为多 DevOps 账户环境选择 Git 分支策略

AWS 规范性指导

AWS 规范性指导: 为多 DevOps 账户环境选择 Git 分支策略

Table of Contents

为多 DevOps 账户环境选择 Git 分支策略

Amazon Web Services（贡献者）

2024 年 2 月（文档历史记录）

转向基于云的方法并提供软件解决方案 AWS 可能具有变革性。这可能需要更改您的软件开发生命周期流程。通常，AWS 账户 在开发过程中会使用多个 AWS 云。选择兼容的 Git 分支策略与您的 DevOps 流程配对是成功的关键。为您的组织选择正确的 Git 分支策略可以帮助您在开发团队之间简洁地传达 DevOps 标准和最佳实践。Git 分支在单个环境中可能很简单，但是当应用于多个环境（例如沙盒、开发、测试、暂存和生产环境）时，它可能会变得混乱。拥有多个环境会增加 DevOps实施的复杂性。

本指南提供了 Git 分支策略的可视化图表，展示了组织如何实现多账户流程 DevOps 。可视化指南可帮助团队了解如何将 Git 分支策略与 DevOps 实践相结合。使用标准分支模型（例如 Gitflow、Fl GitHub ow 或 Trunk）来管理源代码存储库可以帮助开发团队协调工作。这些团队还可以使用互联网上的标准 Git 培训资源来理解和实施这些模型和策略。

有关 DevOps 的最佳实践 AWS，请查看《Well-Architect AWS ed》中的DevOps指南。在阅读本指南时，请使用尽职调查为您的组织选择正确的分支策略。有些策略可能比其他策略更适合您的用例。

目标

本指南是关于为拥有多个 AWS 账户分支机构的组织选择和实施 DevOps分支策略的文档系列的一部分。本系列旨在帮助您从一开始就应用最符合您的要求、目标和最佳实践的策略，以简化您的体验 AWS 云。本指南不包含 DevOps 可执行脚本，因为它们因您的组织使用的持续集成和持续交付 (CI/CD) 引擎和技术框架而异。

本指南解释了三种常见的 Git 分支策略之间的区别： GitHub Flow、Gitflow 和 Trunk。本指南中的建议可帮助团队确定与其组织目标一致的分支策略。阅读本指南后，您应该能够为组织选择分支策略。选择策略后，您可以使用以下模式之一来帮助您与开发团队一起实施该策略：

- 为多 DevOps 账户环境实施中继分支策略
- 为多 DevOps 账户环境实施 GitHub Flow 分支策略
- 为多账户环境实施 Gitflow 分支策略 DevOps

值得注意的是，对一个组织、团队或项目有效的方法可能不适合其他组织、团队或项目。Git 分支策略之间的选择取决于各种因素，例如团队规模、项目要求以及协作、集成频率和发布管理之间的理想平衡。

使用 CI/CD 实践

AWS 建议您实施持续集成和持续交付（CI/CD), which is the process of automating the software release lifecycle. It automates much or all of the manual DevOps processes that are traditionally required to get new code from development into production. A CI/CD pipeline encompasses the sandbox, development, testing, staging, and production environments. In each environment, the CI/CD pipeline provisions any infrastructure that is needed to deploy or test the code. By using CI/CD, development teams can make changes to code that are then automatically tested and deployed. CI/CD管道还为开发团队提供管理和护栏。他们强制执行功能接受和部署的一致性、标准、最佳实践和最低接受水平。有关更多信息，请参阅上的 “练习持续集成和持续交付” AWS。

本指南中讨论的所有分支策略都非常适合为开发团队CI/CD practices. The complexity of the CI/CD pipeline increases with the complexity of the branching strategy. For example, Gitflow is the most complex branching strategy discussed in this guide. CI/CD pipelines for this strategy require more steps (such as for compliance reasons), and they must support multiple, simultaneous production releases. Using CI/CD also becomes more important as the complexity of the branching strategy increases. This is because CI/CD建立护栏和机制，以防止开发人员有意或无意地绕过定义的流程。

AWS 提供了一套开发人员服务，旨在帮助您构建 CI/CD 管道。例如，AWS CodePipeline是一项完全托管的持续交付服务，可帮助您自动化发布管道，以实现快速可靠的应用程序和基础设施更新。 AWS CodeBuild编译源代码、运行测试和生成 ready-to-deploy软件包。有关更多信息，请参阅 AWS上的开发者工具。

了解 DevOps 环境

要了解分支策略，您必须了解每种环境中发生的目的和活动。建立多个环境可以帮助您将开发活动分成几个阶段，监控这些活动，并防止无意中发布未经批准的功能。每个环境 AWS 账户 中都可以有一个或多个。

大多数组织都列出了几种可供使用的环境。但是，环境的数量可能因组织和软件开发策略而异。本文档系列假设您有以下五个跨越开发管道的常见环境，尽管它们的名称可能不同：

- 沙箱 — 开发人员编写代码、犯错误和执行概念验证工作的环境。
- 开发 — 开发人员可以在其中集成其代码以确认所有代码均可作为一个统一的应用程序运行的环境。
- 测试 — 进行 QA 团队或验收测试的环境。团队经常在这种环境中进行性能或集成测试。
- 暂存 — 一种预生产环境，您可以在其中验证代码和基础架构在生产等效环境下是否按预期运行。此环境被配置为尽可能与生产环境相似。
- 生产 — 一种处理来自最终用户和客户的流量的环境。

本节详细描述了每种环境。它还描述了每个环境的构建步骤、部署步骤和退出标准，以便您可以继续下一个环境。下图按顺序显示了这些环境。

本节中的主题：

- 沙盒环境
- 开发环境
- 测试环境
- 暂存环境
- 生产环境

沙盒环境

沙盒环境是开发人员编写代码、犯错误和执行概念验证工作的地方。您可以从本地工作站或通过本地工作站上的脚本部署到沙盒环境。

访问

开发人员应该拥有对沙盒环境的完全访问权限。

生成步骤

当开发人员准备好将更改部署到沙盒环境时，他们会在本地工作站上手动运行构建。

1. 使用 git-sec rets (GitHub) 扫描敏感信息
2. 整理源代码
3. 生成并编译源代码（如果适用）
4. 执行单元测试
5. 执行代码覆盖率分析
6. 执行静态代码分析
7. 以代码形式构建基础架构 (IaC)
8. 执行 IaC 安全分析
9. 提取开源许可证
10. 发布构建工件

部署步骤

如果您使用的是 Gitflow 或 Trunk 模型，则在沙盒环境中成功构建`feature`分支后，部署步骤会自动启动。如果您使用的是 GitHub Flow 模型，则需要手动执行以下部署步骤。以下是沙盒环境中的部署步骤：

1. 下载已发布的文物
2. 执行数据库版本控制
3. 执行 IaC 部署
4. 执行集成测试

迁移到开发环境之前的期望

- 在沙盒环境中成功构建`feature`分支
- 开发人员已在沙盒环境中手动部署并测试了该功能

开发环境

开发环境是开发人员将他们的代码集成在一起的地方，以确保所有代码都作为一个有凝聚力的应用程序运行。在 Gitflow 中，开发环境包含合并请求中包含的最新功能，并且已准备好发布。在 GitHub Flow 和 Trunk 策略中，开发环境被视为测试环境，代码库可能不稳定，不适合部署到生产环境。

访问

根据最小权限原则分配权限。最低权限是授予执行任务所需的最低权限的安全最佳实践。开发人员对开发环境的访问权限应少于对沙盒环境的访问权限。

生成步骤

向`develop`分支（Gitflow）或分支（Trunk 或 Fl GitHub ow）创建合并请求会自动开始构建。`main`

1. 使用 git-sec rets (GitHub) 扫描敏感信息
2. 整理源代码
3. 生成并编译源代码（如果适用）
4. 执行单元测试
5. 执行代码覆盖率分析
6. 执行静态代码分析
7. 构建 IaC
8. 执行 IaC 安全分析
9. 提取开源许可证

