gi-pov-2

Technical point of view documents for GitHub and IBM

GitHub/IBM POV-2 (GI-POV-2)
Branching Strategies

Prepared by: Christopher Kovacs
Date: May 5, 2025
Version: 1.0

Executive Summary

Branching strategies are essential frameworks that guide how development teams collaborate on code changes within version control systems. They define how code flows from development to production and significantly impact a team's ability to deliver software efficiently, safely, and predictably. The choice of branching strategy affects release cadence, code quality, and the team's ability to respond to production issues.

This document examines three prominent branching strategies—Trunk-Based Development, Branch-for-Release, and Git Flow—and discusses the release management practices necessary for their successful implementation. Each strategy offers distinct advantages and challenges, making them suitable for different team sizes, release cadences, and organizational contexts. Additionally, this document explores how feature flags, parallel change patterns, and versioning approaches can complement these branching strategies to enhance deployment safety and flexibility.

Selecting the appropriate branching strategy is a critical architectural decision that should align with an organization's development culture, release requirements, and operational capabilities. This document aims to provide guidance for making informed decisions about which branching strategy to adopt and how to implement it effectively.

Guiding Principles

1. **Simplicity and clarity.** Adopt branching strategies that are easy to understand and follow, reducing cognitive overhead for developers and minimizing the risk of errors.
2. **Continuous integration.** Prioritize frequent integration of code changes to detect conflicts early and ensure that the codebase remains in a releasable state.
3. **Deployment safety.** Implement practices that minimize the risk of introducing defects into production while enabling rapid recovery when issues occur.
4. **Development velocity.** Choose approaches that enable teams to deliver value quickly without compromising quality or stability.
5. **Scalability.** Ensure that the selected branching strategy can accommodate growth in team size, codebase complexity, and deployment frequency.

Recommended Best Practices

1. Trunk-Based Development

Trunk-Based Development is a branching strategy where developers collaborate on code in a single branch called ‘trunk’ (or ‘main’ in Git). This approach emphasizes continuous integration and short-lived feature branches that merge back to the trunk frequently.

Workflow

- Developers create short-lived feature branches (typically lasting less than a day) or commit directly to the trunk for smaller changes.
- Continuous integration runs on all commits to verify that they don’t break the build.
- The trunk is kept in a releasable state at all times.
- Releases can be made directly from the trunk or through dedicated release branches.

Required Release Management Practices

1. **Robust automated testing.** Comprehensive test suites must run on every commit to ensure the trunk remains stable.
2. **Feature flags.** Use feature flags (like those provided by LaunchDarkly) to hide incomplete features in production until they're ready, allowing code to be merged to trunk before features are complete.
3. **Continuous deployment pipeline.** Implement a reliable CI/CD pipeline that can quickly deploy changes to production.
4. **Monitoring and observability.** Robust monitoring systems must be in place to quickly detect and address issues in production.
5. **Branch by abstraction.** For larger changes, use the branch by abstraction technique to make significant architectural changes without long-lived branches.

Benefits

- **Reduced merge conflicts.** Frequent integration minimizes complex merge conflicts.
- **Faster feedback cycles.** Developers get immediate feedback on their changes.
- **Improved collaboration.** The entire team works on the same codebase, enhancing knowledge sharing.
- **Simplified release process.** The trunk is always in a releasable state, making releases more predictable.
- **Better alignment with CI/CD.** This approach naturally supports continuous integration and delivery practices.

Drawbacks

- **Requires disciplined development practices.** Developers must ensure their changes don't break the trunk.
- **May be challenging for less experienced teams.** The approach requires strong testing and collaboration skills.
- **Feature flags add complexity.** Managing feature flags introduces additional complexity to the codebase.
- **Potential for incomplete features in production.** Even with feature flags, incomplete code exists in the production codebase.

Alternative - GitHub Flow

GitHub Flow is a simplified alternative to Trunk-Based Development that maintains the focus on the main branch while introducing a more structured pull request workflow. Like Trunk-Based Development, GitHub Flow centers around a single main branch that is always deployable, but differs in its approach to feature development. While Trunk-Based Development emphasizes extremely short-lived branches or direct commits to trunk, GitHub Flow allows for longer-lived feature branches with a formal pull request process for code review before merging. Both strategies prioritize continuous integration and deployment, but GitHub Flow provides more opportunity for collaboration and review before changes reach the main branch, making it particularly well-suited for open source projects and teams that require more structured code review processes.

