為多帳戶 DevOps 環境選擇 Git 分支策略

AWS 方案指引

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS 方案指引: 為多帳戶 DevOps 環境選擇 Git 分支策略

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標均為其各自擁有者的財產，這些擁有者可能附屬於 Amazon，或與 Amazon 有合作關係，亦或受到 Amazon 贊助。

Table of Contents

簡介 1
目標 1
使用 CI/CD 實務 2
了解 DevOps 環境 3
沙盒環境 3
存取 4
建置步驟 4
部署步驟 4
移至開發環境之前的期望 4
開發環境 5
存取 4
建置步驟 4
部署步驟 4
移至測試環境之前的期望 6
測試環境 6
存取 4
建置步驟 4
部署步驟 4
移至預備環境之前的期望 7
預備環境 7
存取 4
建置步驟 4
部署步驟 4
移至生產環境之前的期望 8
生產環境 8
存取 4
建置步驟 4
部署步驟 4
Git 型開發的最佳實務 10
Git 分支策略 11
中繼線分支策略 11
中繼線策略的視覺化概觀 12
中繼線策略中的分支 13
Trunk 策略的優點和缺點 15

GitHub 流程分支策略 17
GitHub 流程策略的視覺化概觀 17
GitHub 流程策略中的分支 18
GitHub Flow 策略的優點和缺點 20
Gitflow 分支策略 22
Gitflow 策略的視覺化概觀 22
Gitflow 策略中的分支 24
Gitflow 策略的優點和缺點 27
後續步驟 29
資源 30
AWS 規定指引 30
其他 AWS 指引 30
其他資源 30
貢獻者 32
撰寫 32
檢閱 32
技術寫作 32
文件歷史紀錄 33
..... xxxiv

為多帳戶 DevOps 環境選擇 Git 分支策略

Amazon Web Services（貢獻者）

2024 年 2 月 (文件歷史記錄)

轉向雲端型方法並在 上交付軟體解決方案 AWS 可以是變革性的。這可能需要變更您的軟體開發生命週期程序。一般而言， AWS 帳戶 在 中的開發過程中會使用多個 AWS 雲端。選擇相容的 Git 分支策略來與 DevOps 程序配對對於成功至關重要。為您的組織選擇正確的 Git 分支策略，可協助您在開發團隊之間簡潔地傳達 DevOps 標準和最佳實務。Git 分支在單一環境中可能很簡單，但在跨多個環境套用時可能會變得令人混淆，例如沙盒、開發、測試、預備和生產環境。擁有多個環境會增加 DevOps 實作的複雜性。

本指南提供 Git 分支策略的視覺化圖表，顯示組織如何實作多帳戶 DevOps 程序。視覺化指南可協助團隊了解如何將 Git 分支策略與 DevOps 實務合併。使用 Gitflow、GitHub Flow 或 Trunk 等標準分支模型來管理原始程式碼儲存庫，有助於開發團隊調整其工作。這些團隊也可以在網際網路上使用標準 Git 訓練資源來了解和實作這些模型和策略。

如需 DevOps 最佳實務 AWS，請參閱 AWS Well-Architected 中的 DevOps 指南。當您檢閱本指南時，請使用盡職調查來為您的組織選擇正確的分支策略。有些策略可能比其他策略更符合您的使用案例。

目標

本指南是文件系列的一部分，旨在為具有多個 的組織選擇和實作 DevOps 分支策略 AWS 帳戶。此系列旨在協助您從一開始就套用最符合您的需求、目標和最佳實務的策略，以簡化您在 中的體驗 AWS 雲端。本指南不包含 DevOps 可執行指令碼，因為它們會根據組織使用的持續整合和持續交付 (CI/CD) 引擎和技術架構而有所不同。

本指南說明三種常見 Git 分支策略之間的差異：GitHub Flow、Gitflow 和 Trunk。本指南中的建議可協助團隊識別與其組織目標一致的分支策略。檢閱本指南後，您應該能夠為您的組織選擇分支策略。選擇策略後，您可以使用下列其中一種模式，協助您與開發團隊實作該策略：

- 實作多帳戶 DevOps 環境的主體分支策略
- 針對多帳戶 DevOps 環境實作 GitHub 流程分支策略
- 為多帳戶 DevOps 環境實作 Gitflow 分支策略

請務必注意，適用於某個組織、團隊或專案的項目可能不適合其他組織、團隊或專案。Git 分支策略之間的選擇取決於各種因素，例如團隊大小、專案需求，以及協作、整合頻率和發行管理之間的所需平衡。

使用 CI/CD 實務

AWS 建議您實作持續整合和持續交付 (CI/CD)，這是自動化軟體版本生命週期的程序。它會自動化傳統上從開發取得新程式碼到生產所需的許多或所有手動 DevOps 程序。CI/CD 管道包含沙盒、開發、測試、預備和生產環境。在每個環境中，CI/CD 管道會佈建部署或測試程式碼所需的任何基礎設施。透過使用 CI/CD，開發團隊可以對程式碼進行變更，然後自動測試和部署。CI/CD 管道也為開發團隊提供控管和防護機制。它們會強制執行一致性、標準、最佳實務，以及功能接受和部署的最低接受程度。如需詳細資訊，請參閱實作持續整合和持續交付 AWS。

本指南中討論的所有分支策略都非常適合 CI/CD 實務。CI/CD 管道的複雜性會隨著分支策略的複雜性而增加。例如，Gitflow 是本指南中討論的最複雜分支策略。此策略的 CI/CD 管道需要更多步驟（例如基於合規原因），而且必須支援多個同時的生產版本。隨著分支策略的複雜性增加，使用 CI/CD 也變得更加重要。這是因為 CI/CD 為開發團隊建立防護機制和機制，以防止開發人員有意或無意地規避定義的程序。

AWS 提供一套開發人員服務，旨在協助您建置 CI/CD 管道。例如， AWS CodePipeline 是一種全受管的持續交付服務，可協助您自動化發行管道，以取得快速且可靠的應用程式和基礎設施更新。 會AWS CodeBuild編譯原始程式碼、執行測試，並產生ready-to-deploy的軟體套件。如需詳細資訊，請參閱 上的開發人員工具 AWS。

了解 DevOps 環境

若要了解分支策略，您必須了解每個環境中發生的目的和活動。建立數個環境可協助您將開發活動分成不同階段、監控這些活動，並防止意外發行未經核准的功能。您可以在 AWS 帳戶 每個環境中有一或多個 。

大多數組織都有幾個概述使用的環境。不過，環境數量可能因組織和軟體開發政策而異。此文件系列假設您有下列五個通用環境，這些環境橫跨您的開發管道，但它們可能由不同的名稱呼叫：

- 沙盒 – 開發人員編寫程式碼、犯錯和執行概念驗證工作的環境。
- 開發 – 一種環境，開發人員會整合其程式碼，以確認其可做為單一且具凝聚力的應用程式運作。
- 測試 – 進行 QA 團隊或接受度測試的環境。團隊通常會在此環境中執行效能或整合測試。
- 預備 – 生產前環境，您可以在此環境驗證程式碼和基礎設施在生產同等情況下如預期般執行。此環境設定為盡可能類似於生產環境。
- 生產 – 處理來自最終使用者和客戶流量的環境。

本節詳細說明每個環境。它還描述了每個環境的建置步驟、部署步驟和結束條件，以便您可以繼續進行下一個操作。下圖依序顯示這些環境。

本節主題：

- 沙盒環境
- 開發環境
- 測試環境
- 預備環境
- 生產環境

沙盒環境

沙盒環境是開發人員編寫程式碼、犯錯和執行概念驗證工作的地方。您可以從本機工作站或透過本機工作站上的指令碼部署到沙盒環境。

存取

開發人員應擁有沙盒環境的完整存取權。

建置步驟

開發人員準備好將變更部署到沙盒環境時，會在其本機工作站上手動執行建置。

1. 使用 git-secrets (GitHub) 掃描敏感資訊
2. 填入原始程式碼
3. 如果適用，請建置和編譯原始程式碼
4. 執行單位測試
5. 執行程式碼涵蓋範圍分析
6. 執行靜態程式碼分析
7. 建置基礎設施即程式碼 (IaC)
8. 執行 IaC 安全分析
9. 擷取開放原始碼授權
10. 發佈建置成品

