Branching Strategies

Current Workflow

Platform	Git Workflow	Feature Toggles	Release	Environments	Versioning	Sprint Length	Repo	Confluence
Android	~~GitFlow~~ TBD + No Release Branch	Occasionally, as needed		Dev, Ephemeral, Staging, Preprod, Prod		N/A	Mono	Branching Strategy
Web	~~GitFlow~~ ~~Moving to TBD~~ TBD + No Release Branch			Dev, Ephemeral, Staging, Preprod, Prod	N/A	N/A	Multi	Branching Strategy
iOS	~~TBD + Hybrid~~ TBD + No Release Branch	Occasionally, as needed	Branch to "codefreeze"	Dev, **Staging**, Preprod, Prod	Semantic	N/A	Mono	
Backend	TBD + Multifunctional TBD + No Release Branch (Mob)	Hardly used for Backend		Dev, Ephemeral, Staging, Preprod, Prod	Semantic	N/A	Multi	

Branch Definitions / Prefixes

Trunk Branch

- `master` - This is our trunk. Collectively, all code is merged into this

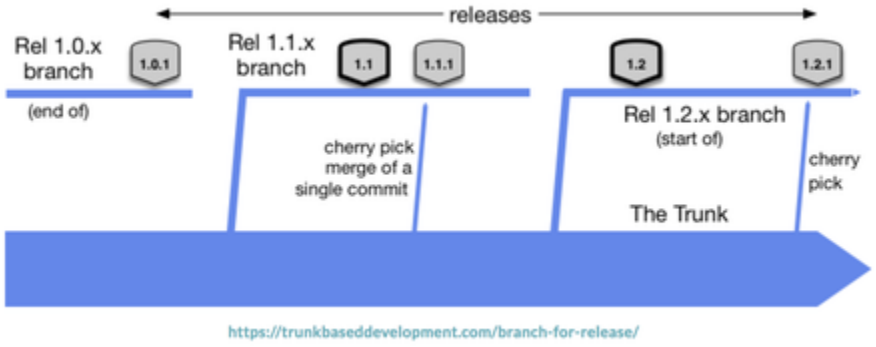
Development and Fix Branches

- The naming convention here can become a bit looser, the important thing being that these branches are **short lived**, ideally living no longer than 24-48 hours
- We do want to categorize these branches by name though:
    - `feature/* OR task/* OR topic/*` - development branch prefixes

- `bugfix/*` - Bug fix branch prefix

Version Control

- GitFlow is recommended to use GitVersion
- `Trunk`-Based Development supports Semantic Versioning Backend Versioning
- Chris Beams - How to write a Git commit message



