付平台发布周期。我们将讨论如何使用功能管理来支持跨团队协作和渐进式交付。最后，我们将探讨功能标志如何帮助我们管理技术债务。

利用功能标志加速开发周期

最简单地讲，功能标志允许我们部署”关闭”的新功能，从而将部署与功能发布解耦。然后，我们可以像开关一样拨动功能标志，以便稍后激活功能，而无需部署新代码。这种方法有助于我们实现主干开发。正如我们在第二章中探讨的，持续集成涉及将代码更改定期合并到共享存储库中，并通过自动化测试确保每次集成的质量。主干开发在此基础上鼓励开发人员直接向主分支（通常称为”主干”）进行小而频繁的提交。

人工智能系统通过根据简单的提示生成将代码块包装在功能标志中所需的代码，从而加速向主干开发的过渡。这减少了可能不熟悉功能标志的开发人员的认知负担。用功能标志包装所有新更改可确保主分支保持稳定，即使有频繁的小型提交。

主干开发的替代方案涉及长期存在的功能分支。使用这些替代方案，集成会随着时间的推移变得越来越困难，因为当多个团队长时间在隔离的分支中工作时，他们通常只在集成期间发现代价高昂的冲突。延迟集成也削弱了持续集成实践的好处，因为问题可能在代码编写后很长时间才被检测到。主干开发被广泛认为是行业最佳实践，因为它有助于团队最大限度地减少合并冲突并保持对主分支的稳定更新流。更改合并得越频繁，主分支在任何时候都可部署的可能性就越高。这意味着更快、更可靠的发布。

在没有功能标志的情况下，难以部署主干开发所特有的小型更改，因为所有更改都会立即在生产环境中激活。这需要团队之间紧密同步发布，限制了安全合并和部署不完整功能的能力。

功能标志为这一挑战提供了优雅的解决方案。通过允许开发人员将新功能或实验性更改封装在功能标志中，他们即使功能尚未完全开发或未经过生产测试，也可以将工作提交到主分支。该标志有效地充当了看门人，确保不完整的功能在生产环境中保持关闭，直到您准备就绪。这种方法消除了对长期存在的功能分支的需求。这有助于团队保持高部署速度并验证代码库的其他方面，而不会受到功能完成时间线的阻碍。

解耦团队以减少协调开销

回到我们的支付平台，假设我们要引入一个新的“订阅支付”功能。前端团队负责更新用户界面以支持定期支付选项，后端团队必须实现用于管理订阅计划的 API，分析团队则负责跟踪订阅的用户行为。

没有功能标志，发布将成为一个紧密耦合、高风险的事件。前端团队无法在后端 API 上线之前部署更新的 UI，导致他们的工作在暂存环境中未完成。后端团队无法完全测试 API，因为前端未集成，从而延迟了工作流的验证。分析团队无法实现跟踪，因为订阅系统在生产环境中无法运行。

这种依赖关系迫使所有团队调整他们的日程并协调一次大规模、整体且有风险的发布。一个团队的任何延迟都会波及其他团队，造成瓶颈。如果发现一个关键错误，回滚功能意味着撤销所有团队的工作，通常需要重新部署整个应用程序。

使用功能标志，每个团队都可以独立工作并逐步发布其更改。前端团队可以提前部署订阅管理 UI，并将其隐藏在功能标志之后。这样做可以让他们在等待后端准备就绪时在生产环境中验证基本功能。后端团队可以实现并部署订阅 API 到生产环境，同样通过功能标志进行门控。这些 API 可以使用测试数据或有限用户进行测试，即使前端尚未上线。分析团队可以添加跟踪机制并将其部署在另一个标志之后。他们可以模拟用户流程，以确保正确收集指标，而无需向实际用户公开功能。

一旦所有组件准备就绪，功能标志将为内部测试而打开。一旦在内部验证，该功能便可以推广到生产环境。我们不仅降低了发布“订阅支付”功能的风险，还减少了跨多个团队协调的开销。借助人工智能驱动的功能标志，这种协调变得更加简化。人工智能助手可以自动建议跨团队的标志依赖关系，在可能出现冲突时发出警报，甚至根据历史部署模式推荐多团队功能发布的最优顺序。

通过分阶段推出支持渐进式交付

当我们的“订阅支付”功能准备就绪时，我们可以使用功能标志逐步推出更新。借助现代功能标志系统，我们可以应用目标标准，例如用户属性或百分比，以针对部分用户启用功能。这使我们能够通过缓慢启用功能、监控其性能并进行调整，然后在扩大受众范

围之前在生产环境中进行验证。

在分阶段推出期间，我们正在关注关键指标，例如 API 错误率、响应时间、支付成功率和客户反馈。如果我们观察到异常，例如支付失败尝试激增、延迟增加或用户体验中断的报告，这些可能表明新功能引入了我们需要在继续之前调查的问题。

人工智能通过预测性目标定位和自适应控制显著增强了渐进式推出。机器学习模型分析用户行为模式并预测最佳推出策略 —— 确定哪些用户应该首先看到某个功能以获得最大影响。在推出期间，人工智能系统可以实时监控指标并根据性能数据自动调整速度，加速成功的部署，同时快速识别和遏制有问题的部署。

如果确实发现问题，我们可以通过简单地关闭标志轻松回滚。新功能将对所有用户禁用，而无需重新部署代码库。系统会立即恢复到稳定、先前测试过的应用程序版本，从而最大限度地减少中断并为团队提供时间来调查和解决问题。

使用功能标志管理技术债务

功能标志不仅仅用于启动新功能 —— 它们在现代化遗留代码时可以作为安全网。通过这种方式，它们更像是调光器开关，而不仅仅是开/关按钮。在重构时，您可以从旧代码逐步过渡到新实现，同时保持在出现问题时回滚的能力。

现代人工智能原生功能管理系统擅长管理这种复杂性，它们通过跟踪功能标志的使用模式并识别不再需要的标志。机器学习算法可以分析代码依赖关系、标志状态和使用指标，从而自动识别过时的标志并建议删除它们，防止技术债务的积累，同时保持系统完整性。

这在实践中通常是这样工作的：首先，您在新改进的代码实现与现有代码并行编写。然后，您创建一个功能标志，让您可以控制运行哪个版本 —— 旧实现还是新实现。这允许您在生产环境中用少量流量测试新代码，而大多数用户继续使用经过验证的遗留代码。随着您对新实现信心的增加，您可以逐渐增加其处理的流量百分比。

这种方法对于大规模重构项目尤其有价值。您可以使用功能标志以受控波次迁移用户，而不是进行有风险的“大爆炸”替换。如果您发现任何问题，可以立即为受影响的用户恢复到旧系统，而不会扰乱整个用户群。

此模式的真正力量体现在多个组件同时进行现代化改造的复杂系统中。功能标志让您可以对现代化工作进行精细控制，让您可以在协调多个重构计划的同时保持系统稳定性。

通过实验优化结果

我们已经看到功能标志如何通过主干开发帮助我们更快地发布，并消除对协调多团队发布的需要。我们还看到如何通过渐进式功能发布降低发布风险。但是，如果我们发布的功能没有提供价值，这还有意义吗？这就是人工智能驱动的实验发挥作用的地方。

只有少数功能管理系统包含对实验的强大支持，使团队能够直接在现有应用程序基础设施中运行受控的、可测量的测试。通过结合细粒度目标定位、用户群体的随机百分比分配和自动化统计分析，这些系统允许工程和产品团队在用于功能发布的相同基础设施内无缝地进行实验。这消除了对单独的实验平台的需求，这意味着您只需要管理、监控和为单个模式编写集成代码。

精心设计的实验改变了产品开发。您不再依赖主观意见和无休止的争论，而是可以使用真实世界用户行为来指导您的决策。这样做取代了会议室猜测和无休止的争论，转而使用关于实际能够吸引用户的更改和新功能的具体数据。功能标志允许我们将用户分成不同的组（例如 A 组和 B 组），并让每个组体验不同的功能变体。例如，在一个在线支付平台中，一个组可能会看到一个“快速支付”按钮，而另一个组则会体验一个更新的“快捷结账”工作流。

通过使用功能标志，我们可以并行地实时部署这些变体，从而实现并排实验，提供版本之间的实时、直接比较。这种方法比顺序测试变体具有明显的优势，因为在顺序测试中，常规波动、季节性差异、营销活动的存在或其他因素可能会扭曲结果。通过并排实验，我们确保两个版本都受到相同的条件，以获得最可靠和最准确的见解。这些比较有助于我们自信地识别哪个版本为用户提供了最大价值，而没有顺序测试带来的噪音和不确定性。

现代人工智能极大地加速了我们的实验能力。例如，一种称为“多臂老虎机”的机器学习方法利用强化学习实时动态地将更多流量分配给表现更好的变体。例如，如果早期数据显示“快捷结账”优于“快速支付”，人工智能会自动在实验仍在运行时将更多用户路由到获胜变体，从而在测试进行期间最大化业务价值（并最小化损失）。这种自适应优化确保用户更快地体验到最佳版本，而无需等待手动分析和决策。

构建结构良好的实验

有效的实验始于一个定义明确的假设，该假设与您的业务目标保持一致，并概述了具体的、可测量的目标。例如，您可能会假设“快捷结账”工作流将通过简化支付流程来提高转化率。重要的是要记住，实验的目的不仅仅是确认您的假设 —— 而是为了学习。与您的假设相悖的早期结果并非失败；这些实验提供了宝贵的见解，可以节省数月对一个

不太可能实现其目标的项目的投资。

一个好的实验能确保结果有意义且可操作。它将功能性能与外部因素分离，以便观察到的结果可以完全归因于正在测试的更改。随着实验在团队和产品之间扩展，实验应遵循以下标准进行指导，这一点很重要。

强大、清晰的指标

每个实验都应以一个明确定义的假设和一个关键指标开始，该指标捕获成功的样子。例如，如果您正在测试“快捷结账”工作流，您的主要指标可能是从结账启动到完成的转化率。清晰的指标使实验集中，实现可衡量的进展，并防止事后合理化。同样重要的是要确定护栏指标 —— 如错误率或客户流失率等辅助指标 —— 它们可以标记意外的副作用，并防止您盲目优化单个数字而牺牲整体健康。

有针对性且随机化的受众

接下来，重点关注有效地定位您的实验。能够为特定用户群（例如按设备类型、位置或客户层级）定制实验至关重要。但即使在这些定制的受众中，也必须保持随机化以避免结果偏差。例如，不要在不同区域测试相同的功能而不进行随机抽样。这确保了任何观察到的差异都是由于功能本身造成的，而不是外部因素。请记住，在管理多个实验时，受众重叠成为一个问题：确保用户不会接触到可能扭曲结果的冲突实验。

具有统计显著性的样本量和实验持续时间

在运行实验之前，请使用功效分析计算最小样本量。这有助于定义实验需要运行多长时间才能得出可靠的结论。“功效不足”的实验会浪费宝贵的时间，因为样本量不足可能导致不确定或误导性的结果。仔细考虑实验持续时间：如果您的结果看起来不确定，预先定义停止标准有助于您决定何时结束实验，无论是由于达到

统计显著性还是达到性能阈值。这种方法可以防止浪费精力，并确保团队不会陷入结果模糊的无休止实验中。

实验分离

在实践中，大多数组织不会一次只运行一个实验。相反，产品不同部分可能同时有数十个实时实验，这带来了新的复杂性。团队必须考虑测试交互 —— 尤其是当实验重叠或针对相同用户时。管理良好的实验平台通过自动跟踪曝光、在需要时强制互斥以及揭示可能的冲突来提供帮助。

AI 驱动的解释

集成到功能管理平台中的现代人工智能助手简化了团队解释实验结果的方式。人工智能无需统计专业知识来分析复杂数据，而是可以以通俗易懂的语言解释结果，阐明实验结

果的影响。例如，当一个实验显示转化率增加 5% 但平均订单价值略有下降时，您可以要求人工智能解释这些权衡及其业务影响。人工智能可以同时分析多个指标，识别相关性，并建议可能不那么明显的潜在关系。

为了做出自信的产品决策，我们需要相信实验提供的见解。通过设计包含这些关键要素的实验，我们可以确保其可靠性和准确性。

将实验与渐进式交付集成

正如我们渐进式推出像“订阅支付”这样的新功能以降低风险一样，我们也可以使用功能标志以受控和安全的方式实施实验。例如，假设我们开发了一个修订的“订阅支付”工作流。这次迭代旨在简化用户体验。我们的假设是，此版本将导致订阅注册量增加。