部署步驟

如果您使用的是 Gitflow 或主體模型，則當`feature`分支成功建置在沙盒環境中時，部署步驟會自動啟動。如果您使用的是 GitHub Flow 模型，則需手動執行下列部署步驟。以下是沙盒環境中的部署步驟：

1. 下載已發佈的成品
2. 執行資料庫版本控制
3. 執行 IaC 部署
4. 執行整合測試

移至開發環境之前的期望

- 在沙盒環境中成功建置`feature`分支
- 開發人員已手動部署並測試沙盒環境中的功能

開發環境

開發環境是開發人員將其程式碼整合在一起的地方，以確保它們都作為一個具凝聚力的應用程式運作。在 Gitflow 中，開發環境包含合併請求包含的最新功能，並準備好發佈。在 GitHub Flow 和 Trunk 策略中，開發環境被視為測試環境，而且程式碼基底可能不穩定且不適合部署到生產環境。

存取

根據最低權限原則指派許可。最低權限是授予執行任務所需的最低許可的安全最佳實務。開發人員對開發環境的存取應該比對沙盒環境的存取少。

建置步驟

對`develop`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow) 建立合併請求會自動啟動建置。

1. 使用 git-secrets (GitHub) 掃描敏感資訊
2. 填入原始程式碼
3. 如果適用，請建置和編譯原始程式碼
4. 執行單位測試
5. 執行程式碼涵蓋範圍分析
6. 執行靜態程式碼分析
7. 建置 IaC
8. 執行 IaC 安全分析
9. 擷取開放原始碼授權

部署步驟

如果您使用的是 Gitflow 模型，則在開發環境中成功建置`develop`分支時，部署步驟會自動啟動。如果您使用的是 GitHub Flow 模型或主體模型，則當針對`main`分支建立合併請求時，部署步驟會自動啟動。以下是開發環境中的部署步驟：

1. 從建置步驟下載發佈的成品
2. 執行資料庫版本控制
3. 執行 IaC 部署

4. 執行整合測試

移至測試環境之前的期望

- 在開發環境中成功建置和部署`develop`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow)
- 單位測試以 100% 通過
- 成功的 IaC 建置
- 已成功建立部署成品
- 開發人員已執行手動驗證，以確認功能如預期般運作

測試環境

品質保證 (QA) 人員使用測試環境來驗證功能。它們會在完成測試後核准變更。當他們核准時，分支會移至下一個環境：預備。在 Gitflow 中，此環境及其上其他環境只能從`release`分支進行部署。`release` 分支是以包含計劃功能的`develop`分支為基礎。

存取

根據最低權限原則指派許可。開發人員對測試環境的存取應少於他們對開發環境的存取。QA 人員需要足夠的許可來測試功能。

建置步驟

此環境中的建置程序僅適用於使用 Gitflow 策略時的錯誤修正。向`bugfix`分支建立合併請求會自動啟動建置。

1. 使用 git-secrets (GitHub) 掃描敏感資訊
2. 填入原始程式碼
3. 如果適用，請建置和編譯原始程式碼
4. 執行單位測試
5. 執行程式碼涵蓋範圍分析
6. 執行靜態程式碼分析
7. 建置 IaC
8. 執行 IaC 安全分析
9. 擷取開放原始碼授權

部署步驟

在開發環境中部署之後，在測試環境中自動啟動`release`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow) 的部署。以下是測試環境中的部署步驟：

1. 在測試環境中部署`release`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow)
2. 暫停以進行指定人員的手動核准
3. 下載已發佈的成品
4. 執行資料庫版本控制
5. 執行 IaC 部署
6. 執行整合測試
7. 執行效能測試
8. 品質保證核准

移至預備環境之前的期望

- 開發和 QA 團隊已執行足夠的測試，以滿足組織的需求。
- 開發團隊已透過`bugfix`分支解決任何發現的錯誤。

預備環境

預備環境設定為與生產環境相同。例如，資料設定的範圍和大小應與生產工作負載相似。使用預備環境來驗證程式碼和基礎設施是否如預期般運作。此環境也是商業使用案例的首選，例如預覽或客戶示範。

存取

根據最低權限原則指派許可。開發人員應該擁有與生產環境相同的預備環境存取權。

建置步驟

無。在預備環境中重複使用測試環境中使用的相同成品。

部署步驟

在測試環境中核准和部署之後，自動啟動預備環境中`release`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow) 的部署。以下是預備環境中的部署步驟：

1. 在預備環境中部署`release`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow)
2. 暫停以進行指定人員的手動核准
3. 下載已發佈的成品
4. 執行資料庫版本控制
5. 執行 IaC 部署
6. （選用）執行整合測試
7. （選用）執行負載測試
8. 從必要的開發、QA、產品或業務核准者取得核准

移至生產環境之前的期望

- 生產同等版本已成功部署到預備環境
- （選用）整合和負載測試成功

生產環境

生產環境支援發行的產品，由真實用戶端處理真實資料。這是透過最低權限指派存取權的受保護環境，且提升存取權只能在有限的時間內透過稽核的例外狀況程序進行。

存取

在生產環境中，開發人員應該在 AWS 管理主控台中擁有有限的唯讀存取權。例如，開發人員應該能夠存取day-to-day操作的日誌資料。部署之前，應由核准步驟封鎖所有生產版本。

建置步驟

無。在測試和預備環境中使用的相同成品會在生產環境中重複使用。

部署步驟

在預備環境中核准和部署之後，在生產環境中自動啟動`release`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow) 的部署。以下是生產環境中的部署步驟：

1. 在生產環境中部署`release`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow)
2. 暫停以進行指定人員的手動核准

3. 下載已發佈的成品
4. 執行資料庫版本控制
5. 執行 IaC 部署

Git 型開發的最佳實務

若要成功採用 Git 型開發，請務必遵循一組最佳實務，以促進協作、維護程式碼品質，並支援持續整合和持續交付 (CI/CD)。除了本指南中的最佳實務之外，請檢閱 AWS Well-Architected DevOps 指南。以下是 Git 型開發的一些關鍵最佳實務 AWS：

- 保持少量且頻繁的變更 – 鼓勵開發人員遞交小型的增量變更或功能。這可降低合併衝突的風險，並讓您更輕鬆地快速識別和修正問題。
- 使用功能切換 – 若要管理不完整或實驗性功能的發行，請使用功能切換或功能旗標。這可協助您隱藏、啟用或停用生產中的特定功能，而不會影響主要分支的穩定性。
- 維護強大的測試套件 – 全面且維護良好的測試套件對於及早偵測問題，並確認程式碼庫保持穩定至關重要。投資測試自動化，並優先修正任何失敗的測試。
- 接受持續整合 – 使用持續整合工具和實務，將程式碼變更自動建置、測試和整合到`develop`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow)。這可協助您及早發現問題，並簡化開發程序。
- 執行程式碼檢閱 – 鼓勵對程式碼進行對等檢閱，以維護品質、分享知識，並在潛在問題整合至`main`分支之前加以發現。使用提取請求或其他程式碼檢閱工具來促進此程序。
- 監控和修正中斷的建置 – 當建置中斷或測試失敗時，請盡快排定修正問題的優先順序。這可讓`develop`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow) 保持可釋放狀態，並將對其他開發人員的影響降至最低。
- 溝通和協作 – 促進團隊成員之間的開放溝通和協作。確定開發人員知道程式碼庫正在進行的工作和變更。
- 持續重構 – 定期重構程式碼庫，以改善其可維護性並減少技術負債。鼓勵開發人員將程式碼保持在比他們找到的更好的狀態。
- 將短期分支用於複雜的任務 – 對於更大或更複雜的任務，請使用短期分支 （也稱為任務分支) 來處理變更。不過，請務必讓分支生命週期縮短，通常少於一天。盡快將變更合併回`develop`分支 (Gitflow) 或`main`分支 (Trunk 或 GitHub Flow)。更小且更頻繁的合併和檢閱，讓團隊比一個大型合併請求更容易使用和處理。
- 培訓和支援 團隊 – 為剛接觸 Git 型開發或需要採用其最佳實務指引的開發人員提供培訓和支援。