2. Branch-for-Release

Branch-for-Release is a strategy where development occurs on the main branch, and release branches are created just before a release. These branches are used for final stabilization and bug fixes specific to the release.

Workflow

- Development happens on the main branch.
- When preparing for a release, a release branch is created from the main branch.
- Only bug fixes are applied to the release branch; new features continue to be developed on the main branch.
- Critical fixes may be backported from the release branch to the main branch.
- Once the release is deployed and stable, the release branch may be archived.

Required Release Management Practices

1. **Clear release schedule.** Teams need a well-defined release cadence to plan when to create release branches.
2. **Version management.** Each release branch should be associated with a specific version number.
3. **Backporting process.** A process must exist for applying critical fixes from release branches back to the main branch.
4. **Release branch lifecycle policy.** Define how long release branches are maintained and when they are archived.
5. **Hotfix procedure.** Establish a clear process for making emergency fixes to production releases.

Benefits

- **Stable release candidates.** Release branches provide a stable codebase for final testing and deployment.
- **Support for multiple releases.** Multiple versions can be maintained simultaneously.
- **Reduced pressure on release timing.** Development can continue on the main branch while a release is being prepared.
- **Clear separation of concerns.** Development activities are separated from release stabilization.

Drawbacks

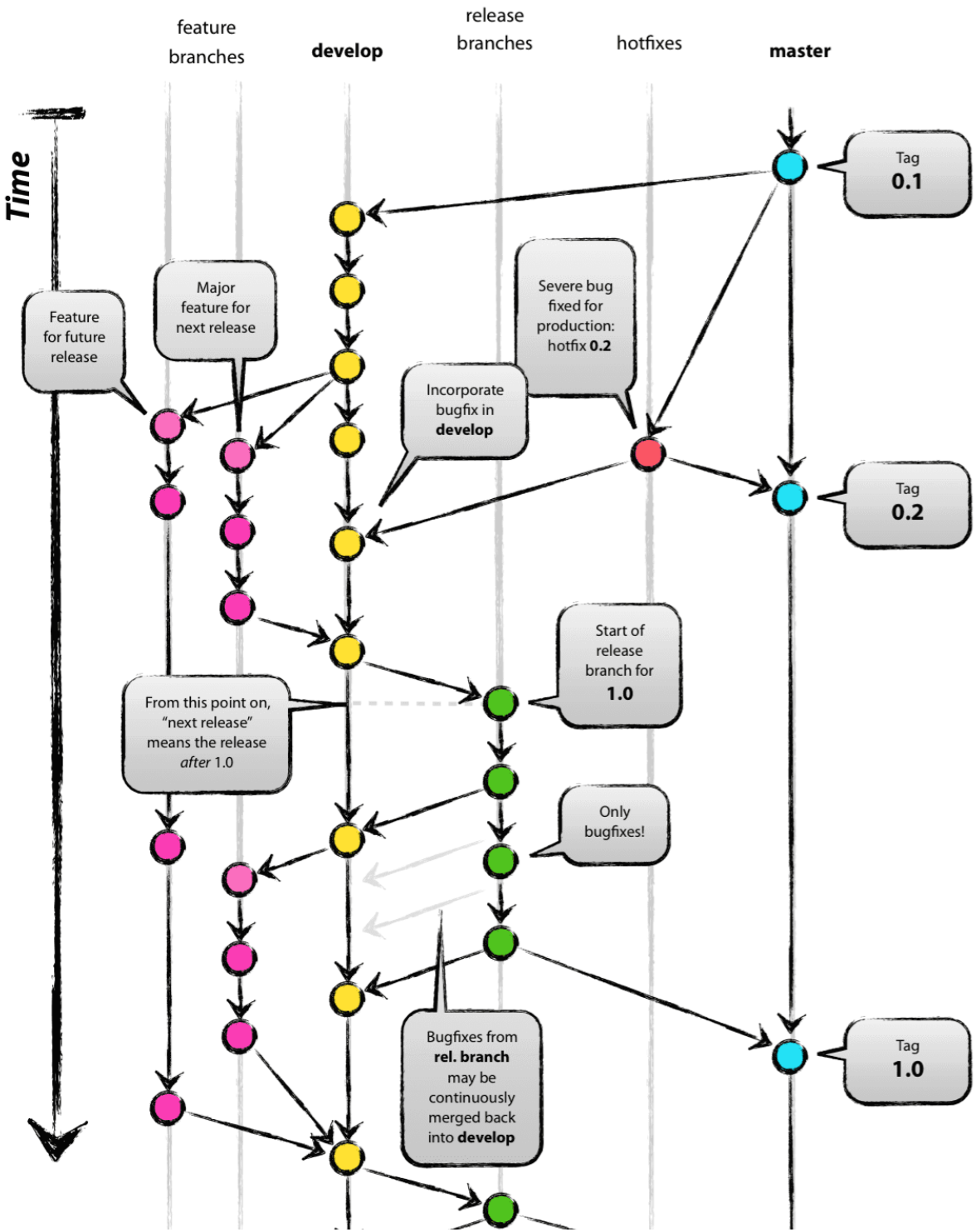
- **Potential for divergence.** The main branch and release branches can drift apart if changes aren't properly backported.
- **Increased complexity.** Managing multiple active branches adds overhead.
- **Delayed integration of fixes.** Fixes made in release branches must be manually backported to the main branch.
- **Possible confusion about where to apply changes.** Developers may be uncertain whether to apply changes to the main branch or a release branch.