为了验证这个假设，我们使用功能标志将用户分成两组：一组体验原始工作流，另一组则与更新版本交互。通过随机分配用户，我们确保公平比较，并收集有关关键指标（例如注册率和完成时间）的可靠数据。这种方法允许我们在真实世界条件下评估新工作流的性能，而不会将整个用户群暴露于潜在问题。

如果指标显示更新版本在驱动注册方面优于原始版本，我们可以开始逐步将其推广到更大比例的用户。这迭代过程不仅最大限度地降低了风险，而且还确保我们根据真实数据做出决策，这些数据要么证实要么驳斥了我们的假设。

建立护栏

我们讨论了在定义实验假设时识别关键指标的重要性。同样重要的是识别一个或多个护栏指标。

例如，我们曾与一家将贷款申请人与贷款提供商匹配的公司合作。贷款申请人使用注册流程提供有关他们感兴趣的贷款类型以及许多其他详细信息。该服务能够将用户与最能满足其需求的贷款提供商进行匹配。

产品团队确信重新设计的注册流程将提高贷款匹配的质量。他们首先小心翼翼地将这个实验推广到一小部分用户。在推广过程中，团队发现新流程导致了更高的流失率，即用户在完成实验之前就离开了。在这种情况下，流失率作为护栏指标帮助团队近乎实时地发现问题并采取行动。

通过分析数据，产品团队可以决定采取何种行动。他们可以缩小用户群规模或暂停更广泛的推出，这将使他们能够继续利用实验了解对目标指标的影响，同时限制副作用。或者，如果产品团队得出结论，副作用对整体业务价值过于有害，他们可以完全取消实验。

护栏指标，如上例中的流失率，与目标指标的目的不同，但对于确保实验的成功同样重要。目标指标衡量实验的主要目标 —— 例如提高转化率、增加收入或增强用户参与度 —— 而护栏指标则充当安全检查，用于监控意外的负面后果。用作护栏的示例指标包括跳出率、页面加载时间、客户流失率、错误率和次要产品线的转化率。

护栏指标帮助您保持对实验影响的整体视图，使您能够在主要目标进展与产品的整体健康和可靠性之间取得平衡。通过同时跟踪两者，您可以做出明智的决策，决定何时继续、暂停或调整实验。

最有效的护栏是自动化的，并无缝集成到实验流程中。现代功能管理系统可以实时监控护栏并自动执行阈值。人工智能驱动的异常检测通过使用机器学习识别可能逃脱人类注意力的微妙模式，从而显著增强护栏监控。这些系统建立基线指标行为，并在实验变体导致意外偏差时自动发出警报，甚至在它们达到预定义阈值之前。此外，人工智能可以同时关联多个指标，以检测简单的阈值监控会遗漏的复杂交互。

自动化通过减少因人为错误而无意中错过性能下降的机会来保护您的系统。使护栏成为实验过程的核心部分有助于支付平台保持敏捷性，同时保持可靠性。这种方法确保新功能可以提供价值，而不会危及用户信任。

没有成熟功能管理工具的生活

虽然功能管理系统提供了巨大的价值，但它们的有效性和成本效益取决于其实现和治理。依赖脆弱的、自研的解决方案或多个分散的实现可能会适得其反，特别是当这些基本解决方案缺乏定义现代功能管理平台的复杂人工智能功能时。对于许多团队来说，当他们的需求基本时，自然会开始构建自己的功能管理系统。然而，随着他们的需求变得越来越复杂，为添加越来越多的功能所付出的努力变得越来越难以证明是合理的。这种努力的成本和累积的技术债务最终会超过最初的任何节省。在本节中，我们将更详细地探讨自研系统的缺点。

低质量工具阻碍有效的功能标志管理

如果没有适当的工具和治理，功能标志可能会成为负资产。挑战在于基本功能标志实现与真正有效的大规模功能标志管理之间的差距。随着功能标志在团队和项目中的广泛采用，最初只关注简单切换而不进行测量的第一代解决方案，以及最初为解决简单用例而构建的内部解决方案，很快就暴露出它们的局限性。如果没有复杂管理能力，团队难以维护对其不断增长的功能标志生态系统的可见性和控制。

考虑一个典型的场景：一个开发团队使用基本的切换系统在他们的应用程序中实现了数

十个功能标志。虽然这最初有效，但他们很快发现无法轻易跟踪标志所有权、监控标志状态或管理标志生命周期。系统缺乏人工智能驱动的自动化清理通知、使用跟踪或依赖映射等关键功能。结果，开发人员不知道哪些标志仍然需要，哪些应该被淘汰。代码库充满了“僵尸标志” —— 没有人敢删除的永久切换，因为他们无法确定该标志是否真的过时。此外，保留僵尸功能标志会限制废弃或过时的代码，这些代码可能未经过测试或维护，从而产生漏洞并增加技术债务。

专业级功能管理工具应提供全面的治理功能，包括清晰的所有权跟踪、自动化清理流程、依赖可视化和强大的访问控制。这些功能可确保功能标志随着系统复杂性的增长而保持资产，而不是负债。

对实验的最小支持限制了您的学习

自研系统通常缺乏支持高质量实验所需的先进功能。例如，虽然基本的功能标志系统可能允许您全局开启或关闭功能，但它通常不支持按地理位置、设备类型或客户层级等属性进行细粒度目标定位。同样，这些系统很少提供真正的随机百分比发布，即用户群体被随机且一致地划分，以确保实验的公平性和可靠性。诸如使用机器学习优化流量路由等高级功能，即使是最复杂的自研工具也无法实现。没有这些功能，实验可能会产生偏差或不可信的结果。

此外，现代实验系统包含用于自动化统计分析和指标跟踪的内置工具，使团队能够直接在平台内评估关键绩效指标（KPI）和护栏指标。例如，如果您正在支付平台上测试更新的结账流程，现代系统可以自动计算转化率，识别统计显著性，并标记异常情况，如错误率增加 —— 所有这些都无需人工干预。相比之下，基本系统严重依赖外部工具和手动数据聚合，这增加了操作复杂性和错误风险。这种缺乏集成和复杂性使得团队难以有效地进行实验，最终限制了数据驱动决策的潜力。

缺乏集成会减慢您的速度

基本功能管理系统的另一个重大限制是它们缺乏与更广泛的软件开发生态系统的集成，这通常会导致更多的交接、手动步骤以及复杂、难以维护的脚本。现代功能管理系统通过与关键工具和平台的紧密集成来解决这些挑战，将功能管理无缝地嵌入到您的工作流中。

脆弱的实现分散您的团队注意力

最值得注意的是，自研系统通常无法很好地扩展。它们可能很脆弱，容易出现性能瓶颈。自研解决方案通常缺乏正式的服务水平协议（SLA）或专门的支持结构，导致正常

运行时间和可靠性降低。当这些系统遇到故障时，您的团队必须投入宝贵的资源来排除故障和解决中断。

相反，一个健壮的功能管理系统可以帮助您高效可靠地交付业务价值。虽然构建内部解决方案可能是开始功能管理的一种简单方法，但这些自研系统通常难以满足高性能开发团队不断变化的需求。

当组织内不同的团队开发自己的独立功能管理实现时，挑战会成倍增加。这种碎片化在几个关键领域造成了不必要的复杂性：管理功能部署、维护安全标准以及在整个组织中建立一致的治理实践。在接下来的部分中，我们将探讨通过现代、专门构建的工具集中功能管理如何简化操作、增强安全性并改善团队之间的协作。

扩展功能管理和实验

扩展功能管理和实验需要人工智能驱动的模式来简化流程并确保一致性。在本节中，我们将探讨统一功能管理单一实现、利用智能集成减少手动工作和改善协作的优势。我们将探讨现代平台如何帮助自动化治理，同时利用您现有的身份管理基础设施。我们将了解现代系统如何确保可伸缩性。最后，我们将了解人工智能驱动的功能如何改变实验。

统一使用单一功能管理实现

我们曾与许多寻求现代化软件交付流程的大型公司合作。在这一旅程中，他们常常惊讶地发现，他们正在管理十几个或更多独立构建的自研功能管理系统，有时还混杂着部分实现的商业或开源解决方案。随着这些组织的成长，他们的软件和交付流程变得更加复杂，对集中式功能管理系统的需求变得清晰。碎片化的实现放大了配置错误、安全漏洞和不合规的风险。在可审计性至关重要的行业中，这些差距使得合规成为一场艰苦的战斗。

此外，在团队之间维护多个定制系统会带来越来越多的技术债务。更新、修补和同步这些系统所需的努力和资源会分散交付业务价值的注意力。学习如何使用多个系统对于开发人员和产品经理在团队之间切换时也是一种负担。

集中式功能管理实现为公司提供了所有环境中功能标志的单一、一致视图，并允许跨团队安全、一致的功能发布能力。通过统一平台，公司能够轻松跟踪活动标志的状态、监控其使用情况并了解功能之间的依赖关系。缺乏统一视图可能导致部署期间的错误。依赖关系可能变得混乱，错误激活或停用标志的真实风险增加，尤其是随着系统复杂性的增长。

人工智能通过自动化上下文决策制定显著增强了这些集成。当与 CI/CD 管道集成时，人

工智能可以自动检测哪些功能标志受到特定代码更改的影响，有助于确保在部署前进行适当的测试。

通过智能集成减少手动步骤

现代系统通过将功能管理直接嵌入到更广泛的软件生态系统中来简化工作流。与集成开发环境（IDE）的集成允许开发人员直接在他们的编码环境中创建和管理功能标志，减少上下文切换并简化开发过程。CI/CD 管道集成使团队能够将功能标志整合到自动化构建和部署过程中，从而使功能标志成为其中的自然组成部分。

同样，与任务管理、通知和审批平台（如 Jira、Slack、Microsoft Teams 和 ServiceNow）的连接确保功能标志更改可以实时跟踪、审批和沟通，使利益相关者保持知情并减少误解。

对于基本的功能管理系统，功能标志本身的配置是导致开发人员繁重工作的主要原因。传统上，实施功能标志和实验需要开发人员仔细配置 SDK、编写目标规则并确保正确跟踪指标。现代人工智能系统解决了这个问题，因为它们可以根据您的实验配置自动生成这些实现代码。以下是它特别强大的原因：通过集成到您的 IDE 中的编码助手，人工智能了解您的实验上下文，并可以生成针对您的用例量身定制的代码。

例如，如果您已配置一个实验来测试某些地理区域中高级用户的新结账流程，人工智能可以为您选择的编程语言生成所有必要的代码。这包括：

- 使用正确的语法和 API 密钥将功能标志注入到您的代码中
- 实施所需的任何实验跟踪遥测
- 处理边缘情况和错误条件

人工智能会根据您的具体需求调整其代码生成，并可以解释其实现选择。如果您需要修改生成的代码或使用不同的编程语言实现它，您可以简单地要求人工智能根据您的新要求重新生成它。这极大地缩短了从实验设计到实现的时间，同时确保了高质量的一致代码。

通过消除手动脚本编写和在这些工具之间实现自动化，内置集成创建了工作流，不仅减少了繁重的工作，还提高了协作、效率和敏捷性。

通过自动化审计跟踪和强制执行简化治理

现代功能管理系统通过自动化批准和策略执行等关键流程来简化治理，这有助于您的团队保持控制，同时减少运营开销。例如，您可以设置自动化工作流，要求对生产环境中的任何功能标志更改进行强制批准，并要求这些标志必须首先在测试环境中激活。这样

做可以保护敏感环境免受意外或有风险的修改，同时在开发或暂存环境中提供更大的灵活性，在这些环境中，实验和迭代更为常见。这种强制执行的差异平衡了工作效率和生产稳定性。

这些系统中的策略还可以帮助标准化跨团队的实践。例如，可以自动强制执行一致的标志命名约定，即使标志数量增加，团队也能一目了然地理解标志的目的。此外，现代系统可以引导标志通过定义的升级生命周期，确保用于测试或实验的临时标志在不再需要时得到适当的淘汰。对于高风险更改，例如生产环境中的部署，这些系统可以强制使用黄金管道 —— 预定义、经过验证的流程，以确保严格的测试和可靠的发布。通过自动化这些治理任务，现代系统消除了歧义，使团队与组织标准保持一致，并显著降低了可能危及可靠性或安全性的配置错误的可能性。