Git 分支策略

本指南會詳細說明下列以 Git 為基礎的分支策略：

- 中繼線 – 中繼線型開發是一種軟體開發實務，其中所有開發人員在單一分支上運作，通常稱為 `trunk`或 `main`分支。此方法背後的概念是透過頻繁整合程式碼變更，並依賴自動化測試和持續整合，讓程式碼基底保持在持續可釋放的狀態。
- GitHub Flow – GitHub Flow 是由 GitHub 開發的輕量型分支型工作流程。它以短期`feature`分支的概念為基礎。當功能完成並準備好部署時，該功能會合併到`main`分支中。
- Gitflow – 使用 Gitflow 方法，在個別特徵分支中完成開發。核准後，您會將`feature`分支合併到通常名為 的整合分支`develop`。當`develop`分支中累積到足夠的功能時，就會建立一個`release`分支，將功能部署到上層環境。

每個分支策略都有優點和缺點。雖然它們都使用相同的環境，但並非所有環境都使用相同的分支或手動核准步驟。在本指南的本節中，詳細檢閱每個分支策略，以便您熟悉其細微差別，並評估其是否符合組織的使用案例。

本節主題：

- 中繼線分支策略
- GitHub 流程分支策略
- Gitflow 分支策略

中繼線分支策略

主體型開發是一種軟體開發實務，其中所有開發人員都在單一分支上工作，通常稱為 `trunk`或 `main`分支。此方法背後的概念是透過頻繁整合程式碼變更，並依賴自動化測試和持續整合，讓程式碼基底保持在持續可釋放的狀態。

在以幹線為基礎的開發中，開發人員每天多次將變更遞交到`main`分支，以小型增量更新為目標。這可啟用快速意見回饋迴圈、降低合併衝突的風險，並促進團隊成員之間的協作。該實務強調維護良好的測試套件的重要性，因為它依賴自動化測試來及早發現潛在問題，並確保程式碼庫保持穩定且可釋放。

主體型開發通常與特徵型開發（也稱為特徵分支或特徵驅動型開發) 形成對比，其中每個新特徵或錯誤修正都在自己的專用分支中開發，與主分支分開。幹線型開發和功能型開發之間的選擇取決於團隊大小、專案需求，以及協作、整合頻率和發行管理之間的所需平衡等因素。

如需中繼線分支策略的詳細資訊，請參閱下列資源：

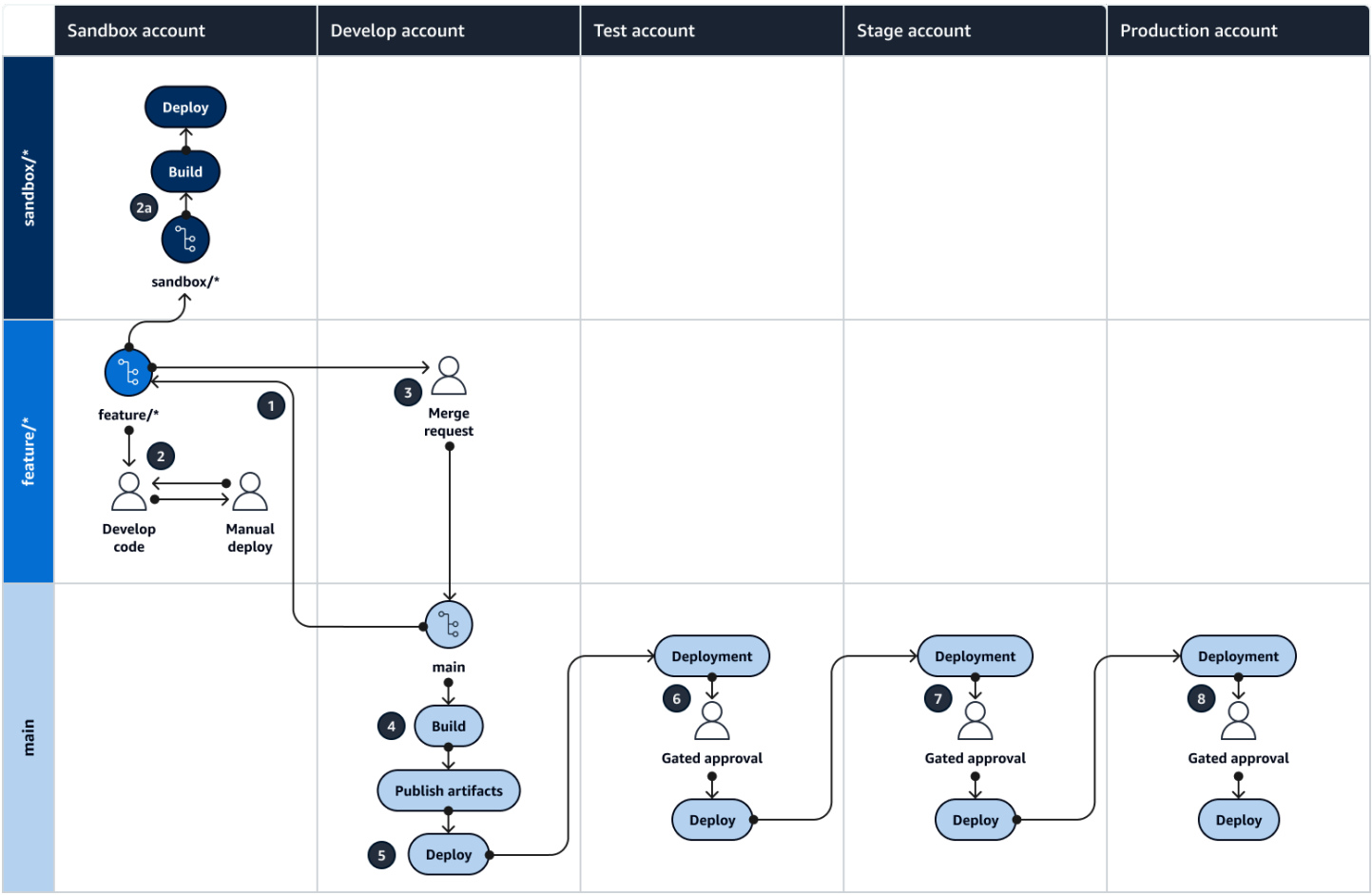
- 實作多帳戶 DevOps 環境的主體分支策略 (AWS 方案指引)
- 以主體為基礎的開發簡介 （以主體為基礎的開發網站）