3. Git Flow

Git Flow is a robust branching model that defines specific branch types and their purposes. It provides a structured framework for managing releases, features, and hotfixes through a set of well-defined branches.

Workflow

- The **main/master** branch contains production-ready code and represents the official release history.
- A parallel **develop** branch serves as the integration branch for ongoing development.
- **Feature branches** branch off from develop and merge back into develop when complete.
- **Release branches** branch off from develop when it's ready for a release, allowing for final polishing and bug fixes.
- **Hotfix branches** branch off from main/master to quickly address critical production issues.
- Release and hotfix branches merge into both main/master and develop to ensure changes are not lost.


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Required Release Management Practices

1. **Branch naming conventions.** Consistent naming for different branch types (e.g., feature/*, release/*, hotfix/*) is essential.
2. **Version tagging.** Each merge to main/master should be tagged with a version number.
3. **Release preparation process.** A defined process for creating and stabilizing release branches.
4. **Merge policies.** Clear rules for when and how branches can be merged.
5. **Environment management.** Different environments for development, testing, and production that align with the branching structure.

Benefits

- **Structured release process.** The model provides a clear path for preparing and deploying releases.
- **Parallel development.** Multiple features can be developed simultaneously without interference.
- **Support for multiple production versions.** The model accommodates maintaining multiple release versions.
- **Clear separation of concerns.** Different branch types serve different purposes, making the workflow predictable.
- **Well-suited for scheduled releases.** The model works well for products with planned release cycles.

Drawbacks

- **Complexity.** The model involves multiple branch types and merge patterns, increasing cognitive load.
- **Potential for long-lived branches.** Feature branches may become long-lived, leading to integration challenges.
- **Overhead for small teams.** The structured approach may be excessive for small teams or simple projects.
- **Less aligned with continuous delivery.** The model was designed for scheduled releases rather than continuous delivery.
- **Merge conflicts.** The frequency of merges between branches can lead to complex merge conflicts.

4. Feature Flags for Safe Deployments

Feature flags (or feature toggles) are a powerful technique that complements any branching strategy by decoupling deployment from release. They allow teams to deploy code to production without making it immediately available to users.

Integration with LaunchDarkly

LaunchDarkly provides a comprehensive feature flag management platform that enables teams to:

1. **Control feature rollout.** Gradually release features to subsets of users to minimize risk.
2. **A/B testing.** Test different implementations with different user segments.
3. **Kill switches.** Quickly disable problematic features without rolling back deployments.
4. **Targeted releases.** Release features to specific user segments based on attributes.
5. **Experimentation.** Measure the impact of features on key metrics before full rollout.

Best Practices for Feature Flags

1. **Temporary vs. permanent flags.** Distinguish between flags used for rollout control (temporary) and those used for long-term configuration (permanent).
2. **Flag lifecycle management.** Establish processes for creating, using, and removing flags to prevent flag debt.
3. **Default safe values.** Ensure that if flag evaluation fails, the system defaults to a safe state.
4. **Testing with flags.** Test both flag states to ensure the system behaves correctly in all scenarios.
5. **Documentation.** Maintain clear documentation of active flags and their purpose.