利用您现有的身份管理基础设施

现代功能系统支持单点登录（SSO），允许您的团队使用内部身份提供商的现有凭据，并且跨域身份管理系统（SCIM）简化了用户配置和角色分配，确保在系统之间存在正确的帐户和权限。结合 RBAC，您可以强制执行一致的治理，确保只有授权用户才能调整功能标志或修改设置。这确保每个用户在需要时拥有其角色所需的权限，不多也不少，从而减少了安全漏洞和合规性违规的可能性。SSO 和 SCIM 共同增强了治理，简化了入职和离职流程，并确保了跨团队的安全、一致的访问控制。

选择为扩展而构建的平台

现代系统以可扩展性和可靠性为核心构建。它们利用内容分发网络（CDN）和低延迟、高可用性架构的其他特性，在峰值负载下保持性能，随着用户群和系统复杂性的增加，负载会随时间显著增长。这些系统还采用推送架构，即时在各个环境中传播配置更新，从而实现近乎即时的回滚或实时目标更改等功能。通过将其他关键任务应用程序的最佳实践（如冗余和容错）结合起来，现代系统确保功能管理在重流量或意外需求高峰期间也能保持健壮和响应迅速。

总结

本章探讨了功能管理和实验如何作为现代软件交付的基础元素，使团队能够更频繁地部署代码，同时通过渐进式发布和强大的回滚能力保持稳定性。我们了解到，功能标志不仅有助于管理部署风险，还通过实验推动业务价值，使团队能够根据真实用户行为而非猜测做出数据驱动的决策。此外，我们还看到现代功能管理平台如何通过提供全面的治理、可扩展性和人工智能驱动的功能来克服自研解决方案的局限性，使实验更易于访问

和富有洞察力。

当我们转向第九章中的云成本管理时，我们将探讨大规模运营的另一个关键方面：了解和优化我们在云环境中架构和运营决策的财务影响，在云环境中，实现快速功能交付和实验的灵活性必须与资源效率和成本效益保持平衡。

第 9 章：云成本管理的 AI 与自动化

对于现代企业而言，云环境及有效的云成本管理策略的重要性无论如何强调都不过分。根据 Gartner 的预测，2025 年全球企业在公共云服务上的支出将达到惊人的 7234 亿美元，这比 2024 年的 5957 亿美元有了显著增长。云环境和服务已成为现代软件交付的核心，云支出也已成为 IT 预算中的一个重要组成部分。

管理这些不断增长的成本已成为一个复杂的问题。最近的行业估算表明，30% 的云支出被浪费了。原因有很多：企业通常配置的云资源超出实际所需，导致出现未被使用或利用不足的实例以及被遗忘的服务，而这些服务仍在产生费用。

在本章中，我们将深入探讨云成本管理这个棘手的问题。我们将回顾实践是如何从云计算的早期发展至今，并催生了 FinOps（财务运营）这一学科。由于碳足迹和云成本管理相互关联，我们还将探讨云成本管理如何推动环境可持续发展计划。

我们还将探讨 AI 驱动的解决方案如何应对不可预测的支出、耗时的优化任务以及多云治理的复杂性等挑战。我们将研究具体的 AI 驱动策略来优化云资源，例如利用经济高效的定价模型和管理容器化环境。此外，我们还将探讨 AI 如何赋能云治理和合规性，以确保您的组织在云投资方面既高效又安全。

云成本管理的发展演变

我们将首先探讨云成本管理如何随着时间演变，并研究 FinOps 如何提供一个框架来解决云成本管理的挑战。最后，我们将探讨自动化在 FinOps 中的重要性。

早期云采纳及其初始挑战

在云时代之前，企业通常拥有并维护自己的本地基础设施，这需要对硬件和软件进行大量前期投资（资本支出）。IT 预算通常是固定的，并与这些资产的折旧周期挂钩，从而形成了一个僵化的框架。尽管成本可预测，但这种模式难以适应不断变化的业务需求。

云计算凭借其按需、按用量付费的 IT 资源，颠覆了这一模式。支出从资本支出转变为运营费用，成本根据实际使用情况随时间摊销。这提供了更大的灵活性，以应对市场变

化、扩展运营并避免过度配置，但按使用付费也带来了新的挑战。云服务的早期采用者常常因这种模式而面临意想不到的成本。对云资源使用方式的可见性有限，带来了新的挑战，控制成本成了一场艰苦的斗争。

FinOps 的兴起

早期的云先行者开发了自己的成本优化实践，以应对管理云支出的挑战。作为早期云成本管理平台之一的 Cloudability，围绕这些挑战培育了一个社区，最终在 2019 年成立了 FinOps 基金会，从而正式确立了“FinOps”这一术语。

FinOps 实践强调协作和云成本的共同所有权，以及个人和团队对云使用及其相关成本的责任。FinOps 的关键在于依赖数据和报告来理解云支出模式并识别优化机会。与 DevOps 一样，持续改进的精神对 FinOps 至关重要。FinOps 实践旨在持续进行，并随着时间的推移迭代优化云使用和成本。

FinOps 的核心原则

FinOps 基金会定义了六项核心原则来帮助组织管理云成本。具体如下：

团队需要协作

FinOps 鼓励技术、财务和业务团队之间紧密协作，以促进对云成本及其与业务目标关系的共同理解。开发人员、工程师和产品经理被授权对其云使用做出明智的决策，并为优化工作做出贡献。

决策由云的业务价值驱动

云支出决策应由其为业务带来的价值驱动，而不仅仅是成本考量。FinOps 鼓励理解在云中交付产品或服务的成本，从而制定更好的定价策略和投资决策。

人人对自己的云使用负责

个人和团队对其消耗的云资源及相关成本负责。这使得团队能够对其云使用做出负责任的选择，并为成本优化工作做出贡献。

FinOps 数据应可访问且及时

云支出数据应易于获取并保持最新，以实现及时的分析和决策。组织应利用数据分析和报告来理解云支出模式并识别优化机会。

集中式团队推动 FinOps

一个专门的 FinOps 团队（通常由 FinOps 实践者领导）推动 FinOps 实践的实施和持续改进。该团队负责费率优化，同时维持共享责任模型，使工程团队能够专注于优化其环

境使用。

团队利用云的可变成本模型

FinOps 鼓励利用云的可变成本模型，根据需要弹性伸缩资源，使支出与业务需求保持一致。该原则强调使用云原生工具和策略来优化成本，例如调整资源大小、利用折扣以及自动化成本节约措施。

FinOps 的阶段

FinOps 的三个阶段 —— 知情（Inform）、优化（Optimize）和运营（Operate） —— 为组织逐步改进其云财务管理提供了框架。以下是每个阶段的详细介绍。

知情（Inform）。这个阶段侧重于了解您的云支出和使用模式。在此阶段，我们提出以下问题：

我们在云上花费了多少?

这涉及从包括云提供商计费系统在内的各种来源收集数据，以创建您的云成本的集中视图。

钱花到哪里去了?

这需要将成本分配给特定的部门、项目或业务单元。为资源打标签和使用成本分配工具对于这一步至关重要。

我们如何使用云资源?

这涉及分析使用模式，以了解不同团队和服务如何影响整体云支出。

作为这项调查的一部分，我们可以创建报告和仪表盘来可视化支出模式并识别趋势。我们还对异常检测感兴趣，使用工具来识别需要进一步调查的异常支出峰值或异常情况。

优化（Optimize）。一旦我们了解了云支出，就该进行优化了。这个阶段侧重于识别和实施成本节约措施：

调整资源大小（Right-sizing）

分析资源利用率，并调整实例大小、存储层级和其他配置，以匹配实际需求。

利用折扣

利用云提供商提供的折扣。我们将在下一节中详细介绍这一点。

自动化成本优化

使用自动化工具来安排实例关闭、优化资源分配和强制执行成本策略。

消除浪费

识别并消除未被使用或利用不足的资源，例如空闲实例、孤立卷和未挂载的存储。

运营（Operate）。运营阶段旨在建立管理和监控云成本的持续流程。我们将 FinOps 实践嵌入到我们的文化和工作流程中：

预算和预测

为云支出设定清晰的预算，并使用预测工具来预测未来成本。

持续监控

持续跟踪云支出和使用模式，以识别任何预算偏差或意外的激增。

自动化成本优化

通过实施脚本或使用云提供商工具来自动化常见任务，以处理诸如资源清理、调整大小和预留管理等日常活动。

通过反复迭代这三个阶段，优化成为一个持续进行的项目。

现代云成本管理挑战

随着云环境因多云和混合基础设施变得日益复杂，传统的成本管理方法已显得不足。需要采用现代策略来有效驾驭和控制这些环境中不断增长的云支出。

451 Research 受 Oracle Cloud Infrastructure 委托进行的一项 2023 年研究发现，98% 的企业正在使用或计划使用至少两家云基础设施提供商。此外，31% 的企业正在使用四家或更多云基础设施提供商。多云或混合方法有助于防止对单一供应商的依赖，并使公司能够根据不同提供商的特定优势选择最佳服务。

多云或混合方法还可以通过将工作负载分布到多个云来最大限度地减少中断的影响。这种方法还可以帮助公司遵守规定数据存储位置的数据主权法律。此外，从不同提供商中挑选最具成本效益的服务和定价模型的能力，为优化云支出带来了重要的机会。

除了增加操作复杂性之外，管理多云或混合环境中的云成本还带来了独特的挑战。每个云提供商都有自己的计费系统，具有不同的格式、指标和报告工具。这使得难以获得所有平台支出的统一视图，尤其是在成本数据分散在不同部门或团队内部时。此外，云提供商具有复杂的定价模型，难以在不同平台之间进行比较。这使得准确地将成本分配给特定项目、部门或业务单元变得具有挑战性。

AI 驱动的云成本优化策略

AI 可以帮助您掌控云支出。该技术的优势在于它能够处理海量生成的数据，识别出人类几乎不可能检测到的模式和异常。将 AI 应用于云计算成本管理，不仅能为我们提供当前支出的洞察，还能以惊人的准确性预测未来的云使用情况。AI 算法，例如长短期记忆（LSTM）和双向 LSTM 网络以及决策树回归，可以提前数周甚至数月预测计算需求。在本节中，我们将探讨在不影响性能的情况下降低云账单的实用策略，并将探讨 AI 如何帮助您调整资源大小、利用折扣以及自动化成本控制，同时支持您的业务目标。

调整云资源大小

调整资源大小是负责任的云成本管理的核心。这是一种优化云资源分配以匹配应用程序和工作负载实际需求的做法。调整资源大小是一项关键策略，它确保您既不会供应不足（可能导致性能问题），也不会过度供应（可能导致不必要的成本）。团队通常在部署服务或应用程序时，出于对性能不佳的担忧，会分配超出实际所需的资源。如果没有准确的使用数据，就很难精确估计所需资源。配置不当的自动化扩展也可能导致资源过度分配。

虽然计算资源（虚拟机和容器）通常是调整资源大小工作的最初重点，但它们也适用于其他云资源，包括：

存储

调整存储大小涉及根据使用模式选择合适的存储层级和卷。这涵盖了块存储（如 Amazon EBS 或 Google Persistent Disk）、文件存储和对象存储（如 Amazon S3 或 Google Cloud Storage）。

数据库

调整数据库大小涉及选择正确的数据库实例类型、存储配置以及数据库性能和容量，以降低成本。

网络

调整网络大小包括优化负载均衡器、VPN 网关和带宽等网络资源的使用。调整大小有助于最大限度地减少与过度配置的网络资源相关的非必要成本。

FinOps 知情阶段所需的云成本可见性

有效的优化需要理解您的资源利用模式和相关成本。这就是 FinOps 知情阶段的作用所在。为了精确理解云成本，工程师必须能够访问详细的云分析数据，这些数据突出显示

了计算、存储、内存及其他资源的使用方式以及它们如何转化为实际支出。

这可能很复杂。通常，成本由财务团队管理，工程师甚至可能无法访问财务数据。即使他们能访问，他们也可能对他们的云资源如何实际转化为具体的金钱只有有限的了解。挑战在于向工程师提供清晰、简洁、可操作的可见性，让他们了解其应用程序所消耗的特定资源的相关成本。这在复杂、多应用或多租户的云环境中尤其困难，因为成本可能会分散到许多资源上。

现代云成本管理工具，如 Harness CCM 和各种云提供商工具，通过向工程师提供自助式可见性来弥合这一鸿沟，使他们能够看到其应用程序、微服务、集群和环境的真实成本。这些工具赋予工程师权力，提供他们直接管理云成本所需的上下文，而无需依赖财务或运营团队提供信息。与仅限于基本基础设施视图的传统系统不同，现代工具与云服务集成，提供详细的应用程序级成本数据。