本節主題：

- 中繼線策略的視覺化概觀
- 中繼線策略中的分支
- Trunk 策略的優點和缺點

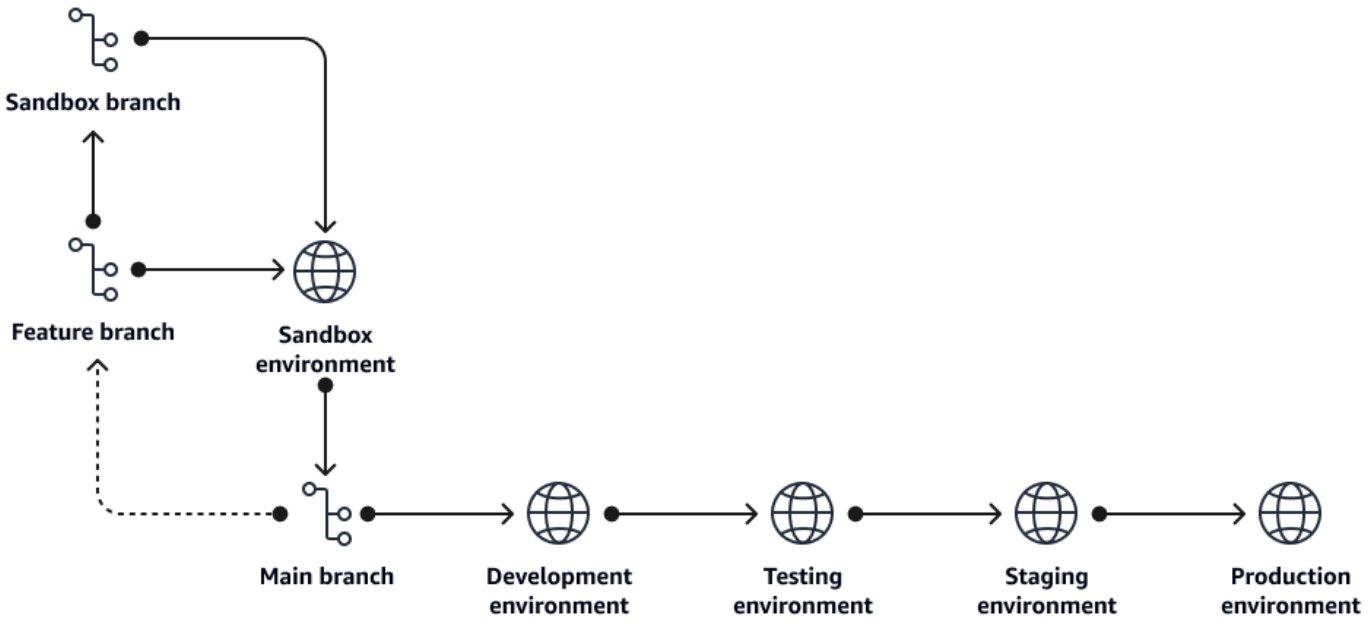
中繼線策略的視覺化概觀

下圖可以像 Punnett 方形 （維基百科） 一樣使用，以了解主體分支策略。將垂直軸上的分支與水平軸上的 AWS 環境對齊，以決定在每個案例中要執行的動作。圓圈數字會引導您完成圖表中所呈現的動作順序。此圖表顯示主體分支策略的開發工作流程，從沙盒環境中的`feature`分支到`main`分支的生產版本。如需每個環境中發生之活動的詳細資訊，請參閱本指南中的 DevOps 環境。



中繼線策略中的分支

中繼線分支策略通常具有下列分支。



功能分支

您可以在feature分支中開發功能或建立 Hotfix。若要建立feature分支，您可以從main分支中分支。開發人員在feature分支中反覆執行、遞交和測試程式碼。當功能完成時，開發人員會提升該功能。只有兩個路徑從feature分支轉送：

- 合併至sandbox分支
- 在main分支中建立合併請求

命名慣例：	feature/<story number>_<developer initials>_<descriptor>
命名慣例範例：	feature/123456_MS_Implement_Feature_A

沙盒分支

此分支是非標準幹線分支，但適用於 CI/CD 管道開發。sandbox 分支主要用於下列目的：

- 使用 CI/CD 管道執行沙盒環境的完整部署
- 在較低環境中提交合併請求進行完整測試之前，開發和測試管道，例如開發或測試。

Sandbox 分支本質上是暫時的，旨在短期運作。在特定測試完成後，應將其刪除。

命名慣例：	sandbox/<story number>_<developer initials>_<descriptor>
命名慣例範例：	sandbox/123456_MS_Test_Pipeline_Deploy

主要分支

main 分支一律代表在生產環境中執行的程式碼。程式碼從 分支main、開發，然後合併回 main。來自 的部署main可以以任何環境為目標。若要防止刪除，請啟用分支的main分支保護。

命名慣例：	main

hotfix 分支

在以幹線為基礎的工作流程中沒有專用hotfix分支。Hotfix 使用feature分支。

Trunk 策略的優點和缺點

主體分支策略非常適合具有強大溝通技能的小型、成熟開發團隊。如果您有應用程式的連續滾動功能版本，它也可以正常運作。如果您有大型或分段的開發團隊，或如果您有廣泛的排程功能版本，則不適合。合併衝突將在此模型中發生，因此請注意，解決合併衝突是關鍵技能。所有團隊成員都必須接受相應的訓練。

優點

以主體為基礎的開發提供多種優勢，可以改善開發程序、簡化協同合作，並增強軟體的整體品質。以下是一些主要優點：

- 更快速的意見回饋迴圈 – 透過以幹線為基礎的開發，開發人員會經常整合其程式碼變更，通常是一天多次。這可讓潛在問題的意見回饋更快，並協助開發人員比以功能為基礎的開發模型更快地識別和修正問題。
- 減少合併衝突 – 在以幹線為基礎的開發中，因為變更會持續整合，所以將大型複雜合併衝突的風險降至最低。這有助於維護更簡潔的程式碼基礎，並減少解決衝突所花費的時間。在以功能為基礎的開發中，解決衝突既耗時又容易出錯。

- 改善協作 – 以主體為基礎的開發鼓勵開發人員在同一分支上合作，在團隊中促進更好的溝通和協作。這可能會導致更快的問題解決和更具凝聚力的團隊動力。
- 更簡單的程式碼檢閱 – 由於程式碼變更在以幹線為基礎的開發中較小且更頻繁，因此執行完整的程式碼檢閱會更容易。較小的變更通常更容易理解和檢閱，從而更有效地識別潛在的問題和改進。
- 持續整合和交付 – 以主體為基礎的開發支援持續整合和持續交付 (CI/CD) 的原則。透過將程式碼基礎保持在可釋放狀態並頻繁整合變更，團隊可以更輕鬆地採用 CI/CD 實務，進而加快部署週期並改善軟體品質。
- 增強的程式碼品質 – 透過頻繁的整合、測試和程式碼檢閱，以幹線為基礎的開發可以改善整體程式碼品質。開發人員可以更快地發現和修正問題，降低技術負債隨著時間累積的可能性。
- 簡化的分支策略 – 以主體為基礎的開發可透過減少長期分支的數量來簡化分支策略。這可以更輕鬆地管理和維護程式碼庫，尤其是大型專案或團隊。

缺點

以主體為基礎的開發確實有一些缺點，這可能會影響開發程序和團隊動態。以下是幾個顯著的缺點：

- 有限的隔離 – 由於所有開發人員都在同一個分支上工作，團隊中的每個人都會立即看到他們的變更。這可能會導致干擾或衝突，導致意外的副作用或破壞建置。相反地，以功能為基礎的開發可以更好地隔離變更，以便開發人員可以更獨立地工作。
- 提高測試壓力 – 以主體為基礎的開發依賴於持續整合和自動化測試，以快速發現問題。不過，這種方法可能會對測試基礎設施造成很大的壓力，並且需要維護良好的測試套件。如果測試不全面或可靠，可能會導致主分支中未偵測到的問題。
- 較少控制 版本 – 以主體為基礎的開發旨在讓程式碼基底保持在持續可釋放的狀態。雖然這可能很有利，但不一定適合具有嚴格發行排程或需要同時發行特定功能的專案。功能型開發可進一步控制發行功能的時間和方式。
- 程式碼流失 – 隨著開發人員持續將變更整合到主分支中，以幹線為基礎的開發可能會導致程式碼流失增加。這可能會讓開發人員難以追蹤程式碼庫的目前狀態，並可能在嘗試了解最近變更的效果時造成混淆。
- 需要強大的團隊文化 – 以主體為基礎的開發需要團隊成員之間高度的紀律、溝通和協作。維護這可能具有挑戰性，特別是在較大的團隊中或與對此方法經驗較少的開發人員合作時。
- 可擴展性挑戰 – 隨著開發團隊的大小增加，整合到主分支的程式碼變更數量可能會快速增加。這可能會導致更頻繁的建置中斷和測試失敗，使得程式碼基底難以保持可釋放狀態。

GitHub 流程分支策略

GitHub Flow 是由 GitHub 開發的輕量型分支型工作流程。GitHub Flow 是以短期功能分支的概念為基礎，這些分支會在功能完成並準備好部署時合併至主要分支。GitHub Flow 的主要原則如下：

- 分支很輕量 – 開發人員只需按幾下滑鼠就能為其工作建立功能分支，改善協作和實驗的能力，而不會影響主要分支。
- 持續部署 – 變更會在合併到主分支後立即部署，以便快速回饋和反覆運算。
- 合併請求 – 開發人員使用合併請求來啟動討論和檢閱程序，然後再將其變更合併到主分支。