部署步骤

如果您使用的是 Gitflow 模型，则在开发环境中成功构建`develop`分支后，部署步骤会自动启动。如果您使用的是 GitHub Flow 模型或 Trunk 模型，则在针对`main`分支创建合并请求时，部署步骤会自动启动。以下是开发环境中的部署步骤：

1. 从构建步骤中下载已发布的工件
2. 执行数据库版本控制
3. 执行 IaC 部署

4. 执行集成测试

迁移到测试环境之前的期望

- 在开发环境中成功构建和部署`develop`分支 (Gitflow) 或`main`分支（Trunk 或 GitHub Flow）
- 单元测试以 100% 的比率通过
- 成功构建 IaC
- 部署对象已成功创建
- 开发人员已执行手动验证，以确认该功能按预期运行

测试环境

质量保证 (QA) 人员使用测试环境来验证功能。他们在完成测试后批准更改。当他们批准后，分支机构就会进入下一个环境，即暂存环境。在 Gitflow 中，此环境及其以上的其他环境只能从`release`分支部署。分`release`支基于包含计划功能的`develop`分支。

访问

根据最小权限原则分配权限。开发人员对测试环境的访问权限应少于对开发环境的访问权限。QA 人员需要足够的权限才能测试该功能。

生成步骤

此环境中的构建过程仅适用于使用 Gitflow 策略时的错误修复。向`bugfix`分支创建合并请求会自动开始构建。

1. 使用 git-sec rets (GitHub) 扫描敏感信息
2. 整理源代码
3. 生成并编译源代码（如果适用）
4. 执行单元测试
5. 执行代码覆盖率分析
6. 执行静态代码分析
7. 构建 IaC
8. 执行 IaC 安全分析
9. 提取开源许可证

部署步骤

在开发环境中部署后，在测试环境中自动启动release分main支 (Gitfl GitHub ow) 或分支（Trunk 或 Flow）的部署。以下是测试环境中的部署步骤：

1. 在测试环境中部署release分支（Gitflow）或main分支（Trunk 或 GitHub Flow）
2. 暂停以供指定人员手动批准
3. 下载已发布的文物
4. 执行数据库版本控制
5. 执行 IaC 部署
6. 执行集成测试
7. 执行性能测试
8. 质量保证批准

迁移到暂存环境之前的期望

- 开发和 QA 团队已经进行了充分的测试，足以满足贵组织的要求。
- 开发团队已通过bugfix分支解决了所有发现的错误。

暂存环境

暂存环境配置为与生产环境相同。例如，数据设置的范围和大小应与生产工作负载相似。使用暂存环境来验证代码和基础架构是否按预期运行。此环境也是业务用例（例如预览或客户演示）的首选。

访问

根据最小权限原则分配权限。开发人员对暂存环境的访问权限应与对生产环境的访问权限相同。

生成步骤

无。在测试环境中使用的相同工件将在暂存环境中重复使用。

部署步骤

在测试环境中获得批准和部署后，自动在暂存环境中启动main分release支 (Git GitHub flow) 或分支（Trunk 或 Flow）的部署。以下是暂存环境中的部署步骤：

1. 在暂存环境中部署release分支（Gitflow）或main分支（Trunk 或 GitHub Flow）
2. 暂停以供指定人员手动批准
3. 下载已发布的文物
4. 执行数据库版本控制
5. 执行 IaC 部署
6. （可选）执行集成测试
7. （可选）执行负载测试
8. 获得所需的开发、QA、产品或业务审批者的批准

迁移到生产环境之前的期望

- 已成功将生产等效版本部署到暂存环境中
- （可选）集成和负载测试成功

生产环境

生产环境支持发布的产品，处理真实客户的真实数据。这是一个受保护的环境，通过最低权限分配访问权限，只有在有限的时间内通过审核的例外流程才能允许使用更高的访问权限。

访问

在生产环境中，开发人员应在 AWS 管理控制台中拥有有限的只读访问权限。例如，开发人员应该能够访问日志数据以进行日常操作。所有发布到生产环境的版本都应在部署之前通过批准步骤进行限制。

生成步骤

无。在测试和暂存环境中使用的相同工件可在生产环境中重复使用。

部署步骤

在暂存环境中获得批准和部署后，自动启动在生产环境中部署main分release支 (Git GitHub flow) 或分支（Trunk 或 Flow）。以下是生产环境中的部署步骤：

1. 在生产环境中部署release分支（Gitflow）或main分支（Trunk 或 GitHub Flow）
2. 暂停以供指定人员手动批准

3. 下载已发布的文物
4. 执行数据库版本控制
5. 执行 IaC 部署

基于 Git 的开发的最佳实践

要成功采用基于 Git 的开发，必须遵循一套最佳实践，以促进协作、维护代码质量并支持持续集成和持续交付 (CI/CD)。除了本指南中的最佳实践外，还请查看《Well-Architect AWS ed 指南》DevOps 。以下是基于 Git 的开发的一些关键最佳实践： AWS

- 保持少量和频繁的更改 — 鼓励开发人员提交较小的增量更改或功能。这样可以降低合并冲突的风险，并且可以更轻松地快速识别和修复问题。
- 使用功能切换-要管理不完整或实验性功能的发布，请使用功能切换或功能标志。这可以帮助您在生产环境中隐藏、启用或禁用特定功能，而不会影响主分支的稳定性。
- 维护强大的测试套件 — 全面、维护良好的测试套件对于及早发现问题并验证代码库是否保持稳定至关重要。投资于测试自动化，并优先修复任何失败的测试。
- 采用持续集成 — 使用持续集成工具和实践自动构建、测试代码变更并将其集成到`develop`分支（Gitflow）或分支（Trunk 或 `main` Fl GitHub ow）中。这可以帮助您尽早发现问题并简化开发过程。
- 执行代码审查 — 鼓励对代码进行同行评审，以保持质量、共享知识并捕捉潜在问题，然后再将其整合到分`main`支中。使用拉取请求或其他代码审查工具来简化此过程。
- 监控并修复损坏的构建-当构建中断或测试失败时，请优先尽快修复问题。这样可以使`develop`分支（Gitflow）或`main`分支（Trunk 或 GitHub Flow）保持可释放状态，并最大限度地减少对其他开发者的影响。
- 沟通和协作-促进团队成员之间的开放式沟通和协作。确保开发人员知道正在进行的工作和对代码库所做的更改。
- 持续重构 — 定期重构代码库以提高其可维护性并减少技术债务。鼓励开发人员让代码保持比他们发现的更好的状态。
- 使用短期分支执行复杂任务-对于较大或更复杂的任务，请使用短期分支（也称为任务分支）来处理更改。但是，请务必缩短分支寿命，通常少于一天。尽快将更改合并回分`develop`支（Gitflow）或`main`分支（Trunk 或 GitHub Flow）。对于团队来说，规模更小、更频繁的合并和审阅比一个大型合并请求更容易使用和处理。
- 培训和支持团队 — 为刚接触基于 Git 的开发或在采用基于 Git 的最佳实践方面需要指导的开发人员提供培训和支持。

Git 分支策略

本指南按从最少到最复杂的顺序详细描述了以下 Git-based 分支策略：

- Trunk — Trunk-based 开发是一种软件开发实践，其中所有开发人员都在单个分支上工作，通常称为`trunk`或`main`分支。这种方法背后的想法是，通过频繁集成代码更改并依靠自动测试和持续集成，使代码库保持持续可发布的状态。
- GitHub Flo GitHub w — Flow 是一个基于分支的轻量级工作流程，由开发。 GitHub它基于短寿命`feature`分支的概念。当某项功能完成并准备部署时，该功能将合并到分`main`支中。
- Gitflow — 使用 Gitflow 方法，开发是在各个功能分支中完成的。批准后，您可以将`feature`分支合并到通常命名的集成分支中`develop`。当`develop`分支中积累了足够多的功能时，就会创建一个`release`分支来将这些功能部署到上层环境。