从知情到优化

一旦您对云使用和成本有了清晰的认识，就可以进入 FinOps 的优化阶段了。您可以根据数据做出决策，调整云资源，使其更高效、更具成本效益。通过对成本和使用模式的可见性，您可以自信地进行调整，使其同时符合性能和预算限制。

AI 驱动的工具在此变得不可或缺，消除了手动监控和调整资源以满足需求的必要。通过持续监控以及空闲资源检测和资源使用分析等功能，这些工具可以帮助识别效率低下之处，并推荐调整或自动调整 CPU、内存和存储配置。

然而，优化应循序渐进。从小规模、增量式的改变开始，并根据观察到的影响不断完善您的策略。AI 驱动的工具可以通过预测未来的使用情况，并利用过去的小幅变化作为反馈来指导后续的微调操作，从而帮助调整资源。这种迭代过程是 FinOps 运营阶段的关键方面。通过以这种方式微调您的云环境，您可以验证性能需求，确保您的应用程序高效运行，同时不牺牲用户体验或应用程序性能。

利用承诺用量定价和竞价型实例

利用承诺用量定价模型和竞价型实例是降低云成本的额外策略。

承诺用量定价涉及在特定期限内承诺使用一定水平的云资源，以换取显著折扣。由 AWS、Microsoft Azure 和 Google Cloud 等主要云提供商提供，这些模型 —— 通常被称为预留实例（Reserved Instances）或承诺使用合同（Committed Use Contracts）—— 与按需定价相比，可节省高达 80% 的成本。更长的承诺期（通常为一到三年）和更高的预付款可带来最大的折扣。例如，AWS 的预留实例对计算资源提供 30% 到 72% 的折扣，而 Google Cloud 的承诺使用合同则在计算、存储及其他服务方面提供类似的

节省。这些产品非常适合可预测的工作负载，例如稳定运行的应用程序，但需要准确的用量预测以避免过度或不足配置。

另一方面，竞价型实例通过以大幅折扣（比按需价格低 90%）利用未使用的云容量，提供了一种动态的成本削减方式。这些实例非常适合非关键、灵活的工作负载，如批处理、数据分析或开发环境，因为它们可以在极短的通知下被中断。通过将承诺用量定价用于稳定工作负载，将竞价型实例用于灵活或瞬时任务，企业可以实现成本效率和操作灵活性的强大平衡。高级工具和 AI 驱动的预测可以帮助组织有效驾驭这些模型，确保最佳的资源分配和最大的成本节约。

关键考量

在利用承诺用量定价模型和竞价型实例优化云成本时，您需要考虑它们独特的优势和挑战，以最大限度地发挥其价值，同时降低潜在风险。

承诺用量定价模型非常适合可预测、稳定的工作负载。然而，它们需要对长期（通常为一到三年）的资源使用进行准确预测。误判使用量可能导致过度承诺，造成资源利用不足和成本浪费，或者承诺不足，可能导致更高的按需费用。灵活性也受到限制，因为这些承诺会根据提供商将业务锁定在特定的实例类型、区域或服务层级。为应对这些挑战，您必须建立以下实践：

- 分析历史使用数据以提高预测准确性。
- 在可用时使用可转换或灵活选项，以根据需求变化调整承诺。
- 定期监控和优化资源使用，使其与承诺保持一致。

竞价型实例的瞬时性（可能在很少通知的情况下被终止）需要仔细规划和工作负载适应。您必须：

- 确保工作负载能够容忍中断而不会产生显著影响。
- 实施检查点或自动化作业恢复机制，以最大限度地减少中断。
- 监控市场趋势以预测竞价型实例的可用性和价格波动。

结合承诺用量定价用于稳定工作负载和竞价型实例用于灵活任务的混合策略可以提供两全其美的优势：可预测的成本节约和低成本的动态扩展。为了有效实施混合策略，您的实践必须：

- 评估工作负载特性，以确定承诺用量资源和竞价容量的适当组合。
- 利用自动化工具，如云原生自动扩缩和工作负载编排系统，优化使用。
- 持续评估和完善策略，以适应不断变化的业务需求和工作负载模式。

AI 如何提供帮助

显然，在承诺用量定价模型和竞价型实例之间进行优化很快就会变得复杂。现代 AI 工具，包括生成式 AI，可以通过准确预测、动态优化和无缝自动化来帮助克服这些挑战。

对于承诺用量定价，AI 代理（例如 Harness FinOps Agent）可以分析历史使用模式并精确预测未来的资源需求，从而降低过度承诺或承诺不足的风险。AI 工具还可以识别预留资源和按需资源的理想组合，同时持续监控和调整承诺以适应不断变化的工作负载。此外，AI 系统可以检测资源消耗异常，确保企业避免效率低下或受到处罚。

对于竞价型实例优化，AI 可以通过预测可用性和价格趋势来解决其不可预测性，从而为容忍中断的工作负载实现更智能的调度。AI 驱动的工作负载编排工具自动化任务的部署和扩展，当竞价型实例中断时，动态地将其转移到备用资源。此外，AI 优化检查点和恢复过程，确保工作负载能够高效恢复，同时最大限度地减少停机时间。这对于批处理或数据分析等任务尤为有价值。

通过整合这两种模型，AI 可以创建一种智能混合策略，平衡预留资源的成本效益与竞价型实例的灵活性和低价。它确保根据工作负载需求进行最佳资源分配，预测需求激增以主动调整资源，并为持续成本优化提供可操作的洞察。

使用 AI 管理容器成本

容器化架构在现代应用开发中扮演着重要角色。容器将应用程序及其依赖项打包成轻量级、可移植的单元，使其能够在不同环境中一致运行。这些容器在节点上运行，节点是提供底层资源的物理或虚拟机（通常称为实例）。多个节点组合形成一个集群，这是一个协同工作的机器组，用于运行和管理容器。在一个节点内，容器通常被分组为 Pod，它们共享资源并作为一个单一的操作单元一起部署。这些集群的管理通过编排完成，其中 Kubernetes 等工具自动化容器的部署、扩展和生命周期管理。

虽然容器化架构提供了无与伦比的可移植性、资源效率和跨多样环境的适应性，但管理容器化环境的云成本与管理虚拟机（VM）环境的成本有所不同。共享的底层基础设施使成本跟踪变得复杂。在 VM 环境中，每个虚拟机都是一个独立的单元，拥有固定的资源，因此成本更容易分配。然而，当多个容器运行在单个服务器实例上时，您的云账单只会提供底层服务器的使用情况，而不是单个容器的使用情况。这使得难以准确理解容器之间的使用成本，因为您需要更详细的信息来了解 CPU 和内存等资源是如何共享的。可见性不足给成本归因带来了挑战，使得在容器层面跟踪和管理费用变得更加困难。

容器化并没有消除对 FinOps 的需求；相同的财务问责制和成本优化原则对于容器化应用程序仍然至关重要。除非您依赖云托管的容器平台，否则您必须收集关于运行中的容器如何利用服务器资源的补充数据。这包括跟踪每个容器在共享服务器实例上消耗的

CPU、内存和存储的比例。将这些细粒度的资源使用数据与您的云账单信息配对，可以实现准确的成本分配，确保团队和应用程序对其资源消耗负责。如果没有这种级别的洞察力，在容器化环境中有效管理和优化成本将变得具有挑战性。

AI 再次可以发挥重要作用。AI 帮助的一个关键方式是通过智能资源分配：它分析历史使用模式和工作负载，预测容器的资源需求，建议 Pod 的最佳配置、扩展策略和节点大小。这减少了过度配置，使容器仅拥有所需的资源而不会浪费容量。此外，AI 实现了动态工作负载扩展，在低使用期间自动缩减工作负载，这补充了 Kubernetes 自动扩展功能，从而最大限度地节省成本。

AI 还通过预测和警报改善成本控制。通过分析历史使用情况和识别趋势，AI 模型可以预测未来的 Kubernetes 相关成本，提供准确的预测，帮助团队有效规划预算。警报可以通知利益相关者潜在的预算超支，从而在成本失控之前采取纠正措施。此外，AI 可以智能地在集群、节点甚至区域之间调度工作负载，以最大限度地减少云支出，同时保持性能、合规性和可靠性。

将成本节约目标与业务目标对齐

虽然 AI 可以更容易地实现成本节约，但请记住 FinOps 的原则是决策由云的业务价值驱动。我们必须权衡云成本节约与我们对质量、速度和创新的业务需求。

考虑一家大型零售公司准备在销售旺季前推出一个新平台。其主要目标是确保平台顺利成功上线，即使这意味着更高的初始云成本。按时且完美上线所带来的收入增长，以及提供卓越的客户体验，可能远比立即降低云成本更为重要。零售商可能会接受更高的初始费用，以确保成功上线，因为他们知道从长远来看可以优化云使用并降低成本。

虽然成本节约是一个关键考虑因素，但请记住，战略性投资以最大化业务效益同样重要。在考虑支出时，应同时考虑诸如实现快速扩展以加速增长、增加收入、加快交付时间、提高客户满意度和降低劳动力成本等因素。

自动化云治理和合规性

到目前为止，我们已经探讨了 FinOps 的核心原则，并研究了优化云资源的策略方法，包括调整资源大小、利用折扣以及管理容器化环境。但您如何确保这些实践得到一致应用并与组织的更广泛目标保持一致呢？这就是云成本治理发挥作用的地方。云治理框架是一套策略、程序和最佳实践，用于定义组织如何使用和管理其云资源。可以将其视为安全高效云采用和成本管理的蓝图。在本节中，我们将探讨一个强大的治理框架如何将云财务管理的各个方面联系起来。最后，我们将研究自动化和 AI 在执行治理策略中可

以发挥的作用。

实施云治理策略

为您的云策略设定明确目标是创建稳固的云成本治理框架的第一步。这有助于确保您的云运营与业务优先级（如成本管理、驱动创新和支持增长）保持一致。与您的策略相关的具体成功指标可能包括将云浪费减少 20%、实现 95% 的标签合规性或将月度成本差异限制在 5% 以内。

有效的治理策略应涵盖以下领域：

成本可见性

这包括诸如您的资源标签策略等考量，我们将在下一节中详细介绍。成本可见性策略还可以包括配置意外成本飙升的警报，使用诸如 Google Cloud Billing 或与 Slack 或电子邮件通知集成的自定义脚本等工具。

预算和预测

策略应设定团队特定的预算，并为不可预见的增长留有缓冲。

优化流程

策略应包括优化实践，例如定期分析资源使用情况并调整实例大小以更好地匹配工作负载，为可预测的工作负载预购预留实例或节省计划，以及为容错工作负载使用竞价型实例以降低计算费用。

安全与合规性

最后，策略应实施 RBAC 以根据用户角色限制资源访问。自动化检查以确保符合 GDPR 或 HIPAA 等法规，使用 Prisma Cloud 或 AWS Security Hub 等工具。

AI 和自动化使自动执行策略变得更容易，减少了错误或疏漏的可能性。例如，AI 驱动的工具可以为资源添加或检查标签，并在不遵守策略时发送实时警报。这些工具还可以帮助分析基础设施并建议可以设置的策略，以改善整体成本、安全和合规性状况。此外，它们通过标记异常活动来帮助避免意外的成本飙升，并确保系统在无需持续手动检查的情况下保持合规。

自动化还简化了容易出现人为错误的过程，例如设置访问控制或分配资源。通过自动化这些任务，您可以降低安全问题的风险，并提高云环境运行的流畅性。将自动化与明确的策略相结合，可确保云使用保持成本效益、安全并与组织目标保持一致。

通过自动化强制执行预算护栏

在强制执行预算护栏方面，自动化至关重要，尤其是在动态资源分配的环境中。预算超支通常是由资源不受控制的激增、意外的使用峰值或未能实时监控成本造成的。当支出接近预设阈值时，监控、警报和行动机制可以自动执行策略，例如关闭利用不足的资源、缩减实例规模或在支出接近预设限额时限制新的资源配置。这确保了成本保持在财务目标之内，而无需仅仅依赖容易出现延迟和错误的手动干预。