如需 GitHub Flow 的詳細資訊，請參閱下列資源：

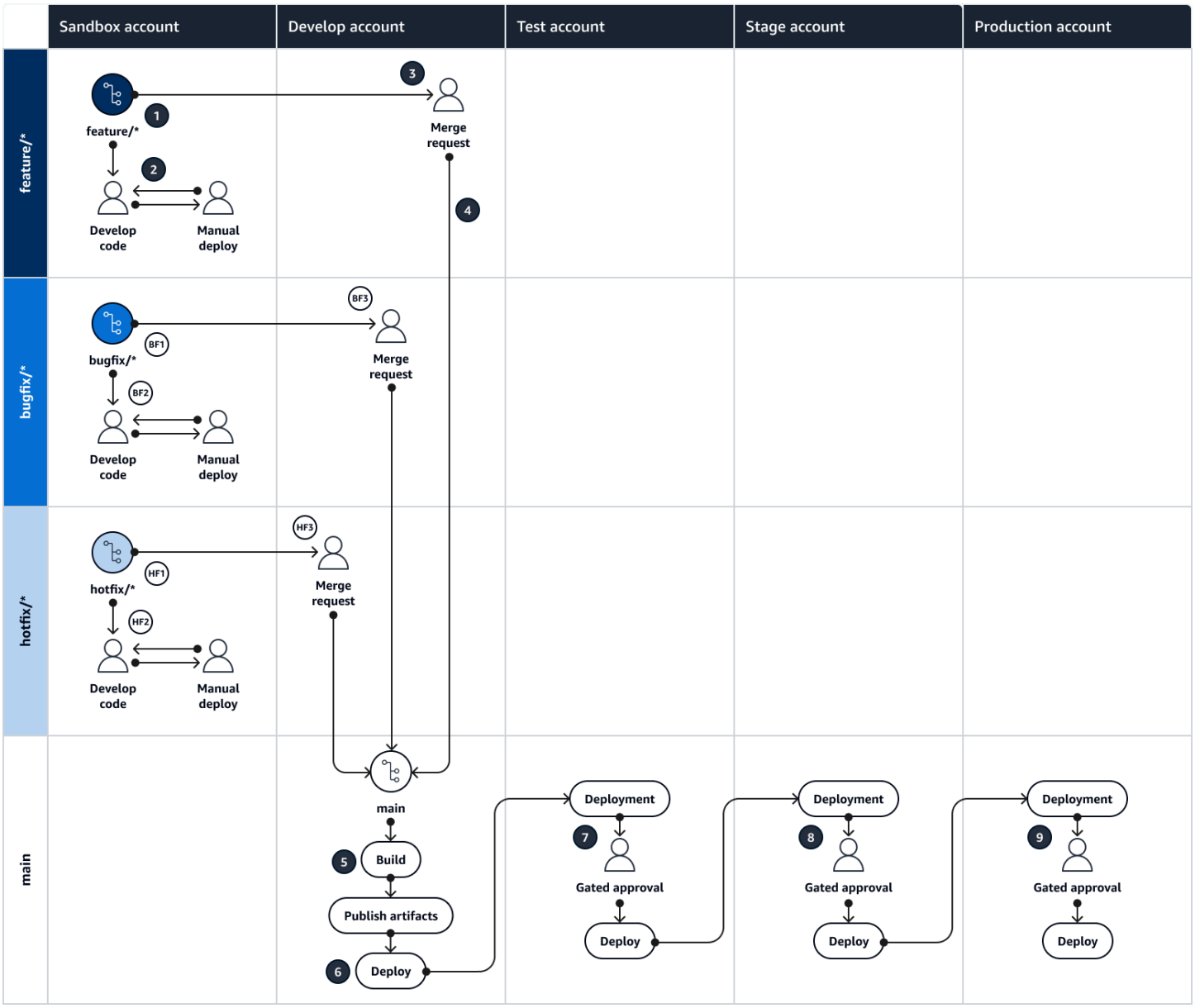
- 針對多帳戶 DevOps 環境實作 GitHub 流程分支策略 (AWS 方案指引)
- GitHub Flow Quickstart (GitHub 文件)
- 為什麼選擇 GitHub 流程？ (GitHub Flow 網站)

本節主題：

- GitHub 流程策略的視覺化概觀
- GitHub 流程策略中的分支
- GitHub Flow 策略的優點和缺點

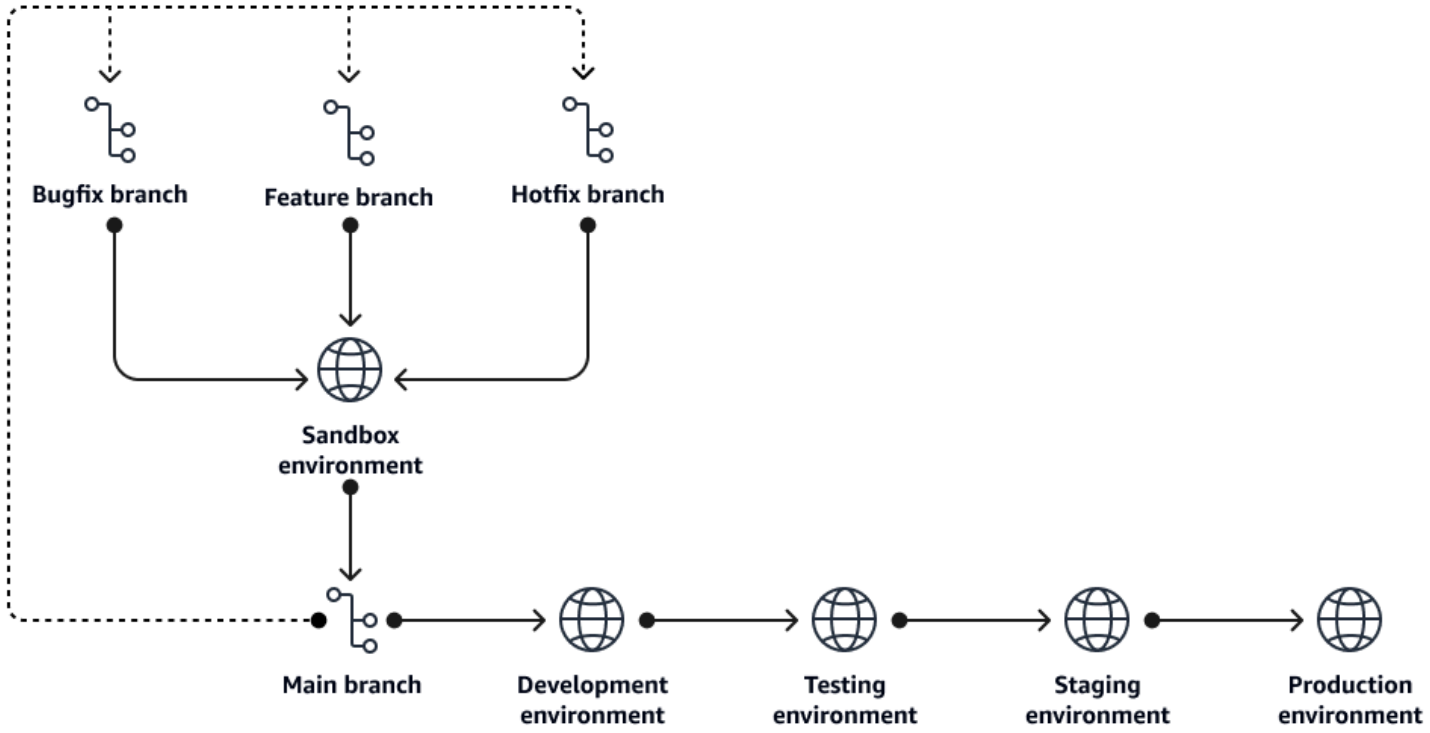
GitHub 流程策略的視覺化概觀

下圖可以像 Punnett 方形一樣使用，以了解 GitHub 流程分支策略。將垂直軸上的分支與水平軸上的 AWS 環境對齊，以決定在每個案例中要執行的動作。圓圈數字會引導您完成圖表中所呈現的動作順序。此圖表顯示 GitHub 流程分支策略的開發工作流程，從沙盒環境中的特徵分支到主分支的生產版本。如需每個環境中發生之活動的詳細資訊，請參閱本指南中的 DevOps 環境。



GitHub 流程策略中的分支

GitHub 流程分支策略通常具有下列分支。



功能分支

您可以在feature分支中開發功能。若要建立feature分支，您可以從main分支中分支。開發人員在feature分支中反覆、遞交和測試程式碼。當功能完成時，開發人員會透過向 建立合併請求來提升功能main。