每种分支策略都有优点和缺点。尽管它们都使用相同的环境，但它们并不都使用相同的分支或手动批准步骤。在本指南的这一部分中，详细查看每种分支策略，以便您熟悉其细微差别，并可以评估它是否适合您组织的用例。

本节中的主题：

- 树干分支策略
- GitHub 流量分支策略
- Gitflow 分支策略

树干分支策略

Trunk-based 开发是一种软件开发实践，其中所有开发人员都在单个分支上工作，通常称为`trunk`或`main`分支。这种方法背后的想法是，通过频繁集成代码更改并依靠自动测试和持续集成，使代码库保持持续可发布的状态。

在基于主干的开发中，开发人员每天多次向`main`分支提交更改，目标是进行小规模的增量更新。这可以实现快速反馈循环，降低合并冲突的风险，并促进团队成员之间的协作。该实践强调维护良好的测试套件的重要性，因为它依赖于自动测试来尽早发现潜在问题，并确保代码库保持稳定和可发布。

Trunk-based 开发通常与基于功能的开发（也称为功能分支或功能驱动开发）形成鲜明对比，在这些开发中，每个新功能或错误修复都是在自己的专用分支中开发的，与主分支是分开的。在基于主干的开发和基于功能的开发之间做出选择取决于团队规模、项目要求以及协作、集成频率和发布管理之间的理想平衡等因素。

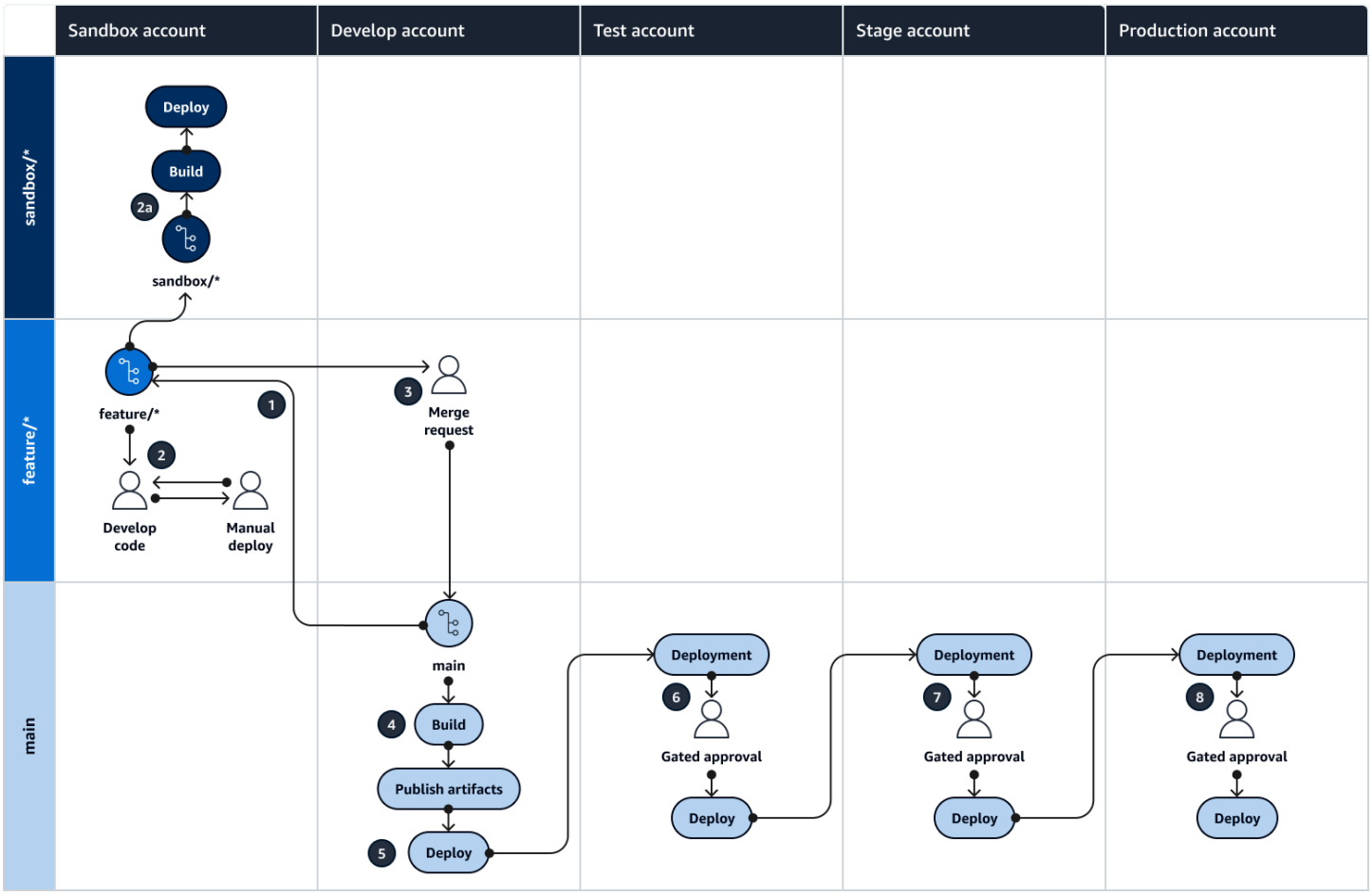
有关 Trunk 分支策略的更多信息，请参阅以下资源：

- 为多账户 DevOps 环境实施中继分支策略（AWS 规范性指导）
- Trunk-Based开发简介（基于主干的开发网站）

本节中的主题：

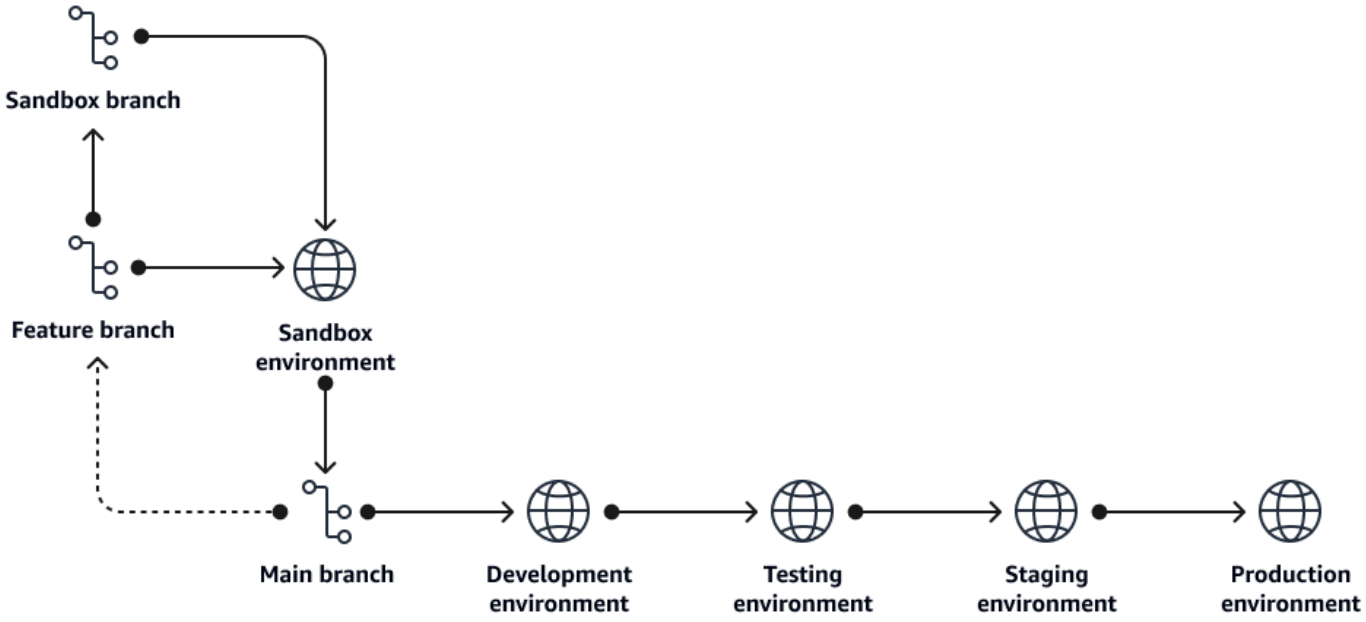
Trunk 策略的直观概述

下图可以像 Punnett 方块（维基百科）一样用来理解主干分支策略。将垂直轴上的分支与水平轴上的 AWS 环境对齐，以确定在每个场景中要执行的操作。带圆圈的数字将引导您完成图中表示的操作顺序。此图显示了 Trunk 分支策略的开发工作流程，从沙盒环境中的`feature`分支到分支的`main`生产版本。有关每种环境中发生的活动的更多信息，请参阅本指南中的DevOps 环境。