AI 可以更进一步，预测成本何时可能超出预算。该技术可以预测支出何时会超出预算，并自动采取措施加以阻止。例如，AI 系统可能会发现某个项目的云资源使用量增长迅速，并预测到月底将超出预算。系统随后可以采取行动，或许通过调整资源以选择更便宜的选项，或通知团队进行调整。例如，在大型促销活动期间，AI 可以确保所需资源到位，同时不让成本失控，自动平衡成本和性能。AI 工具还有可能将销售数据、员工数量和行业市场动态等外部业务指标与历史使用模式以及与这些外部因素的相关性相结合，将其纳入预测引擎。

通过自动化确保标签合规性

成本分配是云成本管理的基础，因为它提供了云资源如何被消耗以及由谁消耗的透明度。准确地将成本归因于特定的团队、项目或应用程序，可以建立问责制并鼓励负责任的云使用。这种透明度确保每个团队都参与到成本优化工作中。

标签是成本分配的关键。标签是附加到云资源的元数据标签，提供其用途和所有权信息。例如，您可以将“环境”标签设置为“生产”、“开发”或“测试”，以区分不同阶段的资源使用情况。您可以使用“成本中心”标签将资源链接到业务单元或预算。云账户是成本分配的另一个工具。通过创建分层账户系统 —— 通常由一个根账户管理不同环境、团队或项目的子账户 —— 您可以在保持单个账户灵活性的同时集中计费和治理。标签与云账户层级结构相结合，可以实现准确的成本跟踪和分配。例如，为虚拟机（VM）打上相关标签，可以轻松筛选和分析云账单，从而了解按项目、部门或应用程序划分的支出模式。

有效的标签策略应全面且在整个组织内易于理解。该策略应定义要跟踪的关键信息，例如项目名称、部门和应用程序名称，并使用清晰、简洁和一致的标签名称。

需要合适的工具来确保您所需的标签得到一致使用。手动打标签可能力不从心，尤其是在复杂的多云环境中。人为错误可能导致不一致、标签缺失或值不正确，从而阻碍成本分配和资源跟踪。自动化可以在此强制执行标签策略并验证所有云平台上的标签值。

自动化工具还可以定期审计标签，以识别、修复或报告不合规资源，确保整个云资产的

统一性和准确性。这不仅节省时间并减少错误，还加强了您的云治理和成本优化工作。AI 还可以帮助将多个相似的标签变体规范化为统一的标签变体，从而降低噪音。

共享平台和服务的成本分配

成本分配的另一个挑战是共享平台和服务的使用，这是现代云架构的常见特性。当基础设施或资源在团队和应用程序之间共享时，准确划分成本变得更加复杂。现代 AI 驱动的工具擅长根据资源使用模式和依赖关系来归因成本。

通过云成本管理实现环境可持续发展目标

随着组织迁移到云端，将成本管理策略与可持续发展目标相结合，可以帮助您同时降低运营费用和减少环境影响。除了通常只关注省钱之外，一种绿色的云成本管理方法还包括积极选择环保选项和跟踪您的环境足迹。以下实践说明了如何实现这些双重效益：

调整云资源大小并优化

持续监控云使用情况，以识别并消除未被充分利用或空闲的资源。调整资源大小确保只配置必要的资源，从而降低成本和能耗。实施自动扩缩机制，根据实时需求调整资源分配，防止过度配置和不必要的能源消耗。您还可以实施自动停止功能，关闭未使用的资源，从而为它们实现零碳足迹。

利用可再生能源和绿色云提供商

选择运营数据中心由可再生能源供电的云服务提供商。例如，Google Cloud Platform 在某些区域使用 100% 可再生能源，使组织能够通过环保基础设施选择来减少其碳足迹。您还应优先选择碳排放强度较低的提供商和区域。

实施成本分配和问责制

将云成本分配给每个部门、项目或团队，以提高可见性并鼓励负责任的资源使用。这不仅优化了支出，还促进了可持续发展工作的问责制。使用标签、标准化工作流程和自动化治理策略，以确保资源高效利用并符合可持续发展目标。

采用节能型工作负载调度

安排非紧急或批处理工作负载在非高峰时段或能源成本和碳排放强度较低的区域运行，进一步降低支出和排放。利用 AI 驱动的工具分析使用模式，并推荐最佳工作负载放置方案，以实现成本和环境双重效益。

将可持续性指标整合到云策略中

组建“绿色团队”，负责设定和跟踪温室气体排放、能源消耗和水资源使用等可持续性目标。您还可以将可持续性 KPI 纳入云管理仪表盘（例如 AWS 提供的仪表盘），并通过游戏化方式激励团队寻找创新方法，以同时降低成本和环境影响。

跟踪碳足迹

使用提供碳足迹跟踪功能的云管理平台，以衡量云使用的环境影响并生成碳抵消。诸如 Cloud Carbon Footprint 和 Harness Cloud Cost Management 之类的工具为组织提供了最佳估算，以衡量、监控和减少云支出及相关的碳排放。您还可以将云成本优化所节省的资金重新投入到碳信用或其他环保倡议中，将财务效率转化为可衡量的气候行动。

通过结合这些策略，组织可以将成本节约直接与可持续发展目标关联起来。这种方法不仅提升了运营和财务绩效，还表明了在数字时代对环境管理的明确承诺。

AI 在云成本管理中的未来

云成本管理工具正在迅速发展，以利用 AI 洞察力来简化云使用优化工作。我们看到一个未来，AI 将通过创新的新功能为云成本管理赋能。

例如，自然语言接口和对话式 AI 有望使云成本管理更易于访问。新兴的界面将使用户能够以简单、对话的方式与复杂系统交互。用户无需浏览仪表盘或解读详细报告，只需提问“本月我的云支出主要由什么驱动?”或“哪些服务超出了预算?”即可获得清晰、可操作的答案。这降低了非技术利益相关者的技术门槛，并使组织内的各个团队都能够使用云成本数据。例如，FinOps 实践者可以使用集成到 Slack 或 Microsoft Teams 中的对话式 AI 工具，请求“显示超出预算的前五个项目”，并即时获得列表以及优化建议。

我们还看到 AI 在平衡成本节约和最小化云使用对环境影响方面发挥着越来越重要的作用。优化工作负载、调整资源大小以及使用节能的云区域通常可以实现这两个目标，尽管可持续发展工作有时可能需要前期投资。AWS、Google Cloud 和 Azure 已经提供了诸如客户碳足迹工具（Customer Carbon Footprint Tool）、碳感知套件（Carbon Sense Suite）和可持续性计算器（Sustainability Calculator）等工具，以帮助深入了解云使用的碳影响。

这些只是 AI 将为帮助我们管理云使用带来的一些功能示例。云服务既强大又复杂。现在和未来，使用 AI 和现代工具将帮助您优化资源分配、利用成本效益高的定价模型，并在云中实现更大的财务控制和可持续性。

总结

随着各行业云支出的激增，管理云成本已成为组织日益复杂且紧迫的问题。从传统的本地基础设施向基于云的环境的转变带来了更大的敏捷性和可扩展性，但也引入了新的成本不可预测性和运营复杂性。为此，FinOps 学科应运而生，强调跨职能协作、共同责任以及利用实时数据来优化云支出并最大化业务价值。

现代策略利用 AI 和自动化来解决资源过度配置、多云复杂性以及成本分配等问题，从而实现调整资源大小、利用承诺用量定价和竞价型实例以及高效管理容器化环境等实践。AI 驱动的工具改善了可见性、预测能力和治理策略的执行。它们自动化诸如标签、异常检测和预算护栏等任务，以确保成本控制、合规性，并与业务和可持续发展目标保持一致。最终，将 AI 和自动化整合到云成本管理中，赋能组织优化支出、提高运营效率并支持环境可持续发展倡议。

第 10 章：平台工程方法论：现代 DevOps 的新范式

前面的章节描绘了现代软件交付的诸多系统和实践。这段漫长的旅程也揭示了现代软件团队必须面对的艰巨复杂性。此外，现代软件开发倾向于将许多以前由运维团队处理的问题 —— 安全、可观测性和基础设施管理工作 —— “左移”到开发环节。如果就连经验最丰富、资源最充足的开发团队都因这种复杂性和额外责任而感到吃力，我们又该如何应对？

平台工程应运而生，旨在帮助现代软件组织解答这一核心问题。它是一门设计、构建和维护内部开发者平台的学科，旨在为软件交付提供集成的工具和基础设施。虽然前几章探讨了现代软件交付的各个组成部分，但本章我们将深入探讨平台工程如何将这些能力整合到一个为开发团队服务的协同平台中。

在本章中，我们将探讨组织如何构建和运行高效的平台工程团队。我们将研究这些团队在组织中的定位，以及它们在实现快速、安全交付方面所扮演的角色。然后，我们将深入探讨构建和运营高性能平台团队的实际方面 —— 从团队结构到日常运营无所不包。我们将探索衡量开发者平台有效性的具体方法，确保我们的投资能带来真正的价值。我们还将讨论标准化与团队自主性之间的平衡 —— 如何在提供保障的同时不扼杀创新。最后，我们将审视平台可持续演进的策略，确保我们的平台能与组织的需求共同成长。

为何选择平台工程？

我们理解，传统的开发者工具方法通常要求开发团队独自应对复杂的工具和实践环境。平台工程将内部开发者平台视为一种战略产品，将开发团队视为有价值的客户。这一转变正当其时，因为软件实践的快速演进给开发者带来了认知负荷危机，他们必须同时处理日益增长的责任。平台工程旨在解决这一危机，我们将深入探讨其商业价值以及它如何增强协作的 DevOps 文化。

开发者的认知负荷危机

我们工具链中添加的每一个新工具，以及交付流程中加入的每一个新实践，都承诺能加速交付或提高软件质量。然而，其累积效应可能给我们的开发团队带来不可持续的认知负担。试想一下，开发者必须同时处理版本控制系统（SCM）、CI/CD 流水线配置、基础设施即代码（IaC）模板、安全扫描工具、部署策略、监控系统以及许多其他专业工具。每种工具都有其自身的复杂性、最佳实践和故障模式。

随着团队在整个开发生命周期中采用 AI 驱动的工具，这种认知负担尤为突出。虽然这些工具承诺加速交付，但它们通常需要深厚的专业知识才能有效实施。平台工程可以封装这种复杂性，通过标准化的接口和模板让 AI 能力易于访问，无需每个开发者都成为 AI 专家。

数据揭示了一个令人担忧的现实：Harness 最近对工程领导者进行的一项调查发现，78% 的开发者至少有 30% 的时间花在手动、重复的任务上，而不是编写代码。时间被运维职责和工具管理所消耗 —— 这些活动虽然必要，却将开发者从他们最有价值的工作中拉开：为业务问题创造创新解决方案。除了时间损失本身，更令人担忧的是专注力的分散，这会产生认知负担，直接影响交付质量和速度。遗留流程往往会加剧这一挑战，产生低价值的工作，阻碍深度和创造性思维。

上下文切换的代价是巨大的。当开发者不断在编写应用程序逻辑、调试流水线、调查安全警报和排查生产问题之间切换时，每一次转换都会带来精神上的损耗。这种损耗不仅会减慢功能交付速度，更从根本上破坏了实现开发者卓越的条件。深度、不受干扰的专注时间能驱动软件质量和创新。当开发者不断在编码和运维任务之间跳跃时，技术卓越和创造性问题解决能力都会受到影响，从而导致技术债务的累积。

这个问题不仅影响生产力指标，还直接影响士气、员工留存和吸引顶尖人才的能力。最优秀的工程组织都明白，卓越的开发者体验 —— 即工程师能花更多时间解决问题，更少时间与低效作斗争 —— 不仅能带来更好的软件，还能培养出顶尖人才蓬勃发展并留下的文化。一个成功的平台将解决这些不满的根源，开发者会蜂拥而至。如果大多数开发者都必须被迫使用某个平台，那么该平台或其推广方式很可能存在问题。

从工具链到平台即产品

平台工程师通过创建能够让开发者最大限度地提高生产力的环境来直接解决开发者体验问题。这意味着设计、构建和维护底层的开发者平台，以实现应用程序和服务的顺畅开发、部署和运营。