命名慣例：	feature/<story number>_<developer initials>_<descriptor>
命名慣例範例：	feature/123456_MS_Implement_Feature_A

bugfix 分支

bugfix 分支用於修正問題。這些分支會從main分支分支分支出來。在沙盒或任何較低環境中測試錯誤修正之後，可以透過合併main請求將其合併到更高的環境。這是組織和追蹤的建議命名慣例，也可以使用功能分支來管理此程序。

命名慣例：	bugfix/<ticket number>_<developer initials>_<descriptor>
命名慣例範例：	bugfix/123456_MS_Fix_Problem_A

hotfix 分支

hotfix 分支用於解決高影響的關鍵問題，並將開發人員與生產環境中部署的程式碼之間的延遲降到最低。這些分支會從main分支分支分支出來。在沙盒或任何較低環境中測試 Hotfix 之後，可以透過合併main請求將 Hotfix 合併到更高環境。這是組織和追蹤的建議命名慣例，也可以使用功能分支來管理此程序。

命名慣例：	hotfix/<ticket number>_<developer initials>_<descriptor>
命名慣例範例：	hotfix/123456_MS_Fix_Problem_A

主要分支

main 分支一律代表在生產環境中執行的程式碼。程式碼會使用合併請求，從main分支合併到feature分支中。為了防止刪除和防止開發人員將程式碼直接推送到 main，請啟用分支的main分支保護。

命名慣例：	main

GitHub Flow 策略的優點和缺點

Github Flow 分支策略非常適合具有強大溝通技能的小型、成熟開發團隊。此策略非常適合想要實作持續交付的團隊，並且受到常見 CI/CD 引擎的良好支援。GitHub Flow 輕量，規則不多，並且能夠支援快速移動的團隊。如果您的團隊有嚴格的合規或發佈程序，則不適合。合併衝突在此模型中很常見，而且可能經常發生。解決合併衝突是一項關鍵技能，您必須相應地培訓所有團隊成員。

優點

GitHub Flow 提供多種優點，可改善開發程序、簡化協同合作，並增強軟體的整體品質。以下是一些主要優點：

- 彈性且輕量 – GitHub Flow 是一種輕量且靈活的工作流程，可協助開發人員協作進行軟體開發專案。它允許以最小的複雜性進行快速反覆運算和實驗。
- 簡化協同合作 – GitHub Flow 提供清晰且簡化的流程來管理功能開發。它鼓勵可以快速檢閱和合併的小型重點變更，從而提高效率。
- 清除版本控制 – 使用 GitHub Flow，每個變更都會在個別分支中進行。這會建立清楚且可追蹤的版本控制歷史記錄。這有助於開發人員追蹤和了解變更、視需要還原，以及維護可靠的程式碼庫。
- 無縫持續整合 – GitHub Flow 與持續整合工具整合。建立提取請求可以啟動自動化測試和部署程序。CI 工具可協助您在變更合併到`main`分支之前徹底測試變更，降低將錯誤引入程式碼庫的風險。
- 快速意見回饋和持續改善 – GitHub Flow 透過提取請求提升頻繁的程式碼檢閱和討論，來鼓勵快速意見回饋迴圈。這有助於儘早偵測問題、提升團隊成員之間的知識分享，最終在開發團隊中帶來更高的程式碼品質和更好的協作。
- 簡化轉返和還原 – 如果程式碼變更造成非預期的錯誤或問題，GitHub Flow 可簡化轉返或還原變更的程序。透過清楚的遞交和分支歷史記錄，更容易識別和還原有問題的變更，有助於維持穩定且功能正常的程式碼基礎。
- 輕量型學習曲線 – GitHub Flow 比 Gitflow 更容易學習和採用，尤其是已經熟悉 Git 和版本控制概念的團隊。其簡單且直覺式的分支模型可讓不同體驗層級的開發人員存取，減少與採用新開發工作流程相關的學習曲線。
- 持續開發 – GitHub Flow 可讓團隊接受持續部署方法，只要每次變更合併到`main`分支，就能立即部署。此簡化程序可消除不必要的延遲，並確保快速為使用者提供最新的更新和改進。這會導致更靈活且回應靈敏的開發週期。

缺點

雖然 GitHub Flow 提供數個優點，但也請務必考慮其潛在缺點：

- 對大型專案的適用性有限 – GitHub Flow 可能不適合具有複雜程式碼基礎和多個長期功能分支的大型專案。在這種情況下，更結構化的工作流程，例如 Gitflow，可以更好地控制並行開發和發行管理。
- 缺乏正式的發行結構 – GitHub Flow 未明確定義發行程序或支援功能，例如版本控制、修正程式或維護分支。對於需要嚴格版本管理或需要長期支援和維護的專案，這可能是限制。
- 對長期發行規劃的有限支援 – GitHub Flow 著重於短期功能分支，這些分支可能不符合需要長期發行規劃的專案，例如具有嚴格藍圖或廣泛功能相依性的專案。在 GitHub Flow 的限制範圍內，管理複雜的發行排程可能具有挑戰性。
- 可能發生頻繁的合併衝突 – 由於 GitHub Flow 鼓勵頻繁的分支和合併，因此可能會發生合併衝突，尤其是在具有大量開發活動的專案中。解決這些衝突可能很耗時，而且可能需要開發團隊的額外努力。

- 缺少正式化工作流程階段 – GitHub Flow 不會定義開發的明確階段，例如 alpha、beta 或發行候選階段。這可能會讓 更難以傳達專案的目前狀態，或不同分支或版本的穩定性層級。
- 中斷變更的影響 – 由於 GitHub Flow 鼓勵經常將變更合併到`main`分支中，因此引入會影響程式碼庫穩定性的中斷變更的風險較高。嚴格的程式碼檢閱和測試實務對於有效降低此風險至關重要。

Gitflow 分支策略

Gitflow 是一種分支模型，涉及使用多個分支將程式碼從開發移至生產環境。Gitflow 非常適合具有排程發行週期且需要將功能集合定義為發行版本的團隊。開發會在個別特徵分支中完成，這些分支會在獲得核准的情況下合併到用於整合的開發分支中。此分支中的功能視為已準備好進行生產。當開發分支中累積所有計劃的功能時，就會建立發行分支以部署到上層環境。此區隔可改善對依定義排程將哪些變更移至哪個具名環境的控制。如有必要，您可以將此程序加速為更快的部署模型。