树干中的分支策略

中继分支策略通常具有以下分支。



功能分支

您可以在feature分支中开发功能或创建修补程序。要创建feature分支，您需要从该分支中main分支。开发人员在feature分支中迭代、提交和测试代码。某项功能完成后，开发者将对该功能进行推广。从feature分支向前只有两条路径：

- 合并到分sandbox支中
- 向main分支创建合并请求

命名惯例：	`feature/<story number>_<developer initials>_<descriptor>`
命名约定示例：	`feature/123456_MS_Implement_Feature_A`

沙盒分支

这个分支是一个非标准的主干分支，但它对 CI/CD管道开发很有用。该sandbox分支主要用于以下目的：

- 使用管道向沙盒环境执行全面部署 CI/CD
- 在提交合并请求以在较低的环境（例如开发或测试）中进行全面测试之前，先开发和测试管道。

Sandbox分支本质上是临时性的，其寿命是短暂的。应在特定测试完成后将其删除。

命名惯例：	sandbox/<story number>_<developer initials>_<descriptor>
命名约定示例：	sandbox/123456_MS_Test_Pipeline_Deploy

主分支

分main支始终代表在生产环境中运行的代码。代码从中分支main、开发，然后合并回到。main来自的部署main可以针对任何环境。要防止删除，请为分支启用main分支保护。

命名惯例：	main

修补程序分支

基于主干的工作流程中没有专门的hotfix分支。修补程序使用feature分支。

中继策略的优缺点

Trunk 分支策略非常适合规模较小、成熟、具有较强沟通能力的开发团队。如果您为应用程序发布了连续、滚动的功能，它也能很好地运行。如果你有庞大或分散的开发团队，或者你有大量的预定功能发布，那么它就不太适合了。在此模型中会发生合并冲突，因此请注意，解决合并冲突是一项关键技能。所有团队成员都必须接受相应的培训。

优点

Trunk-based 开发提供了多种优势，可以改善开发流程、简化协作并提高软件的整体质量。以下是一些主要好处：

- 更快的反馈循环 — 通过基于主干的开发，开发人员可以频繁地整合代码更改，通常每天多次。与基于功能的开发模型相比，这可以更快地提供有关潜在问题的反馈，并帮助开发人员更快地发现和修复问题。
- 减少合并冲突 — 在基于主干的开发中，由于更改是持续集成的，因此可以最大限度地降低大型、复杂合并冲突的风险。这有助于维护更简洁的代码库，并减少解决冲突所花费的时间。在基于功能的开发中，解决冲突既耗时又容易出错。

- 改善协作 — Trunk-based 开发鼓励开发人员在同一个分支上协作，促进团队内部更好的沟通和协作。这可以加快解决问题的速度和更具凝聚力的团队活力。
- 更轻松的代码审查 — 由于在基于主干的开发中，代码更改更小、更频繁，因此可以更轻松地进行全面的代码审查。较小的更改通常更容易理解和审查，从而更有效地识别潜在问题并进行改进。
- 持续集成和交付 — Trunk-based 开发支持持续集成和持续交付的原则 (CI/CD)。通过将代码库保持在可发布状态并经常集成更改，团队可以更轻松地采用 CI/CD 实践，从而缩短部署周期并提高软件质量。
- 提高代码质量 — 通过频繁的集成、测试和代码审查，基于主干的开发可以提高整体代码质量。开发人员可以更快地发现和修复问题，从而降低技术债务随着时间的推移而积累的可能性。
- 简化的分支策略 — Trunk-based 开发通过减少长寿命分支的数量来简化分支策略。这样可以更轻松地管理和维护代码库，特别是对于大型项目或团队而言。

缺点

Trunk-based 开发确实有一些缺点，这可能会影响开发过程和团队动态。以下是一些明显的缺点：

- 有限隔离 — 由于所有开发人员都在同一个分支上工作，因此团队中的每个人都可以立即看到他们的更改。这可能会导致干扰或冲突，导致意想不到的副作用或破坏构建。相比之下，基于功能的开发可以更好地隔离更改，以便开发人员可以更加独立地工作。
- 测试压力增加 — Trunk-based 开发依赖于持续集成和自动测试来快速发现问题。但是，这种方法可能会给测试基础架构带来很大的压力，并且需要维护良好的测试套件。如果测试不全面或不可靠，则可能导致主分支中出现未被发现的问题。
- 减少对发布的控制 —— Trunk-based 开发旨在使代码库保持持续可发布状态。虽然这可能具有优势，但它可能并不总是适合发布时间表严格的项目或需要同时发布特定功能的项目。 Feature-based 开发提供了对发布功能的时间和方式的更多控制。
- 代码流失 — 随着开发人员不断将更改集成到主分支中，基于主干的开发可能会导致代码流失增加。这可能会使开发人员难以跟踪代码库的当前状态，并且在尝试了解最近更改的效果时可能会引起混乱。
- 需要强大的团队文化 —— Trunk-based 发展需要团队成员之间高度的纪律、沟通和协作。这可能很难维护，尤其是在规模较大的团队中，或者与使用这种方法经验较少的开发人员合作时。
- 可扩展性挑战 — 随着开发团队规模的扩大，集成到主分支中的代码更改数量可能会迅速增加。这可能会导致更频繁的构建中断和测试失败，从而使代码库难以保持可发布状态。

GitHub 流量分支策略

GitHub Flow 是一个基于分支的轻量级工作流程，由开发。 GitHub GitHubFlow 基于短期功能分支的概念，当功能完成并准备部署时，这些分支会合并到主分支中。 GitHub Flow 的关键原理是：

- 分支是轻量级的 — 开发人员只需单击几下即可为其工作创建功能分支，从而在不影响主分支的情况下提高协作和实验能力。
- 持续部署 — 更改在合并到主分支后立即进行部署，这样可以快速进行反馈和迭代。
- 合并请求 — 开发人员使用合并请求来启动讨论和审查流程，然后再将其更改合并到主分支。