开发者平台通常涵盖多个领域，如图 10-1 所示。这包括一个门户、CI/CD 流水线和

IaC。确保安全、合规和云成本合规的自动化措施贯穿始终。

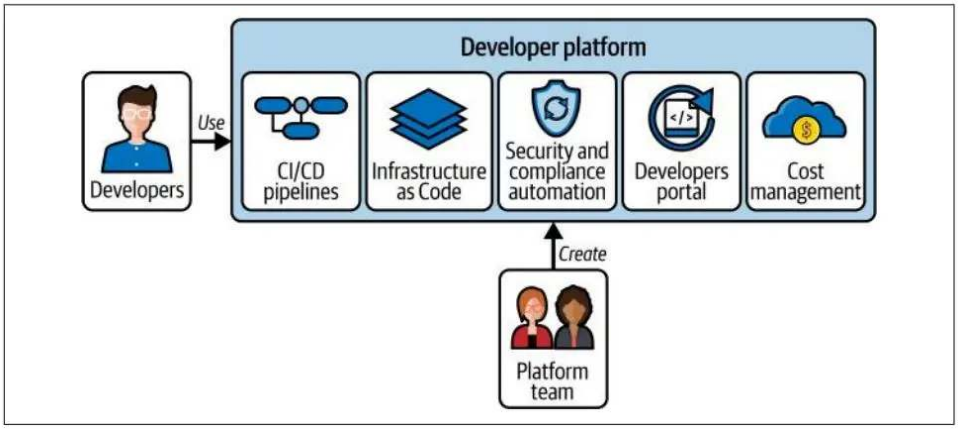

图 10-1. 开发者平台能力

开发者平台可以通过多种方式构建：平台团队可以从各种工具中组装它们，组织可以从 GitLab 或 Harness 等供应商那里购买预打包解决方案，或者他们可以使用 Humanitec 等编排工具在现有工具集之上创建一个统一层。实际上，大多数实施方案都是这些方法的组合，而不是遵循单一的纯粹策略。团队通常会选择一个策略，并对组织重要但不完全符合策略的事项做出一些调整。作为务实的解决问题者，工程师们经常混合使用策略，根据关键的组织需求调整他们选择的方法，并最终使其奏效。

平台提供“铺就的道路” —— 开发团队可以自信遵循的成熟模式和实践。“铺就的道路”通常以应用程序或基础设施模板的形式出现。以一个标准的 Web 应用程序模板为例。从零开始，开发者可能需要几天时间才能手动将基础功能连接起来并设置部署 —— 这是一项繁琐且重复的工作。而平台提供的模板预配置了必要的框架。这包括指标收集的标准化方法、容错模式、具有合理默认值的安全配置，以及带有请求跟踪的结构化日志 —— 所有这些都集成在一个内聚的框架中。模板可以超越应用程序代码，包括基础设施定义和部署流水线配置，形成一个全面的应用程序基础。

此外，平台通常还提供模板，这些模板在应用程序层面标准化了处理认证、日志记录和错误处理等横切关注点的方法，并封装了组织的最佳实践。运维层包括集成安全扫描、负载测试和自动回滚程序的部署流水线。基础设施层利用自动化来配置具有适当安全组、监控配置和灾难恢复程序。

这些能力通过自助服务门户或接口暴露，这些门户或接口抽象了底层复杂性，同时维护了安全和合规的护栏。例如，平台可以提供一个用于配置数据库的 API，该 API 根据组织标准自动配置备份计划、加密和访问控制。

其好处显而易见：模板自动化了繁琐的设置和配置工作，同时嵌入了最佳实践、安全和合规标准，而不是让团队自己拼凑工具。这使得开发者效率更高 —— 有了模板化的代

码，他们可以立即专注于构建功能。此外，当他们转到另一个项目时，无论是由于团队变更还是需要更新依赖项，由于熟悉的环境，他们会感到更舒适、更高效。同时，这种增加的一致性降低了风险，因为管理少量标准工具和模板的风险比管理组织中散布的独特“雪花”更容易。

值得注意的是，平台团队提供的具体模板和自动化工具会因组织而异。没有“一刀切”的方法。平台团队的重点必须是精准地解决其内部开发团队的独特需求和用例。一个拥有严格合规要求的大型金融机构适用的方案，很可能与为快速发展的初创公司构建的平台截然不同。关键在于平台团队要深入理解其“客户” —— 开发团队 —— 并根据他们具体的痛点和工作流程量身定制其产品。在本章后面，我们将深入探讨平台工程所需的产品思维，以确保团队真正成为一个以开发者体验为核心的服务提供商。

平台工程的商业价值

平台工程的商业价值建立在一个简单的前提之上：通过简化软件开发和运维，我们可以实现开发者和运维的生产力提升，从而提高团队效率并降低与风险相关的成本。考虑一下开发者时间的成本。如果大多数开发者将大约 30% 的时间花在重复、无附加值的任务上，对于一个只有 250 名开发者的组织来说，这将转化为可观的可回收成本。在我们调查的公司中，开发者的年平均收入为 107,599 美元，相当于每年每位开发者损失超过 32,000 美元的生产力。对于拥有 250 名开发者的组织来说，这代表了每年因开发时间损失而产生的 800 万美元隐性成本。

平台工程提供的开发者平台自动化了许多此类重复任务，从而挽回了损失的时间。平台标准化和集中式平台开发消除了跨团队重复的努力。这使得运维团队能够在现有人员编制下支持更大的应用程序组合，同时保持一致的安全和合规标准。

平台工程的投资回报率（ROI）超越了开发者生产力。它解决了让团队夜不能寐的业务风险。开发者平台的铺就之路通过消除跨团队不一致实施所产生的安全漏洞，降低了安全漏洞的风险和潜在成本。凭借标准化的部署流程和监控实践，平台团队可以降低服务中断的频率和影响。通过将高可用性和灾难恢复的成熟模式嵌入到平台组件中，组织即使在事件发生期间也能保持业务连续性。同时，内置的合规控制和自动化审计追踪帮助组织避免代价高昂的监管违规。

平台工程还可以作为更广泛组织举措的强大加速器，例如云迁移或应用程序现代化。通过提供标准化和自动化的基础，平台工程创建了可重复的路径，从而加快了时间表并降低了风险。

支持协作式 DevOps 文化

平台工程支持开发、运维和安全团队之间的协作，这是我们在前面章节中反复提及的主题。一个统一的、自助服务的平台充当着协作的桥梁。平台不再是孤立的职责和潜在的摩擦点，而是带来了更集成的生态系统，其中运维和安全要求在平台的铺就之路上得到直接和自动的解决。

这些“铺就之路”为开发者提供了预先批准的、安全的模式和自动化工作流程，它们天然地融入了安全最佳实践。

这在受监管的环境中尤其强大：平台团队无需为每个应用团队重复实施新的规定，从而避免了极可能出现的不一致性；相反，平台团队可以在平台层面一次性实施这些规则，确保所有团队都符合要求。开发者可以专注于创造业务价值，同时通过平台护栏自动遵守运维和安全标准。最终结果是一个更加精简、安全的软件交付流水线。

创建和运营平台团队

既然我们理解了平台工程的价值（“为什么”），接下来我们将转向实施（“如何做”）。在本节中，我们将探讨如何建立和运营高效的平台工程团队。我们将研究成功团队的关键特征，讨论使平台团队与开发需求紧密结合的协作模式，并审视支持可伸缩、高性能平台工程职能的运营模式。

平台团队的关键特征

最成功的平台工程团队不仅拥有深厚的技术专长，还具备产品洞察力和客户同理心。从平台领导层开始：领导者需要具备深厚的技术功底，以理解组织中开发者面临的挑战，同时还要具备战略眼光，使平台与业务目标保持一致。他们必须在工程师和业务利益相关者之间保持信誉，弥合技术与战略之间的鸿沟。理想的领导者应在开发、运维和安全这三个领域拥有深厚的背景。这种侧重将有助于指导团队和组织朝着正确的方向前进。

平台团队本身应该成为您开发组织的一个缩影，涵盖开发、安全和运维方面的专业知识。这种跨职能的知识使团队能够创建集成的解决方案，以满足开发者的所有需求。团队应包括在安全和合规等关键瓶颈领域有经验的工程师，以及精通开发、企业架构和现有工具的工程师。随着 AI 成为软件交付的基础组成部分，平台团队受益于至少包含一名具有 AI/ML 运维专业知识的成员。这个角色弥合了数据科学和软件交付之间的差距，帮助团队有效地将 AI 驱动的工具集成和管理到平台中。他们确保 AI 组件保持可靠、可解释，并符合组织治理要求。

请记住，平台是一种产品，而开发者是其客户。为了确保平台基于开发者需求而非仅仅

平台团队偏好而发展，团队需要强大的产品管理能力。这包括用户研究、路线图制定和采用率衡量等技能。通过理解开发者需求并衡量平台有效性，团队可以确保平台始终是整个组织的宝贵资产。

有效的协作模式

在开始平台实践时，首要考虑的问题之一是确定团队将如何与更广阔的组织进行协作。一个成功的平台团队需要一种协作模式，这种模式能深入理解开发者需求，并促进组织内部的有效协作。平台团队还必须以周到且结构化的方式与其“客户” —— 开发团队 —— 进行互动。

“沉浸式项目”是一些组织中效果良好的协作模式的一个例子。在这种模式下，平台工程师会暂时嵌入到各个开发团队中。这种亲身体验的方式让团队能够洞察开发者日常面临的挑战；它培养了同理心，并加深了对他们需求的理解。通过与开发者并肩工作，平台工程师可以识别痛点、瓶颈和改进机会。这确保了平台在保持中心化治理结构的同时，与特定团队的独特需求同步发展。当你刚开始进行平台工程时，这种模式尤其有效，因为它有助于深入了解团队的挑战和限制。

另一个选择是建立一个卓越中心（Center of Excellence）。这种类型的平台团队作为一个独立的、跨职能的团队运作，服务于所有开发团队。他们就平台功能提供反馈，倡导采用，并协助将平台能力整合到开发工作流程中。组织内部的协作确保平台与开发社区不断演进的需求保持一致。这种模式适用于项目多样化的大型组织，因为它提供了清晰的所有权和集中的最佳实践，同时减少了重复工作。

此外，混合模式（即平台工程师既作为集中式专家又作为嵌入式资源）提供了两全其美的优势 —— 工具和流程的一致性与对特定产品挑战的深入了解相结合。

选择正确的模式取决于您组织的规模、复杂性和战略重点。您平台的成熟度阶段也可能是一个因素。随着平台成熟和用户群增长，协作模式需要扩展。虽然对于早期采用者来说，高接触支持可能可行，但对于广泛采用来说，更具可伸缩性的方法是必要的。通过全面的文档和直观的工具实现的自助式入职，允许团队无缝且自主地与平台集成。

最后，尽管这些模式都为平台的成功提供了结构基础，但它们必须通过强大的机制来衡量和验证平台的有效性。在本章后面，我们将探讨如何确保建立系统化的反馈循环，以衡量开发者满意度和平台采用情况。

高效的运营模式

协作模式指导平台团队如何与开发团队互动，而运营模式则定义了平台团队的内部流程和原则。运营模式对于服务开发团队同时保持高运营标准至关重要。它决定了团队如何

运作、分配资源以及与更广泛的组织互动。一个有效的模式能在赋能与控制之间取得平衡。

一个强大的运营模式会优先考虑自助服务能力，从而赋能开发团队快速且独立地行动。平台团队通过自动化和模板产品维护适当的护栏，这些产品封装了组织的最佳实践。

清晰、及时和易于访问的文档是有效运营模式的另一个特征。文档应赋能开发者理解和采用平台功能，而无需平台团队的亲自指导。

最后，您的运营模式应能同时处理战术和战略需求 —— 为关键问题实施服务水平协议（SLA），同时预留专用时间用于平台改进和处理开发者反馈。支持轮岗和工程冲刺等技术可以帮助团队管理这种平衡。

定义您的平台战略

一个连贯的平台战略应完全聚焦于您的开发者需求。它考虑了您组织的限制以及对您业务最重要的目标。在本节中，我们将讨论您的战略应如何清晰地阐明一套指导决策的原则。我们将探讨深入理解平台客户应如何驱动初始范围的确定。最后，我们将讨论您在通过平台实现标准化与在需要时允许团队灵活偏离之间可能遇到的挑战。

设定平台原则

平台战略始于清晰的原则，这些原则阐明了您的平台团队将如何处理解决方案。当团队被拉向许多不同的方向并被要求解决日益增长的问题时，清晰的原则就像指南针。它们指导决策，并为评估权衡、解决冲突以及在平台开发的复杂性中保持专注提供了框架。一旦您定义了这些原则，将其在整个组织内分享以获得一致性，这将有助于团队取得成功。

以下原则是任何平台战略都应具备的基础：

- 开发者体验和效率是平台设计的主要驱动力。每项功能都应旨在降低认知负荷并简化开发工作流程，而不是增加复杂性。
- 安全和合规要求无缝嵌入平台。这使得开发者“做对事情”比绕过控制更容易。这种方法在不阻碍生产力的情况下，培养了一个安全的开发环境。