5. Parallel Change Pattern for Safe Refactoring

The Parallel Change pattern (also known as Expand and Contract) provides a way to implement breaking changes safely, particularly when modifying data structures or APIs. This pattern is especially valuable in environments where downtime is unacceptable.

Implementation Steps

1. **Expand phase.** Add the new implementation alongside the old one, with the system writing to both but reading from the old implementation.
2. **Migrate phase.** Gradually migrate existing data or clients to the new implementation.
3. **Contract phase.** Once migration is complete, remove the old implementation.

Benefits for Branching Strategies

- **Reduces the need for long-lived branches.** Breaking changes can be implemented incrementally on the main branch.
- **Enhances deployment safety.** Changes can be rolled back at any stage if issues are detected.
- **Supports continuous delivery.** The system remains operational throughout the change process.
- **Facilitates large-scale refactoring.** Complex architectural changes can be implemented without disrupting development.

6. Versioning Approaches

Proper versioning is essential for managing releases and communicating changes to users. Different versioning schemes serve different purposes:

Semantic Versioning (SemVer)

SemVer uses a three-part version number (MAJOR.MINOR.PATCH) where:

- MAJOR version increments for incompatible API changes
- MINOR version increments for backward-compatible new functionality
- PATCH version increments for backward-compatible bug fixes

SemVer works well with branch-for-release strategies, as each release branch can be associated with a specific version number.

Calendar Versioning (CalVer)

CalVer bases version numbers on the date of release (e.g., YY.MM.DD or YYYY.MM). This approach:

- Provides clear information about when a release was made
- Works well for projects with regular release schedules
- Is less focused on the nature of changes and more on the timing

CalVer is often used with trunk-based development and continuous delivery, where releases happen frequently.

Romantic Versioning (RomVer)

RomVer uses more subjective criteria for version increments, focusing on the significance of changes rather than strict compatibility rules. This approach:

- Allows for more flexibility in version numbering
- Can better reflect the perceived importance of changes
- May be less predictable for consumers of the software

Conventional Commits for Automated Versioning

Conventional Commits is a specification for commit messages that makes them machine-readable, enabling automated version increments and changelog generation. The format includes:

- A type (e.g., feat, fix, chore)
- An optional scope
- A description
- Optional body and footer

For example: `feat(api): add user authentication endpoint`

When combined with SemVer, Conventional Commits can automate version increments:

- `feat:` commits trigger a MINOR version increment
- `fix:` commits trigger a PATCH version increment
- Commits with a `BREAKING CHANGE:` footer trigger a MAJOR version increment

This approach integrates well with CI/CD pipelines and reduces the manual effort required for release management.

Key Decisions and Recommendations

Decision ID	Decision Description	Rationale / Justification
1	Use Trunk-Based Development for teams practicing continuous delivery	Trunk-Based Development aligns naturally with continuous delivery practices, enabling frequent, small releases and reducing integration issues.
2	Implement Branch-for-Release for products with defined release cycles	Branch-for-Release provides stability for planned releases while allowing development to continue, making it suitable for products with scheduled releases.
3	Consider Git Flow for complex products with scheduled releases	Git Flow provides a structured framework for managing complex products with multiple versions and scheduled releases, though it comes with significant additional overhead.
4	Integrate feature flags regardless of branching strategy	Feature flags decouple deployment from release, enhancing safety and flexibility for all branching strategies.
5	Use the Parallel Change pattern for breaking changes	This pattern enables safe implementation of breaking changes without long-lived branches, supporting continuous delivery.
6	Adopt SemVer or RomVer for APIs and libraries	SemVer/RomVer communicates the nature of changes clearly to consumers, making it ideal for APIs and libraries.
7	Consider CalVer for rapidly evolving products	CalVer works well for products with frequent releases where the timing of releases is more relevant than the nature of changes.

Decision ID	Decision Description	Rationale / Justification
8	Implement Conventional Commits for automated versioning	Conventional Commits reduces manual effort in version management and changelog generation, improving consistency and efficiency.

References and Additional Reading

- Trunk-Based Development
- LaunchDarkly Feature Flag Platform
- Parallel Change Pattern
- Expand and Contract Pattern
- Semantic Versioning Specification
- Romantic Versioning Specification
- Conventional Commits Specification
- Git Flow Original Article
- Feature Toggles (Feature Flags)
- Branch by Abstraction