有关 GitHub Flow 的更多信息，请参阅以下资源：

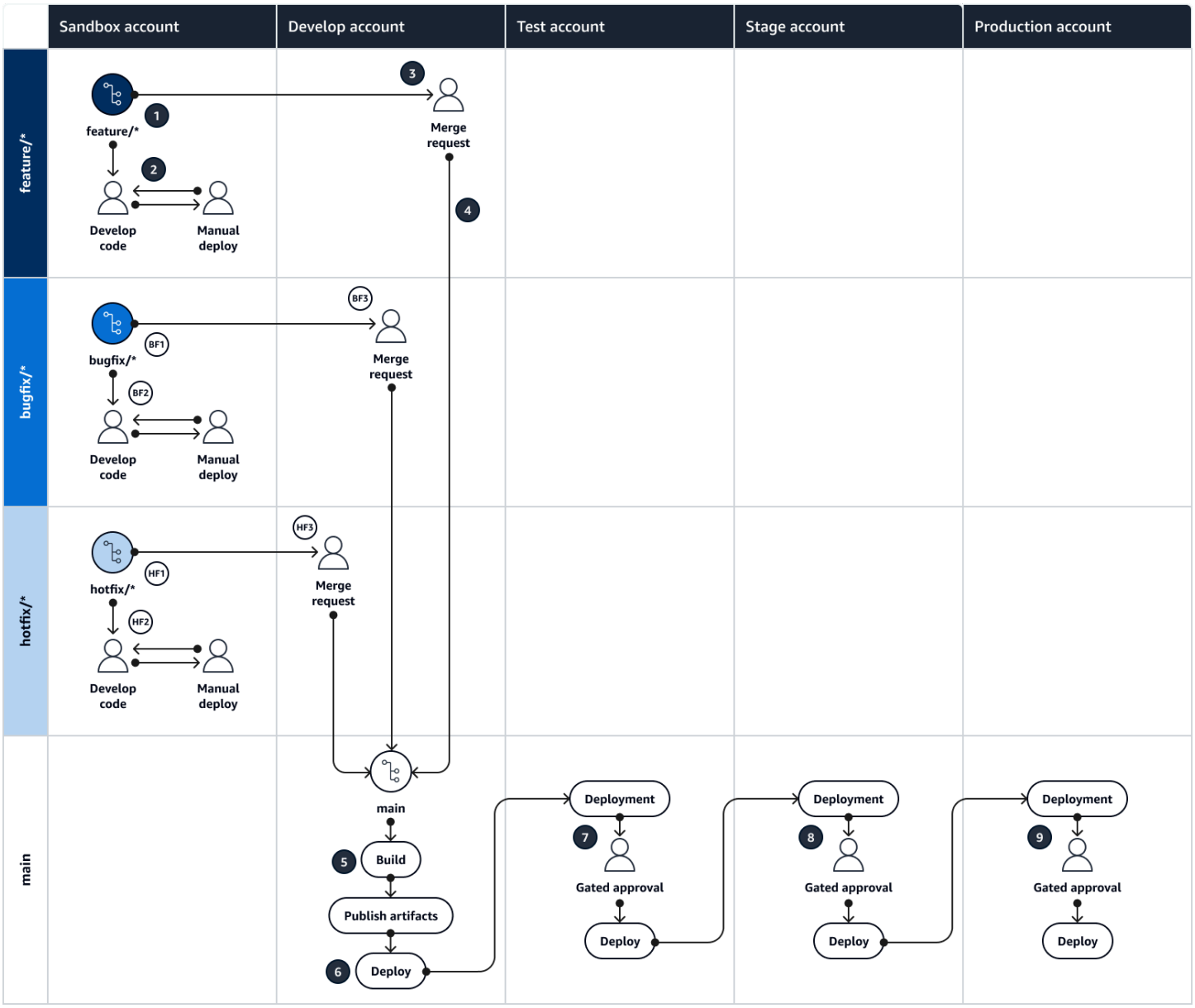
- 为多账户 DevOps 环境实施 GitHub Flow 分支策略（AWS 规范性指导）
- GitHub Flow 快速入门（GitHub 文档）
- 为什么 GitHub Flow？（GitHubFlow 网站）

本节中的主题：

- GitHub Flow 策略的可视化概述
- GitHub Flow 策略中的分支
- GitHub Flow 策略的优缺点

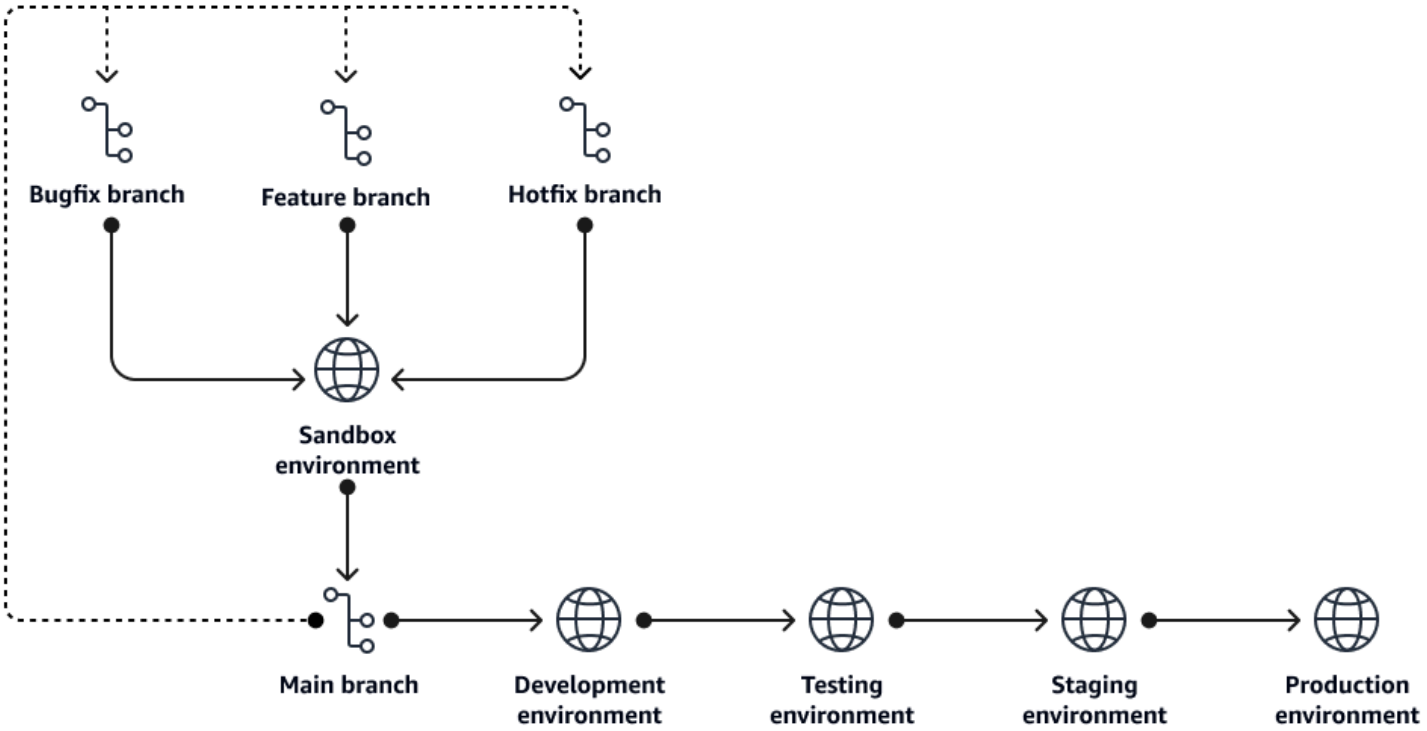
GitHub Flow 策略的可视化概述

下图可以像 Punnett 方块一样用来理解 GitHub Flow 分支策略。将垂直轴上的分支与水平轴上的 AWS 环境对齐，以确定在每个场景中要执行的操作。带圆圈的数字将引导您完成图中表示的操作顺序。此图显示了 GitHub Flow 分支策略的开发工作流程，从沙盒环境中的功能分支到主分支的生产版本。有关每种环境中发生的活动的更多信息，请参阅本指南中的DevOps 环境。



GitHub Flow 策略中的分支

GitHub Flow 分支策略通常具有以下分支。



功能分支

你在feature分支中开发功能。要创建feature分支，您需要从该分支中main分支。开发人员在feature分支中迭代、提交和测试代码。功能完成后，开发者通过向创建合并请求来推广该功能main。

命名惯例：	feature/<story number>_<developer initials>_<descriptor>
命名约定示例：	feature/123456_MS_Implement_Feature_A

错误修复分支

该bugfix分支用于修复问题。这些分支是从分支上分main支的。在沙盒或任何较低的环境中测试错误修复后，可以通过合并请求将其合并到更高的环境中，从而将其提升到main更高的环境。这是组织和跟踪的建议命名惯例，也可以使用功能分支来管理此过程。

命名惯例：	bugfix/<ticket number>_<developer initials>_<descriptor>
命名约定示例：	bugfix/123456_MS_Fix_Problem_A

修补程序分支

该hotfix分支用于解决影响大的关键问题，同时最大限度地减少开发人员与部署在生产环境中的代码之间的延迟。这些分支是从分支上分main支的。在沙盒或任何较低的环境中测试此修补程序后，可以通过合并请求将其合并到更高的环境中，将其升级到main更高的环境。这是组织和跟踪的建议命名惯例，也可以使用功能分支来管理此过程。