如需 Gitflow 分支策略的詳細資訊，請參閱下列資源：

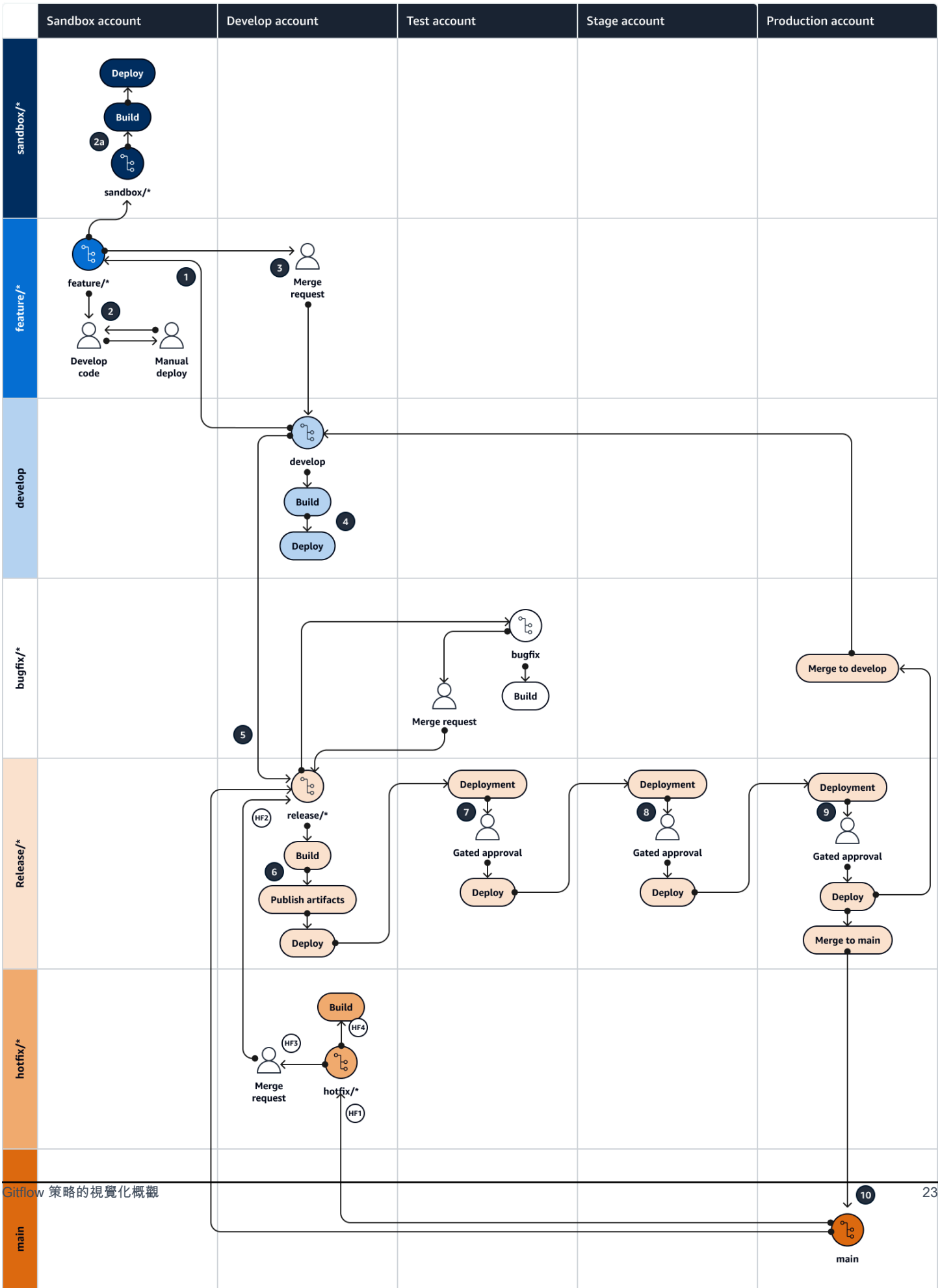
- 實作多帳戶 DevOps 環境的 Gitflow 分支策略 (AWS 方案指引)
- 原始 Gitflow 部落格 (Vincent Driessen 部落格文章)
- Gitflow 工作流程 (Atlassian)

本節主題：

- Gitflow 策略的視覺化概觀
- Gitflow 策略中的分支
- Gitflow 策略的優點和缺點

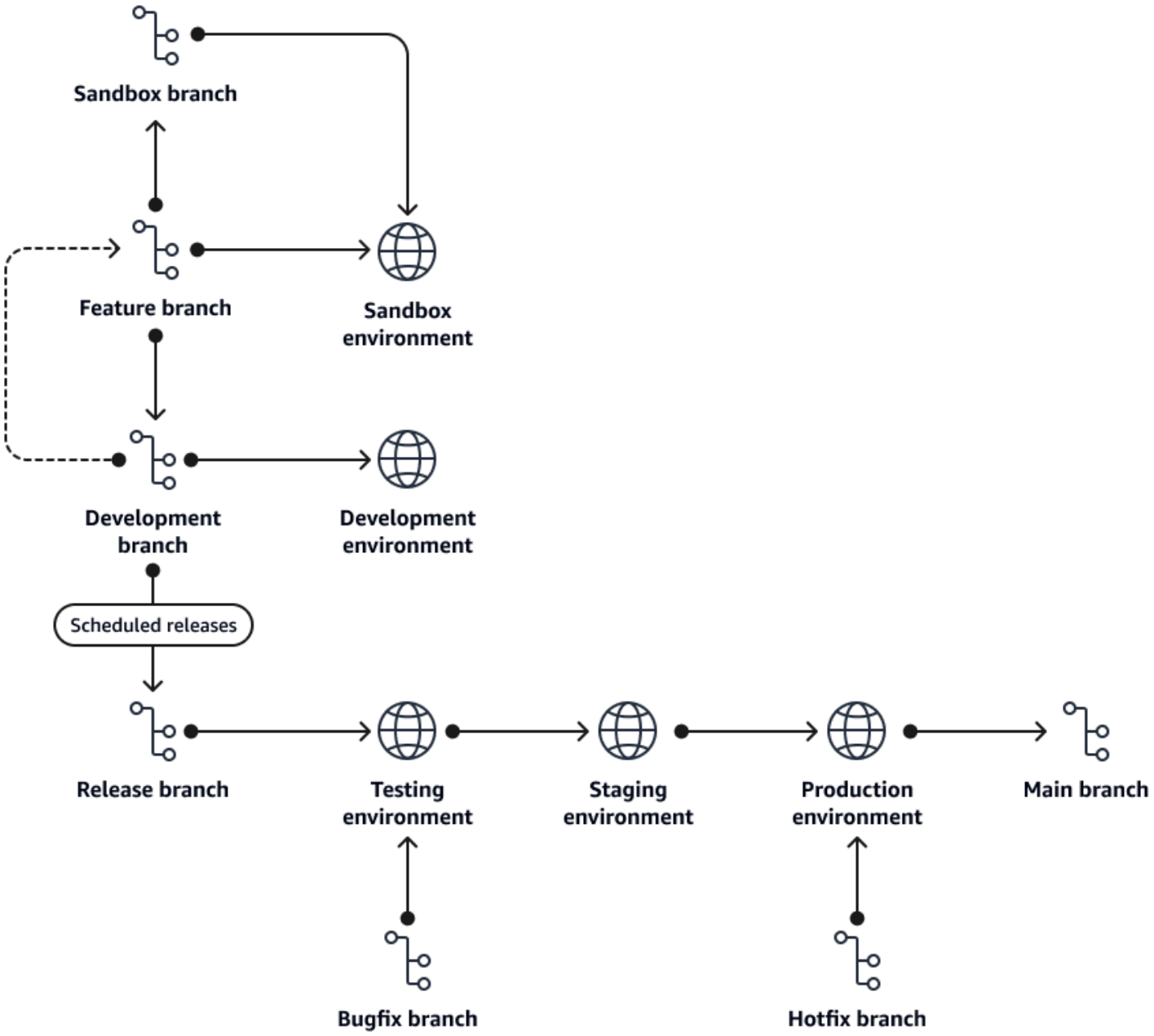
Gitflow 策略的視覺化概觀

下圖可以像 Punnett 方形一樣使用，以了解 Gitflow 分支策略。將垂直軸上的分支與水平軸上的 AWS 環境對齊，以決定在每個案例中要執行的動作。圓圈數字會引導您完成圖表中所呈現的動作順序。如需每個環境中發生之活動的詳細資訊，請參閱本指南中的 DevOps 環境。


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
Merge to develop
5
release/*
HF2
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 分支策略通常具有下列分支。



功能分支

Feature 分支是您開發功能的短期分支。feature 分支是透過分支離開develop分支來建立的。開發人員在feature分支中反覆執行、遞交和測試程式碼。當功能完成時，開發人員會提升功能。只有兩個路徑從特徵分支轉送：

- 合併至sandbox分支
- 在develop分支中建立合併請求

命名慣例：	feature/<story number>_<developer initials>_<descriptor>
命名慣例範例：	feature/123456_MS_Implement_Feature_A

沙盒分支

sandbox 分支是 Gitflow 的非標準短期分支。不過，它適用於 CI/CD 管道開發。sandbox 分支主要用於下列目的：

- 使用 CI/CD 管道而非手動部署，對沙盒環境執行完整部署。
- 在較低環境中提交合併請求進行完整測試之前，開發和測試管道，例如開發或測試。

Sandbox 分支本質上是暫時的，並非長期存在。在特定測試完成後，應將其刪除。

命名慣例：	sandbox/<story number>_<developer initials>_<descriptor>
命名慣例範例：	sandbox/123456_MS_Test_Pipeline_Deploy

開發分支

develop 分支是長期分支，可將功能整合、建置、驗證和部署到開發環境。所有feature分支都會合併到develop分支中。合併至develop分支是透過需要成功建置和兩個開發人員核准的合併請求完成。若要防止刪除，請在分支上啟用develop分支保護。