平台演进由可衡量的开发者需求和可证明的业务成果驱动。

平台演进并非仅由平台团队的技术偏好或任何特定开发团队的强烈意见驱动。平台团队将征求开发者的反馈，跟踪平台使用情况，并衡量其对部署频率和交付周期等关键指标的影响。

一个关键的战略决策是平台采用应该是强制性的还是可选的。强制性平台可以推动一致性和标准化，但有降低提供卓越开发者体验的激励的风险。另一方面，可选采用则迫使平台团队赢得信任并证明价值，从而带来更多以用户为中心、创新的解决方案。虽然这种方法可能造成碎片化，但它在实践中而非仅仅理论上促进了卓越。有些组织从可选采用开始，后期再引入强制性措施，以巩固成果并吸引后来的采用者。

平台反模式与冲突解决

与您的平台战略应该包含什么同样重要的是，它应该避免什么。常见的、会破坏平台成功的反模式包括：

完美主义而非进展

延迟平台发布直到“完美”，通常意味着开发者在此期间会创建自己的解决方案，从而使最终的采用变得更加困难。

技术驱动的开发

构建平台功能仅仅因为它们在技术上有趣或很酷，而不是因为它们解决了真实的开发者问题。

未证明价值的强制采用

在平台价值尚未证明之前就强制团队使用，会产生抵触情绪，并可能长期损害平台的声誉。

当原则相互冲突时（这不可避免），拥有一个清晰的优先级框架会有所帮助。例如，当安全要求与开发者体验发生冲突时，大多数组织需要一种结构化的方法来解决这种矛盾。成功的平台团队通常会优先考虑：

1. 具有监管风险的安全和合规要求
2. 针对高频活动的开发者体验
3. 运营一致性的标准化
4. 创新和灵活性

这种层级结构有助于团队在原则冲突时做出一致的决策。面对此类冲突时，平台团队应记录矛盾、决策过程和最终解决方案，为未来的决策创建先例。

允许这种模式以安全之名支持导致糟糕开发者体验的选择是危险的。你已经种下了规避或操纵平台的种子。将此类决定视为必要的权宜之计，然后努力开发一种更高效、更愉快的方式来满足治理控制，才是长期成功的关键。

了解您的平台受众

回到我们的第一条原则：开发者体验和效率是平台设计的主要驱动力。一个成功的平台战略始于深入了解您的开发团队需求以及更广泛的组织背景。沉浸式协作模式在这里可以发挥作用。通过这种模式，平台工程师会暂时嵌入到各个应用团队中。通过与应用开发者并肩工作，并通过密切观察和提问来了解他们的工作，平台团队成员能够更好地识别常见的摩擦点、瓶颈和改进机会。在考虑时，既要关注阻碍开发者生产力的即时痛点，也要关注长期的战略目标。

选择平台范围

一旦您开始对平台受众有了清晰的认识，下一步就是定义平台功能的初始范围。最有效的方法是从小处着手，专注于能够立即提升开发者生产力的基础要素。精简的基础设施配置、自动化交付流水线和集成安全自动化都是很好的例子。随着平台成熟和组织需求演变，您可以逐步扩展到更高级的领域，例如创建全面的开发者门户或引入自助分析工具。关键在于抵制一次性解决所有问题的诱惑 —— 成功的平台会稳步增长，以所展示的价值和其服务团队的持续反馈为指导。

一个实用的路线图示例

让我们通过一个实际例子来说明如何根据对受众的理解来选择初始范围。一个有用的方法是，将重点放在那些已经在推动快速创新的团队和用例上。例如，正在采用新技术或向云平台跃迁的团队常常面临严峻的挑战，即使是一个简单的平台也能迅速帮助解决这些问题。

在我们的例子中，我们决定将精力集中在支持一个正在向 AWS 和 Kubernetes 上的微服务转型的应用团队。这个团队正 struggling with 标准化的基础设施和部署模式。通过解决这一具体需求，我们可以展示平台的价值，并在组织内部获得牵引力。在确定早期采用者团队时，我们谨慎地选择了一个积极参与平台产品改进并乐于提供频繁反馈的团队。

在这种情况下，我们首先创建一条“铺就之路”，以简化常见任务。我们自动化了从仓库创建到 CI/CD 流水线配置和基础设施配置的整个新微服务创建过程。一开始，这不需要涵盖流水线中的所有工具。我们专注于只包含核心的构建、部署和治理层，并将组织标准直接嵌入到我们的模板中。根据我们的第二条原则，安全要求、合规控制和运维最佳实践都已内嵌。我们与合规和安全团队紧密合作，以确保我们的自动化模式符合他们的要求。目标是，通过使用我们的平台，应用团队默认就能“做对事情”。

另一个容易取得成功的建议是，考虑将审计帮助作为一项服务，作为平台的一部分提供

给应用团队。构建自动代表他们回答某些审计问题的功能。因为您在构建系统时就考虑到了内部审计，所以这将很容易提供，并有助于推动采用并取悦您的用户。

平衡标准化与灵活性

在建立平台工程实践时，您可能会发现在通过平台路径标准化软件交付与赋予团队所需（或仅是想要）的灵活性之间存在挑战。通过标准化平台路径实现一致的软件交付，是确保平台可靠性、运营效率以及最终业务价值的关键。然而，您的平台应允许一定程度的团队自主性；团队应能够创新，以找到最适合其特定需求且高效的解决方案。一个过于主观且路径僵化的模板只会阻碍您平台的采用。

解决这一挑战的一种方法是将模板组件分为以下三类：

强制性组件

这组组件处理日志配置、监控设置和安全控制等功能。这些都是任务关键型关注点（安全、可观测性、合规性），并且严格标准化。它们的包含是强制执行的。

可配置组件

这些组件包括资源扩展配置、缓存设置和数据库连接。团队应能根据需要进行配置，而不会影响标准化的总体目标。

扩展点

最后，扩展点应允许团队自定义健康检查、配置专用中间件以及定义团队特定指标等方面。清晰的 API 文档提供了这些扩展点，定义了标准化组件和灵活组件之间的明确接口。

通过这种模块化方法，团队可以使用铺就的路径模板，同时又具有根据自身需求进行调整的灵活性。构建高吞吐量服务的团队可能会自定义扩展配置并添加专门的性能指标，而构建安全敏感服务的团队可能会添加额外身份验证中间件和审计日志。

治理在这种平衡中起着至关重要的作用。有效的平台利用策略即代码（PaC）和自动化验证来创建护栏，防止严重问题，同时允许在安全范围内进行偏离。例如，您可能不会强制规定每一种技术选择，而是实施自动化检查，以验证是否满足了关键要求，而无论具体实现如何。这种方法允许团队创新，同时确保维持基本标准。

平台度量与演进

平台战略就绪后，下一步是衡量其有效性并推动其采用。本节将探讨如何定义和跟踪指标以证明平台价值，然后审视鼓励开发者参与和平台利用的策略。

衡量平台成功

不衡量，便无法改进。将您的平台视为一个同时拥有技术和业务关键绩效指标（KPI）的产品。各项指标应帮助您评估平台对开发者和业务的价值。关键指标类别包括：

开发者生产力

团队使用平台能以多快的速度发布功能？跟踪部署频率、变更交付周期和平均恢复时间（MTTR）等指标。利用这些指标更好地了解编写代码与管理基础设施所花费的时间。

平台采用率

团队是否真正使用平台？指标应衡量广度（活跃用户数量、利用平台的项目以及新项目入驻的百分比）和深度（团队对现有功能的利用程度）。使用模式可以帮助识别成功的产品和潜在的摩擦点。

运营效率

平台在多大程度上降低了成本或提高了运营性能？查看基础设施成本、事件发生率和支持工单量。

业务影响

最终，平台是否为业务目标做出了贡献？指标应将平台投资与组织成果联系起来，例如更快的上市时间、提高客户满意度或改进产品质量。

我们跟踪平台性能和价值创造，以验证持续改进并指导平台计划。接下来，我们将探讨鼓励平台初步采用并促进平台演进以解锁更多价值的策略。

推动平台采用

在清晰的平台愿景武装下，并基于对受众最迫切痛点的理解，我们接下来的挑战在于推动平台的采用。一个执行良好的采用策略结合了仔细的能力选择、无缝访问和积极主动的参与。

最小可行平台（MVP）方法将有助于推动早期采用。平台工作应专注于交付一组小而高价值、低投入的功能，以解决开发团队最紧迫的痛点，而不是试图一次性构建所有功

能。这些初步的成功将建立信誉并展示我们平台的潜力，从而更容易获得未来扩展的认可和资源。基于我们对受众的理解，考虑那些消耗最多开发者时间或造成最大摩擦的任务。通过首先解决这些挑战，我们可以迅速展示切实的好处，并激发对平台的兴趣。

最后，如果您是一个平台团队，您必须积极推广您的能力。构建一个出色的平台只是成功的一半；您还需要说服开发者使用它。考虑开展开发者教育项目、技术展示，并清晰地沟通平台优势和路线图。举办研讨会和培训课程，教开发者如何有效地使用平台。展示成功的用例，并强调平台对其他团队产生的积极影响。维护一个清晰的路线图，并沟通即将推出的功能和改进，以激发兴趣并鼓励采用。通过积极与开发者社区互动，平台团队可以培养采用文化，并确保平台成为开发工作流程不可或缺的一部分。

利用内部开发者门户推动平台成功

内部开发者门户（IDP）充当着开发团队与您的平台能力之间的接口。门户将所有内容汇集一处，使平台易于发现和使用。最有效的门户包括服务发现、上下文相关文档和自助服务功能，使其成为开发者与平台进行任何互动的自然起点。

核心门户组件

一个精心设计的 IDP 通常包括：

软件目录

一个集中式的服务、API 和组件注册表，包含所有权信息和依赖映射

自助服务工作流

用于常见任务的自动化流程，例如使用“黄金路径”模板创建新服务或配置环境

文档中心

在需要时出现的上下文相关、可搜索的技术资源

记分卡

显示服务成熟度、合规状态和最佳实践采用情况的指标

AI 增强的开发者体验

现代 IDP 越来越多地利用以下 AI 能力来降低认知负荷并加速平台采用：

自然语言界面

允许开发者使用会话式查询查找资源和执行工作流。

智能推荐

根据开发者的上下文和历史记录，推荐相关文档、服务和配置选项。

自动化故障排除

分析错误模式，并在开发者遇到问题时建议潜在解决方案。

预测性辅助

根据开发者的当前活动预测其需求，并主动提供相关资源。

这些 AI 能力将门户从一个被动资源转变为一个主动助手，引导开发者完成复杂操作，而无需他们成为专家。

构建一个有效的门户

为了实现可持续成功，请将您的 IDP 视为一个产品，并为其持续开发投入专用资源。通过衡量开发者采用率、通过自助服务工作流节省的时间以及新开发者达到生产力所需的时间等指标来评估其有效性。

IDP 最近已开始提供现成解决方案。Backstage 最初由 Spotify 开发，于 2020 年开源并捐赠给云原生计算基金会（Cloud Native Computing Foundation）。包括 Roadie、Spotify 和 Harness 在内的供应商已推出了简化 Backstage 采用和管理的商业产品。非 Backstage 的商业 IDP 也已上市，例如 Atlassian 的 Compass。

一个精心设计的 IDP 能够降低认知负荷，加速入职流程，并使平台采用成为阻力最小的路径，从而彻底改变开发者体验您整个平台产品的方式。

平台的可持续演进

在初步采用之后，下一步是确保平台的可持续演进。这取决于平衡开发者赋能与平台完整性，这通过策略即代码（PaC）和自动化、持续反馈循环以及对可靠性和可伸缩性的持续投资来实现。利用实现“信任但验证”的 PaC 自动化是实现这种平衡的一种方式。通过将组织标准编码为可编程策略（使用 OPA 或自定义强制执行引擎等工具），您的平台团队可以在保持合规性的同时委托控制权。

策略和策略即代码可以赋予开发团队一定的控制权，以减少瓶颈并提升敏捷性，同时又不牺牲标准化。例如，策略可以定义可接受的资源使用限制，强制执行安全最佳实践，或确保符合组织标准。开发者可以在预定义的边界内操作，知道他们的行为不会损害平台的稳定性或安全性。至关重要的是，策略还可以用于预告即将到来的变更，例如弃用旧系统或模板。通过提前发出警告，开发者有时间迁移到更新、受支持的选项，确保平稳过渡，并在旧系统最终退役时最大程度地减少中断。