命名惯例：	hotfix/<ticket number>_<developer initials>_<descriptor>
命名约定示例：	hotfix/123456_MS_Fix_Problem_A

主分支

分main支始终代表在生产环境中运行的代码。使用合并请求将代码从main分feature支合并到分支中。为了防止删除并防止开发人员将代码直接推送到main，请为分支启用main分支保护。

命名惯例：	main

GitHub Flow 策略的优缺点

Github Flow 分支策略非常适合具有较强沟通能力的小型、成熟的开发团队。这种策略非常适合想要实现持续交付的团队，并且得到了通用 CI/CD引擎的大力支持。 GitHub Flow 是轻量级的，没有太多的规则，并且能够支持快速移动的团队。如果您的团队有严格的合规或发布流程需要遵守，则它不太合适。合并冲突在此模型中很常见，而且可能经常发生。解决合并冲突是一项关键技能，你必须相应地训练所有团队成员。

优点

GitHub Flow 提供了多种优势，可以改善开发流程、简化协作并提高软件的整体质量。以下是一些主要好处：

- 灵活轻便 — GitHub Flow 是一种轻巧灵活的工作流程，可帮助开发人员在软件开发项目上进行协作。它允许以最小的复杂性进行快速迭代和实验。
- 简化协作 — GitHub Flow 为管理功能开发提供了清晰而简化的流程。它鼓励可以快速审查和合并的小型、有针对性的更改，从而提高效率。
- 清晰的版本控制 — 使用 GitHub Flow，每项更改都是在单独的分支中进行的。这样可以建立清晰且可追溯的版本控制历史记录。这可以帮助开发人员跟踪和了解更改，在必要时进行恢复，并维护可靠的代码库。
- 无缝持续集成 — GitHub Flow 与持续集成工具集成。创建拉取请求可以启动自动测试和部署流程。CI 工具可帮助您在更改合并到`main`分支之前对其进行全面测试，从而降低在代码库中引入错误的风险。
- 快速反馈和持续改进 — GitHub Flow 通过拉取请求促进频繁的代码审查和讨论，从而鼓励快速反馈循环。这有助于及早发现问题，促进团队成员之间的知识共享，最终提高代码质量并改善开发团队内部的协作。
- 简化了回滚和还原 — 如果代码更改引入了意想不到的错误或问题， GitHub Flow 可以简化回滚或还原更改的过程。通过清晰的提交和分支历史记录，可以更轻松地识别和还原有问题的更改，从而有助于维护稳定且功能齐全的代码库。
- 轻量级学习曲线 — GitHub Flow 比 Gitflow 更容易学习和采用，特别是对于已经熟悉 Git 和版本控制概念的团队而言。它的简单性和直观的分支模型使不同经验水平的开发人员都可以使用，从而缩短了与采用新开发工作流程相关的学习曲线。
- 持续开发 — GitHub Flow 使团队能够采用持续部署方法，因为每项更改都可以在合并到分`main`支后立即部署。这种简化的流程消除了不必要的延迟，并确保用户可以快速获得最新的更新和改进。这使得开发周期更加敏捷，响应速度更快。

缺点

虽然 GitHub Flow 有几个优点，但也必须考虑其潜在的缺点：

- 对大型项目的适用性有限 — GitHub Flow 可能不太适合具有复杂代码库和多个长期功能分支的大型项目。在这种情况下，结构化程度更高的工作流程（例如 Gitflow）可能会更好地控制并发开发和发布管理。
- 缺乏正式的发布结构 — GitHub Flow 没有明确定义发布流程或支持功能，例如版本控制、修补程序或维护分支。对于需要严格发布管理或需要长期支持和维护的项目来说，这可能是一个限制。
- 对长期发布计划的支持有限 — GitHub Flow 专注于短期功能分支，这些分支可能与需要长期发布计划的项目（例如具有严格路线图或广泛功能依赖关系的项目）不太一致。在 Flo GitHub w 的限制下，管理复杂的发布时间表可能具有挑战性。

- 可能出现频繁的合并冲突 — 由于 GitHub Flow 鼓励频繁进行分支和合并，因此有可能遇到合并冲突，尤其是在有大量开发活动的项目中。解决这些冲突可能很耗时，可能需要开发团队付出额外的努力。
- 缺乏正式的工作流程阶段 — GitHub Flow 未定义明确的开发阶段，例如 alpha、beta 或候选发布阶段。这可能会使传达项目的当前状态或不同分支或版本的稳定性级别变得更加困难。
- 重大更改的影响 — 由于 GitHub Flow 鼓励经常将更改合并到`main`分支中，因此引入影响代码库稳定性的重大更改的风险更高。严格的代码审查和测试实践对于有效降低这种风险至关重要。

Gitflow 分支策略

Gitflow 是一种分支模型，它涉及使用多个分支将代码从开发转移到生产。Gitflow 非常适合那些有计划发布周期并且需要将一系列功能定义为发布的团队。开发是在各个功能分支中完成的，这些分支经批准后合并到用于集成的开发分支中。该分支中的功能被认为已准备就绪，可以投入生产。当所有计划中的功能都积累到开发分支中后，将创建一个发布分支，用于部署到上层环境。这种分离可以更好地控制哪些更改将按定义的时间表移动到哪个命名环境。如有必要，您可以将此过程加快到更快的部署模式。

有关 Gitflow 分支策略的更多信息，请参阅以下资源：

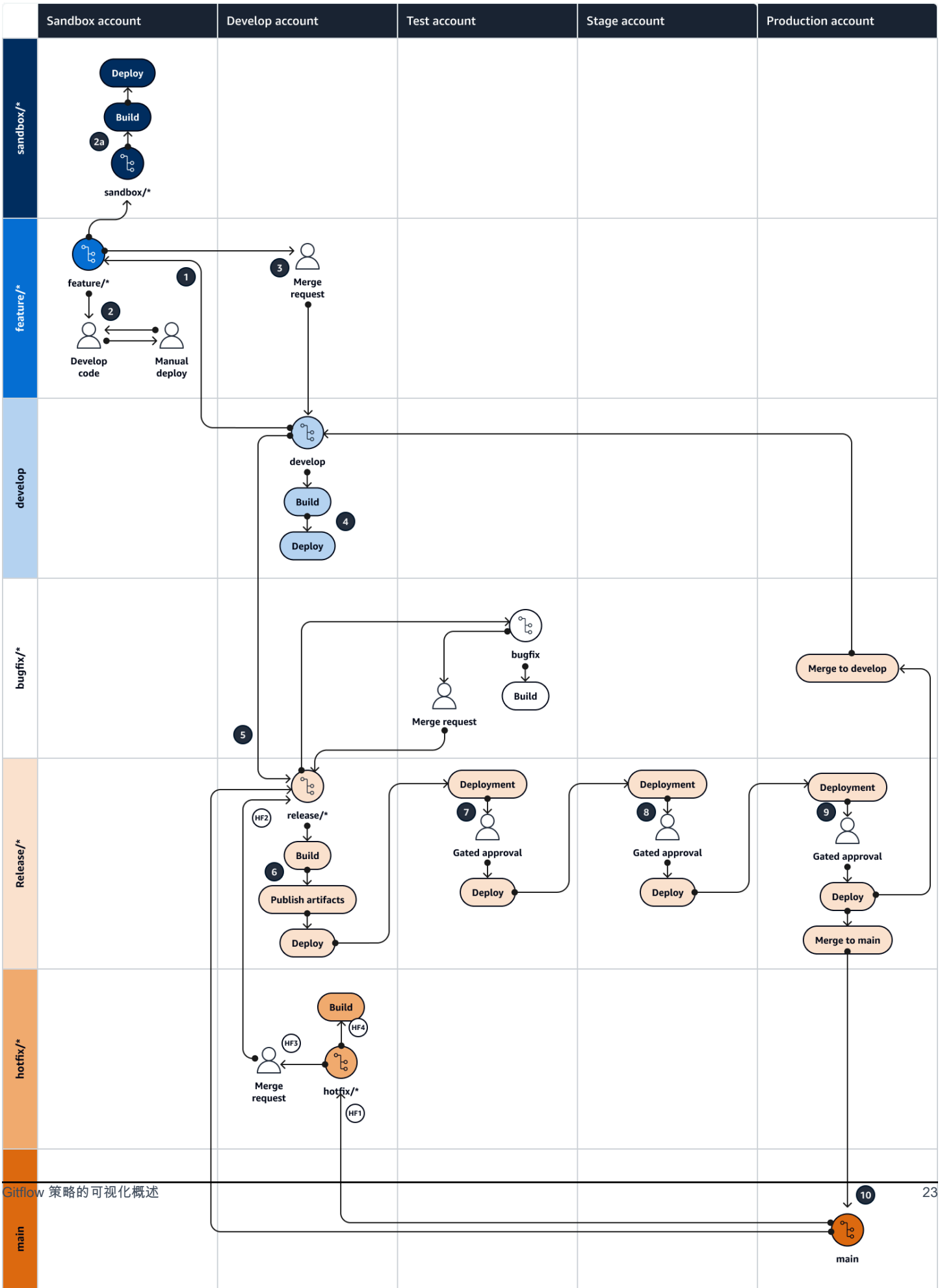
- 为多账户 DevOps 环境实施 Gitflow 分支策略（规范性指南）AWS
- 原始 Gitflow 博客（Vincent Driessen 博客文章）
- Gitflow 工作流（Atlassian）