命名慣例：	develop

發行分支

在 Gitflow 中，release分支是短期分支。這些分支很特殊，因為您可以將它們部署到多個環境，採用建置一次、部署許多方法。 Release分支可以鎖定測試、預備或生產環境。在開發團隊決定將功能提升到更高的環境之後，他們會建立新的release分支，並使用先前版本的版本編號遞增。在每個環境的閘道上，部署需要手動核准才能繼續。 Release分支應該需要變更合併請求。

在release分支部署到生產環境之後，應該將其合併回 develop和 main分支，以確保任何錯誤修正或 Hotfix 都合併回未來的開發工作。

命名慣例：	release/v{major}.{minor}
命名慣例範例：	release/v1.0

主要分支

main 分支是長期分支，一律代表在生產環境中執行的程式碼。從發行管道成功部署後，程式碼main會自動從發行分支合併到分支。若要防止刪除，請在分支上啟用main分支保護。

命名慣例：	main

bugfix 分支

bugfix 分支是短期分支，用於修正尚未發佈至生產環境的發行分支中的問題。bugfix 分支應該僅用於將release分支中的修正提升為測試、預備或生產環境。bugfix 分支一律從release分支分支分支。

測試錯誤修正之後，可以透過合併請求將其提升為release分支。然後，您可以按照標準發行程序向前推送release分支。

命名慣例：	bugfix/<ticket>_<developer initials>_<descriptor>
命名慣例範例：	bugfix/123456_MS_Fix_Problem_A

hotfix 分支

hotfix 分支是用於修正生產中問題的短期分支。它僅用於提升必須加速才能到達生產環境的修正。hotfix 分支一律從 分支main。

測試 Hotfix 之後，您可以透過將請求合併到從 建立的release分支，將其提升至生產環境main。然後，您可以遵循標準發行程序，將release分支向前推送以進行測試。

命名慣例：	hotfix/<ticket>_<developer initials>_<descriptor>
命名慣例範例：	hotfix/123456_MS_Fix_Problem_A

Gitflow 策略的優點和缺點

Gitflow 分支策略非常適合具有嚴格發行和合規要求的更大、更分佈的團隊。Gitflow 為組織的可預測發行週期做出貢獻，這通常是大型組織所偏好的。Gitflow 也非常適合需要護欄才能正確完成其軟體開發生命週期的團隊。這是因為策略中有許多檢閱和品質保證的機會。Gitflow 也非常適合必須同時維護多個生產版本的團隊。GItflow 的某些缺點是它比其他分支模型更複雜，並且需要嚴格遵守模式才能成功完成。由於管理發行分支的剛性，Gitflow 不適用於努力持續交付的組織。Gitflow 發行分支可以是長期分支，如果未及時正確解決，則可能會累積技術債務。

優點

Gitflow 型開發提供多種優勢，可改善開發程序、簡化協同合作，並增強軟體的整體品質。以下是一些主要優點：

- 可預測發行程序 – Gitflow 遵循一般且可預測的發行程序。它非常適合具有定期開發和發行節奏的團隊。
- 改善協同合作 – Gitflow 鼓勵使用 feature和 release分支。這兩個分支可協助團隊以最小的相依性並行運作。
- 非常適合多個環境 – Gitflow 使用release分支，這可能是更長的分支。這些分支可讓團隊以較長時間的個別版本為目標。
- 生產中的多個版本 – 如果您的團隊在生產環境中支援多個版本的軟體，則 Gitflow release分支支援此要求。
- 內建程式碼品質審查 – Gitflow 要求並鼓勵在程式碼提升到另一個環境之前，使用程式碼審查和核准。此程序要求所有程式碼提升執行此步驟，以消除開發人員之間的摩擦。

- 組織優勢 – Gitflow 在組織層級也有優勢。Gitflow 鼓勵使用標準發行週期，這有助於組織了解和預測發行排程。由於企業現在了解何時可以交付新功能，因此由於有設定交付日期，因此時間軸的摩擦會減少。

缺點

Gitflow 型開發確實有一些缺點，可能會影響開發程序和團隊動態。以下是幾個顯著的缺點：

- 複雜性 – Gitflow 是新團隊學習的複雜模式，您必須遵循 Gitflow 的規則才能成功使用。
- 持續部署 – Gitflow 不適合許多部署以快速方式發佈至生產環境的模型。這是因為 Gitflow 需要使用多個分支和管理`release`分支的嚴格工作流程。
- 分支管理 – Gitflow 使用許多分支，這可能會變得難以維護。追蹤各種分支並合併發行的程式碼，以便讓分支彼此正確對齊，可能具有挑戰性。
- 技術負債 – 由於 Gitflow 發行的速度通常比其他分支模型慢，因此有更多功能可以累積發行，這可能會導致技術負債累積。

團隊在決定 Gitflow 型開發是否為適合其專案的正確方法時，應仔細考慮這些缺點。

後續步驟

本指南解釋了三種常見的 Git 分支策略之間的差異： GitHub流量，Gitflow 和幹線。它詳細描述了他們的工作流程，並提供了每個工作流程的優點和缺點。接下來的步驟是為您的組織選擇其中一個標準工作流程。若要實作這些分支策略之一，請參閱下列內容：

- 為多帳戶環境實作幹線分支 DevOps 策略
- 為多帳戶環境實作 GitHub Flow 分支 DevOps 策略
- 為多帳戶環境實施 Gitflow 分支策略 DevOps

如果您不確定團隊要從何處開始使用 Git 和 DevOps 程序，我們建議您選擇標準解決方案並進行測試。使用標準分支慣例有助於團隊建立在現有文件之上，並了解哪些方法最適合他們。

如果策略不適用於您的組織或開發團隊，請不要害怕改變策略。開發團隊的需求和需求可能會隨著時間而改變，並且沒有單一的完美解決方案。

資源

本指南不包含 Git 的訓練；不過，如果您需要這項訓練，網際網路上有許多高品質的資源可供使用。我們建議您從 Git 文件網站開始。

下列資源可協助您在 AWS 雲端.

AWS 規定指引

- 為多帳戶環境實作幹線分支 DevOps 策略
- 為多帳戶環境實作 GitHub Flow 分支 DevOps 策略
- 為多帳戶環境實施 Gitflow 分支策略 DevOps

其他 AWS 指引

- AWS DevOps 指引
- AWS 部署管線參考架構
- 什麼是 DevOps？
- DevOps資源

其他資源

- 十二因素應用程序方法（12 因素 .net）
- Git 的秘密（ ）GitHub
- Gitflow
  - 原來的 Gitflow 博客（文森特·德森博客文章）
  - Gitflow 工作流程 (大地圖)
  - 啟用 Gitflow GitHub：如何使用 Git 流程工作流程搭配 GitHub 基礎存放庫 (影片) YouTube
  - Git 流程初始化示例（YouTube 視頻）
  - 從開始到結束的 Gitflow 發布分支（YouTube 視頻）
- GitHub 流程
  - GitHub 流快速入門（GitHub 文檔）

  - 為什麼要 GitHub 流動？（GitHub 流量網站）
- 行李箱
  - 基於主幹的開發簡介（基於後備箱的開發網站）

貢獻者

撰寫

- 邁克·斯蒂芬斯，高級雲端應用程式架構師 AWS
- Rayjan 威爾遜，高級雲應用程序架構師， AWS
- 阿布拉什·維諾德，團隊負責人，高級雲端應用程式架構師， AWS

檢閱

- 史蒂芬 DiCato, 高級安全顧問, AWS
- 雲端應用程式架構師 AWS
- 史蒂芬·古根海默，團隊負責人，高級雲應用程序架構師， AWS

技術寫作

- 莉莉 AbouHarb，高級技術作家， AWS

文件歷史紀錄

下表描述了本指南的重大變更。如果您想收到有關未來更新的通知，可以訂閱 RSS 摘要。

變更	描述	日期
更新	我們取代了中繼線策略中分支中的映像、GitHub 流程策略中的分支，以及 Gitflow 策略區段中的分支。	2026 年 6 月 5 日
初次出版	—	2024 年 2 月 15 日

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。