AI 技术正在迅速发展，为平台团队带来了机遇和挑战。在将新兴 AI 工具整合到您的平台之前，建立一个系统化的评估方法至关重要。创建一个沙盒环境，您可以在其中使用您组织的数据针对真实世界的场景测试有前景的技术。制定明确的标准，将 AI 能力从实验阶段提升到生产就绪阶段，包括对可靠性、可解释性和治理的考虑。这种方法允许您的平台受益于 AI 进步，同时管理快速发展技术带来的风险。

平台演进必须由经验证据而非假设来指导。您的团队必须养成定期审查“平台智能三角”的习惯 —— 即平台使用指标、支持请求和开发者反馈的组合。利用这些信息来指导路线图制定，并识别开发者需求和痛点。某些功能是否未充分利用？是否有常见的支持请求表明需要改进的领域？例如，如果您注意到某个特定服务的支持工单量增加，同时使用模式下降，这可能表明存在需要立即关注的可靠性问题。定期平台健康检查应同时检查技术指标（错误率、响应时间）和采用指标（功能使用情况、团队入职成功率）。

最后，对平台可靠性和可伸缩性的持续投资是必不可少的 —— 一次重大的中断可能会抹去数月建立的信任。随着平台采用率的增长 —— 以及平台负载的增加 —— 对可靠性工程的投资变得更加关键。开发者需要相信平台在需要时可用，并且能够持续稳定运行。这包括实施

健壮的可观测性、建立清晰的事件管理流程，并与开发团队保持透明的沟通渠道。

一个实际案例：平台工程实践

举例来说，一个拥有 1400 名开发者、分布在 80 个产品团队的金融服务机构正面临着严峻的交付挑战。一项审计揭示了安全实践的不一致性，而首席技术官（CTO）则担忧速度和合规风险。同时，内部指标显示开发者将近 45% 的时间花在了非编码活动上 —— 管理流水线、配置环境以及处理安全和合规要求。

为了应对这些挑战，该组织组建了一个由六人组成的专门平台团队：一名平台工程主管、一名拥有 CI/CD 专长的高级开发者、一名安全工程师、一名运维工程师、一名拥有 Kubernetes 专长的平台工程师，以及一名兼顾文档工作的技术产品经理。

发现与战略制定

团队首先进入了密集的发现阶段，以了解开发者痛点和合规要求。他们实施了一个结构化的沉浸式项目，将团队成员嵌入到代表性的产品团队中，观察工作流程并记录挑战。这项研究揭示了常见的痛点：

- 跨团队维护独立 CI/CD 流水线的重复工作
- 安全扫描实施不一致
- 手动环境配置导致延迟和不一致

- 跨团队对合规要求理解不清且实施方式不同

同时，团队通过与首席信息安全官（CISO）团队、企业架构、合规和法务部门的研讨会，与治理利益相关者进行互动，以了解安全控制、技术标准、审计要求和数据处理要求。

基于这项研究，团队制定了具有明确原则的平台战略：

- 开发者体验驱动平台设计 —— 降低认知负荷。
- 安全与合规内置，而非外挂。
- 一切皆可衡量 —— 只关注有价值的产出。
- 平台可选但具有吸引力 —— 解决实际问题。

团队特意将初始范围聚焦在一个高影响力能力上：带有嵌入式安全控制的安全 CI/CD 流水线。

构建最小可行平台

团队在八周内交付了 MVP：一个模板驱动的流水线系统，内置安全扫描。主要组件包括：

- 包含不同应用类型最佳实践的流水线模板
- 直接集成到流水线中的预配置安全扫描
- 针对合规要求的自动化证据收集
- 通过简单的 YAML 文件实现自助服务配置

团队将所需的安全控制直接嵌入到流水线模板中，并自动运行静态分析、依赖扫描、容器扫描和合规性检查。结果反馈到一个集中式仪表板，同时自动化证据收集简化了审计准备。

通过自动化关键检查，包括所需的代码覆盖率，平台上的应用团队的变更管理流程得到了显著简化。他们很高兴获得免于参加 CAB（变更审查委员会）的特权。

平台团队与两个试点团队紧密合作，根据每周反馈完善他们的产品。文档与代码同步开发，包括技术参考资料和实用指南。

在三个月内，MVP 在试点团队中显示出令人印象深刻的成果：

- 部署时间从五天缩短到六小时。
- 安全扫描覆盖率从 40% 提高到 100%。
- 审计准备时间从几天缩短到几小时。

- 通过自动化扫描发现并修复了关键漏洞。

在六个月时，有 15 个团队（约 250 名工程师）正在使用安全流水线模板，这些团队的安全事件比非采用团队减少了 40%。

扩展平台能力

基于全面的遥测数据和用户反馈，平台团队确定环境配置为他们的下一个目标。他们也意识到，随着采用率的增长，他们需要一种更具可伸缩性的方法来进行入职和支持。

他们的下一阶段专注于两个关键能力：

- 一个作为所有平台能力单一接口的 IDP，包括：
- 带有自动化发现的服务目录
- 安全流水线的自助入职
- 集成文档和指南
- 流水线状态和安全发现的实时可见性
- 适用于常见应用模式的 IaC 模板，其中包含了以下最佳实践：
- 安全网络配置
- 正确配置的访问控制
- 符合合规性的日志和监控
- 资源限制和成本控制

这些模板与 IDP 集成，使开发者能够以最少的努力配置符合要求的环境。

扩展后的平台显著提高了采用率。在六个月内：

- 45 个团队（约 600 名开发者）正在使用安全流水线。
- 30 个团队已采用 IaC 模板进行环境配置。
- 平台每月处理超过 2000 次部署。
- 由于自助服务能力，每位用户的支持请求下降了 70%。

尽管服务了数百名开发者，平台团队仍保持六人规模，通过利用自助服务能力和自动化来扩大其影响力。

企业级采用和业务影响

为了推动更广泛的采用，平台团队制定了一个多方面的策略：

- 内部活动，展示平台能力和成功案例
- “平台冠军”计划，在每个部门识别倡导者
- 高管仪表板，展示采用指标和业务影响
- 培训计划，包括自学和讲师指导选项

首席技术官（CTO）建立了采用激励机制，平台用户在云资源和简化合规审查方面获得优先权。

随着平台采用的增长，治理方法也随之演进。安全和合规要求不再通过手动审查和文档，而是直接通过以下方式编码到平台中：

- 自动强制执行标准的策略即代码（PaC）框架
- 满足审计要求的内置证据收集
- 针对合法边缘情况的自助服务异常处理流程
- 通过平台遥测实现自动化合规报告

截至第 18 个月，平台取得了显著成果：

- 85% 的开发团队（约 1200 名开发者）使用该平台。
- 开发者生产力提高了 35%。
- 组织范围内的部署频率增加了 6 倍。
- 从故障中恢复的平均时间减少了 70%。
- 平台用户的安全事件减少了 65%。
- 审计准备时间缩短了 90%。
- 新功能上市时间缩短了 40%。

这些改进转化为切实的业务成果：更快的功能发布、对市场变化的更迅速响应、更少的停机时间以及更低的安全和合规风险。

持续改进

在旅程三年后，平台团队持续发展其产品。他们的路线图包括 AI 辅助开发能力、高级可观测性工具、扩展的安全自动化，以及基于持续研究的开发者体验改进。

本次平台工程之旅的关键经验包括：

产品思维至关重要 将开发者视为客户能驱动更好的决策。

自助服务是规模化的关键

采用敏捷方法，迭代交付。

指标驱动投资

衡量技术成果和业务影响。

治理集成创造双赢

正确的工具可以将合规性从束缚变为加速器。

这个例子展示了一个小型、专注的平台团队如何通过系统性地满足开发者需求，同时精简治理要求，从而推动组织实现重大转型。通过从高影响力能力入手，衡量成果，并将平台视为产品，该团队实现了惊人的规模，仅用六名平台工程师就服务了 1400 名开发者。

结论

当我们结束对现代软件交付的探索时，有一点是明确的：AI 不仅仅是 DevOps 工具箱中的另一个工具；它正在从根本上改变我们构建和交付软件的方式。

在本书中，我们已经看到 AI 如何影响软件生命周期的每个阶段。从仓库中的代码模式检测到测试选择优化，从识别安全漏洞到自动化云成本优化，AI 能力正迅速成为现代交付实践不可或缺的一部分。

平台工程将这些元素结合在一起，创建了一个基础，其中 AI 驱动的能力协同工作，在保持治理的同时加速交付。通过构建抽象复杂性并嵌入最佳实践的开发者平台，团队可以更多地专注于创造业务价值，而减少在传统上消耗大量开发者时间的无差异繁重工作上。

撰写本书时，用于开发者编码助手的 AI 比用于软件交付许多其他部分的 AI 更加成熟。这预示着 DevOps 团队将面临新的压力，因为创新将越来越受限于组织验证和交付这些应用的能力，而非生成代码的能力。交付卓越性，越来越依赖于 AI，将是决定谁能充分利用编码技术进步以及谁将感到沮丧的关键因素。

展望未来

向 AI 原生交付实践的转变仍处于早期阶段，但方向是明确的。有效将 AI 整合到其交付流水线中的组织将获得显著优势：

- AI 消除瓶颈并自动化常规任务，从而加速交付周期
- 通过 AI 驱动的测试和漏洞检测，提高质量和安全性

- 通过智能资源优化和自动化修复，降低运营成本
- 随着认知负荷从运维问题转向创造性问题解决，开发者体验得到提升

在实践中，这意味着部署决策将越来越多地基于复杂的 AI 分析而非仅凭人工判断。测试策略将根据代码变更动态调整，而非遵循僵化的模式。基础设施将根据应用需求进行自我优化，而无需人工调优。

开始行动

如果您希望在您的组织中实施这些实践，我们建议采取务实的方法：

- 首先找出您最痛苦的瓶颈。您的团队在哪些低价值活动上花费了最多时间？这些是您采用 AI 驱动自动化时的主要目标。
- 小步开始，持续衡量。实施聚焦的改进，验证其影响，并利用这些成功推动进一步的采用。
- 为您的开发者构建，而不是为工具构建。对您的交付平台采取产品思维，确保它真正解决了团队的实际问题。
- 内嵌治理，而非事后补救。利用您的平台使合规性和安全性成为开发流程的无缝组成部分，而不是事后考虑。

在这个新时代中蓬勃发展的组织，不一定是拥有最大工程团队或最多预算的组织。相反，它们将是那些最有效地利用 AI 来交付更好、更快软件，同时保持企业运营所需的治理护栏的组织。

这不会一蹴而就。像任何重大转型一样，它需要坚定的领导力、持续学习以及挑战既定流程的意愿。但其回报 —— 以开发速度、产品质量以及最终的业务成果衡量 —— 使得这一旅程值得 undertaking。

软件交付的未来是智能的、自动化的，并为以人为本的开发者需求而构建。我们希望本书为您提供了技术理解和实用策略，助您今天就开始在您的组织中构建这样的未来。

关于作者

Nick Durkin 是 Harness 的现场首席技术官（Field CTO），负责该公司的全球现场工程团队、售后工程团队以及部分平台业务。他此前曾在 OverOps、DataTorrent 和 Zelle (Early Warning) 担任技术和高管职务，期间负责美国政府的关键基础设施运维。他还曾担任美国国土安全部金融机构身份凭证验证服务 (FIVICS) 项目的首席架构师，并在此期间开发了多项反欺诈技术专利。这些技术目前不仅被联邦政府采用，也被全球一些大型金融机构所使用。

Eric Minick 是 Harness 的 DevOps 解决方案高级总监，专注于赋能 AI 原生流水线和提升开发者体验。二十多年来，他一直协助包括 IBM 和 UrbanCode 在内的全球企业大规模采纳 DevOps 实践。他通过写作、演讲和咨询，定期为 DevOps 社区做出贡献。

Chinmay Gaikwad 是 Harness 的产品营销总监，他将深厚的 AI 原生 DevOps 和应用安全 (AppSec) 专业知识与战略营销领导力相结合。他是一名训练有素的软件工程师，在应用安全、DevOps 实践和开发者体验方面享有盛誉，并在早期和晚期初创公司以及上市公司担任技术和营销职务，拥有丰富的成功经验。

扫码关注几米宋
了解更多

jimmysong.io