本节中的主题：

- Gitflow 策略的可视化概述
- Gitflow 策略中的分支
- Gitflow 策略的优缺点

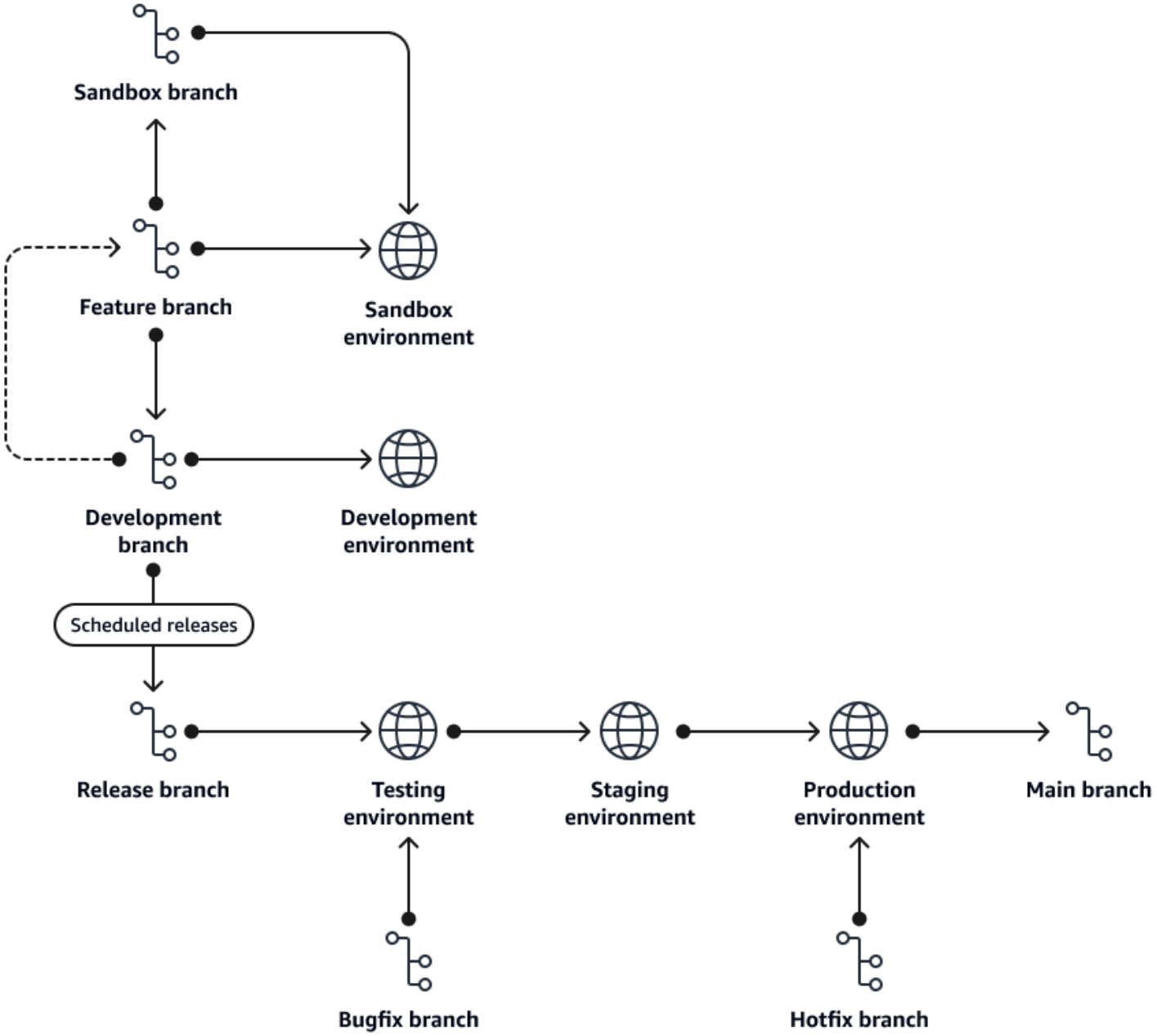
Gitflow 策略的可视化概述

下图可以像 Punnett 方块一样用来理解 Gitflow 的分支策略。将垂直轴上的分支与水平轴上的 AWS 环境对齐，以确定在每个场景中要执行的操作。带圆圈的数字将引导您完成图中表示的操作顺序。有关每种环境中发生的活动的更多信息，请参阅本指南中的DevOps 环境。


Sandbox account
Develop account
Test account
Stage account
Production account
sandbox/*
feature/*
develop
bugfix/*
Release/*
hotfix/*
main
Deploy
Build
2a
sandbox/*
feature/*
1
2
Develop code
Manual deploy
3
Merge request
develop
Build
4
Deploy
bugfix
Build
Merge request
5
Merge to develop
HF2
release/*
Build
6
Publish artifacts
Deploy
Deployment
7
Gated approval
Deploy
Deployment
8
Gated approval
Deploy
Deployment
9
Gated approval
Deploy
Merge to main
Build
HF4
HF3
Merge request
hotfix/*
HF1
10
main

Gitflow 策略中的分支

Gitflow 分支策略通常具有以下分支。



功能分支

Feature分支是你开发功能的短期分支。分feature支是通过从分支中分develop支来创建的。开发人员在feature分支中迭代、提交和测试代码。该功能完成后，开发者将对该功能进行推广。从功能分支出发，只有两条前进的道路：

- 合并到分sandbox支中
- 向develop分支创建合并请求

命名惯例：	feature/<story number>_<developer initials>_<descriptor>
命名约定示例：	feature/123456_MS_Implement_Feature_A

沙盒分支

该sandbox分支是 Gitflow 的非标准短期分支。但是，它对于 CI/CD 管道开发很有用。该sandbox分支主要用于以下目的：

- 使用 CI/CD 管道而不是手动部署，向沙盒环境执行全面部署。
- 在提交合并请求以在较低的环境（例如开发或测试）中进行全面测试之前，先开发和测试管道。

Sandbox分支本质上是暂时的，并不意味着寿命长。应在特定测试完成后将其删除。

命名惯例：	sandbox/<story number>_<developer initials>_<descriptor>
命名约定示例：	sandbox/123456_MS_Test_Pipeline_Deploy

开发分支

该develop分支是一个长期存在的分支，其中的功能被集成、构建、验证并部署到开发环境中。所有feature分支都合并到该develop分支中。合并到develop分支是通过合并请求完成的，合并请求需要成功构建并获得两位开发人员的批准。为防止删除，请在分支上启用develop分支保护。

命名惯例：	develop

发布分支

在 Gitflow 中，release分支是短期分支。这些分支之所以特别，是因为你可以将它们部署到多个环境中，采用一次构建、多次部署的方法。 Release分支可以针对测试、暂存或生产环境。在开发团队决定将功能提升到更高的环境后，他们会创建一个新release分支，并使用与先前版本相比的增量版本号。在每个环境的门口，部署都需要手动批准才能继续。 Release分支应该要求更改合并请求。

在release分支部署到生产环境后，应将其合并回develop和main分支，以确保将任何错误修复或修补程序合并回未来的开发工作中。

命名惯例：	release/v{major}.{minor}
命名约定示例：	release/v1.0

主分支

该main分支是一个长期存在的分支，它始终代表生产中运行的代码。从发布管道成功部署后，代码将自动从发布分支合并到分支中。main为防止删除，请在分支上启用main分支保护。

命名惯例：	main

错误修复分支

该bugfix分支是一个短期分支，用于修复尚未发布到生产环境的发布分支中的问题。分bugfix支只能用于将分release支中的修复提升到测试、暂存或生产环境。分bugfix支总是从分支上分release支。

测试错误修正后，可以通过合并请求将其提升到release分支。然后，您可以按照标准发布流程推动release分支向前发展。

命名惯例：	bugfix/<ticket>_<developer initials>_<descriptor>
命名约定示例：	bugfix/123456_MS_Fix_Problem_A

修补程序分支

该hotfix分支是一个短期分支，用于修复生产中的问题。它仅用于推广必须加快才能进入生产环境的修复程序。分hotfix支总是从中分支。main

测试完此修补程序后，您可以通过向从main中创建的release分支发出合并请求，将其提升到生产环境。为了进行测试，您可以按照标准发布流程将release分支向前推进。

命名惯例：	hotfix/<ticket>_<developer initials>_<descriptor>
命名约定示例：	hotfix/123456_MS_Fix_Problem_A

Gitflow 策略的优缺点

Gitflow 分支策略非常适合规模更大、分布更分散、有严格发布和合规要求的团队。Gitflow为组织提供了可预测的发布周期，而大型组织通常更喜欢这样做。Gitflow 也非常适合需要护栏才能正确完成软件开发生命周期的团队。这是因为该战略中有多种机会进行审查和质量保证。Gitflow 也非常适合必须同时维护多个生产版本的团队。GitFlow 的一些缺点是，它比其他分支模型更复杂，需要严格遵守模式才能成功完成。由于管理发布分支的严格性质，Gitflow不适合追求持续交付的组织。Gitflow发行分支机构可能是长期存在的分支机构，如果不及时解决，可能会积累技术债务。

优点

Gitflow-based 开发提供了多种优势，可以改善开发流程、简化协作并提高软件的整体质量。以下是一些主要好处：

- 可预测的发布流程 — Gitflow 遵循定期且可预测的发布流程。它非常适合有规律开发和发布节奏的团队。
- 改善协作 — Gitflow 鼓励使用feature和分支。release这两个分支可以帮助团队并行工作，同时最大限度地减少相互依赖。
- 非常适合多种环境 — Gitflow 使用release分支，分支可以是寿命更长的分支。这些分支使团队能够在更长的时间内定位单个版本。
- 生产中的多个版本 — 如果您的团队支持生产中的软件的多个版本，则 Gitflow release 分支支持此要求。
- Built-in 代码质量审查 — Gitflow 要求并鼓励在将代码提升到其他环境之前使用代码审查和批准。此过程要求所有代码促销都必须执行此步骤，从而消除了开发者之间的摩擦。

- 组织优势 — Gitflow在组织层面也有优势。Gitflow 鼓励使用标准发布周期，这有助于组织了解和预测发布时间表。由于企业现在知道何时可以交付新功能，因此由于有固定的交付日期，因此减少了时间表上的摩擦。

缺点

Gitflow-based 开发确实有一些缺点，可能会影响开发过程和团队动态。以下是一些明显的缺点：

- 复杂性 — Gitflow 是新团队需要学习的复杂模式，你必须遵守 Gitflow 的规则才能成功使用它。
- 持续部署 — Gitflow 不适合将许多部署快速发布到生产环境的模式。这是因为 Gitflow 需要使用多个分支和严格的工作流程来管理分支。`release`
- 分支管理 — Gitflow 使用许多分支，维护起来可能会变得繁重。为了使分支彼此正确对齐，跟踪各个分支并合并已发布的代码可能很困难。
- 技术债务 — 由于 Gitflow 的发布速度通常比其他分支模型慢，因此可以积累更多的功能来发布，这可能会导致技术债务积累。

团队在决定 Gitflow-based 开发是否是适合其项目的方法时，应仔细考虑这些缺点。

后续步骤

本指南解释了三种常见的 Git 分支策略之间的区别： GitHubFlow、Gitflow 和 Trunk。它详细描述了他们的工作流程，还提供了每种工作流程的优缺点。接下来的步骤是为您的组织选择其中一个标准工作流程。要实现其中一种分支策略，请参阅以下内容：

- 为多 DevOps 账户环境实施中继分支策略
- 为多 DevOps 账户环境实施 GitHub Flow 分支策略
- 为多账户环境实施 Gitflow 分支策略 DevOps

如果您不确定从哪里开始团队使用 Git 和 DevOps 流程，我们建议您选择一个标准解决方案并对其进行测试。使用标准的分支约定可以帮助团队在现有文档的基础上再接再厉，了解最适合他们的方法。

如果你的策略对你的组织或开发团队不起作用，不要害怕改变策略。开发团队的需求和要求会随着时间的推移而发生变化，因此没有单一的完美解决方案。

资源

本指南不包括针对 Git 的培训；但是，如果您需要此培训，互联网上有许多高质量的资源可供选择。我们建议您从 Git 文档网站开始。

以下资源可以帮助您完成 Git 分支之 AWS 云旅。

AWS 规范性指导

- 为多 DevOps 账户环境实施中继分支策略
- 为多 DevOps 账户环境实施 GitHub Flow 分支策略
- 为多账户环境实施 Gitflow 分支策略 DevOps

其他 AWS 指导

- AWS DevOps 指导
- AWS 部署管道参考架构
- 什么是 DevOps ？
- DevOps资源

其他资源

- 十二因子应用程序方法论 (12factor.net)
- Git-Sec rets () GitHub
- Gitflow
  - 最初的 Gitflow 博客（Vincent Driessen 博客文章）
  - Gitflow 工作流程 (Atlassian)
  - Gitflow 开启 GitHub：如何将 Git Flow 工作流程与 GitHub 基于 Repos 的存储库一起使用（视频）YouTube
  - Git Flow 初始化示例（YouTube 视频）
  - Gitflow 发布分支从头到尾（视频）YouTube
- GitHub 流量

  - GitHub Flow 快速入门（GitHub 文档）
  - 为什么 GitHub Flow？（GitHub Flow 网站）
- 后备箱
  - 基于@@ 主干的开发简介（基于主干的开发网站）

贡献者

编写

- 迈克·斯蒂芬斯，高级云应用程序架构师， AWS
- Rayjan Wilson，高级云应用程序架构师， AWS
- Abhilash Vinod，团队负责人，高级云应用程序架构师， AWS

正在审阅

- 斯蒂芬 DiCato，高级安全顾问， AWS
- Gaurav Samudra，云应用程序架构师， AWS
- Steven Guggenheimer，团队负责人，高级云应用程序架构师， AWS

技术写作

- Lilly AbouHarb，高级技术撰稿人， AWS

文档历史记录

下表介绍了本指南的一些重要更改。如果您希望收到有关未来更新的通知，可以订阅 RSS 源。

变更	说明	日期
更新	我们替换了 “主干中的分支” 策略、“ GitHub 流中的分支” 策略和 “Gitflow 策略中的分支” 部分中的图像。	2026年6月5日
初次发布	—	2024 年 2 月 15 日

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。