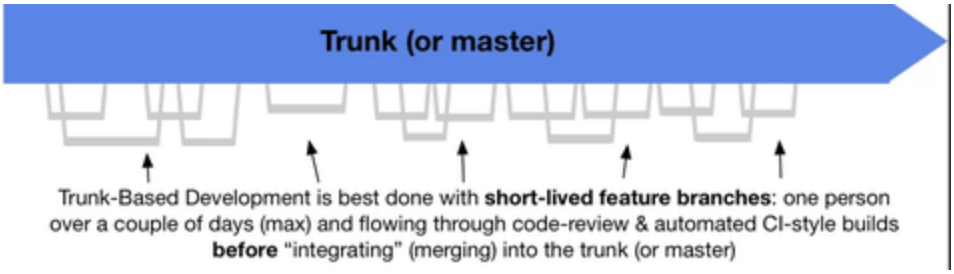
Example Naming Conventions

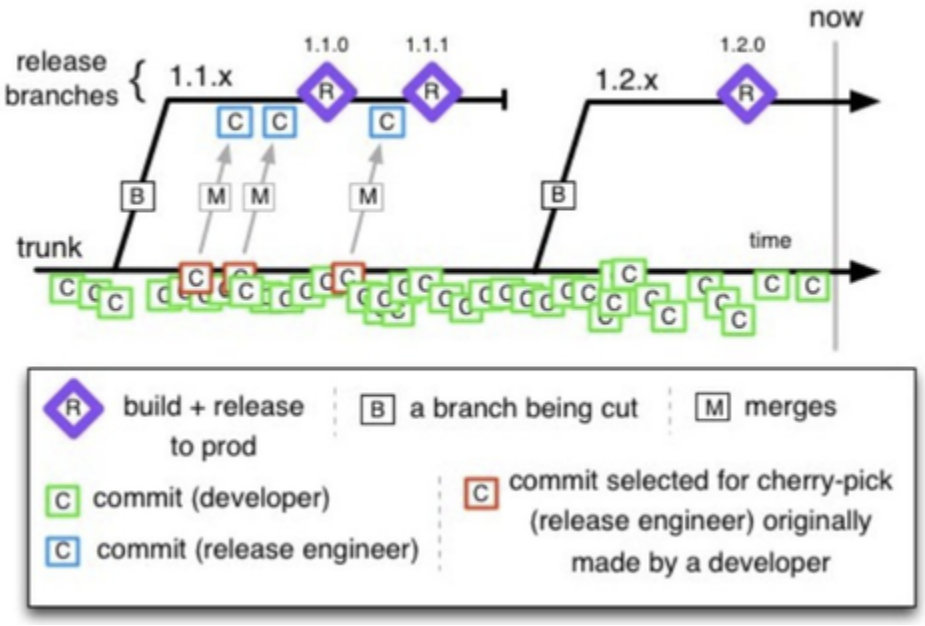
- When we name our branches, we want to aim for clear, concise names. Don't include any information that is not useful
- Typically, you just want to include Jira issue number and brief summary/title:
  - `{branch-prefix}/{jira-id}-{title-summary}`
- For example, a feature branch for a Jira ticket that drops in a vendor script in the document head might look like this:
  - `feature/pbr-98-optimizly-include`
- Similarly bugfix branch could look the same:
  - `bugfix/bac-101-card-layout-fix`

Commit Message Conventions

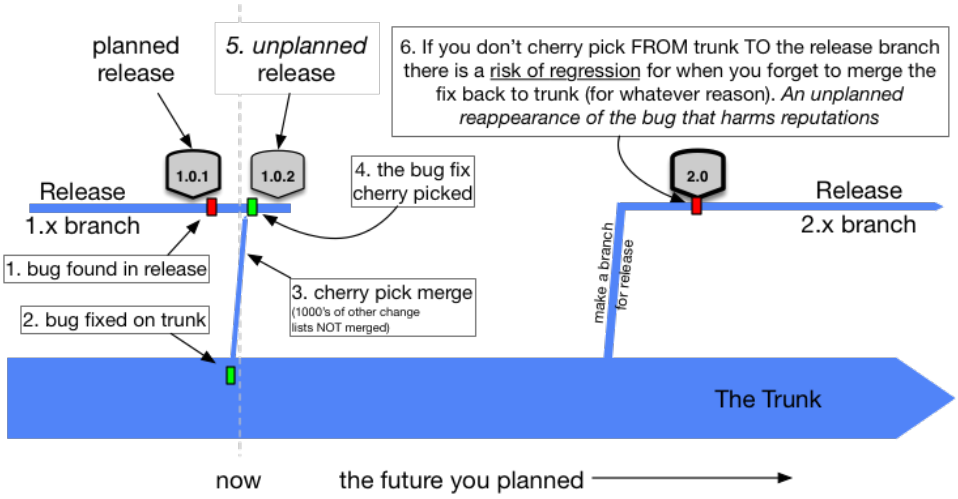
- We want our commit messages to be clear, effective, and valuable
- In general, we like to see commits that follow the Guidelines Proposed by Chris Beams
- The TLDR of this convention is:
  - Separate subject from body with a blank line
  - Limit the subject line to ~~50~~ 60 characters
  - Capitalize the subject line
  - Do not end the subject line with a period
  - Use the imperative mood in the subject line
  - Wrap the body at 72 characters
  - Use the body to explain what and why vs. how (sometimes a commit does not need a body, you will have to make that call based on the work)
- In our case, be sure to put any relevant Jira ticket numbers in the commit message
  - An example of a simple commit message might look like this:
    - `Update API payload schema - BAC-25`

Trunk Based Development



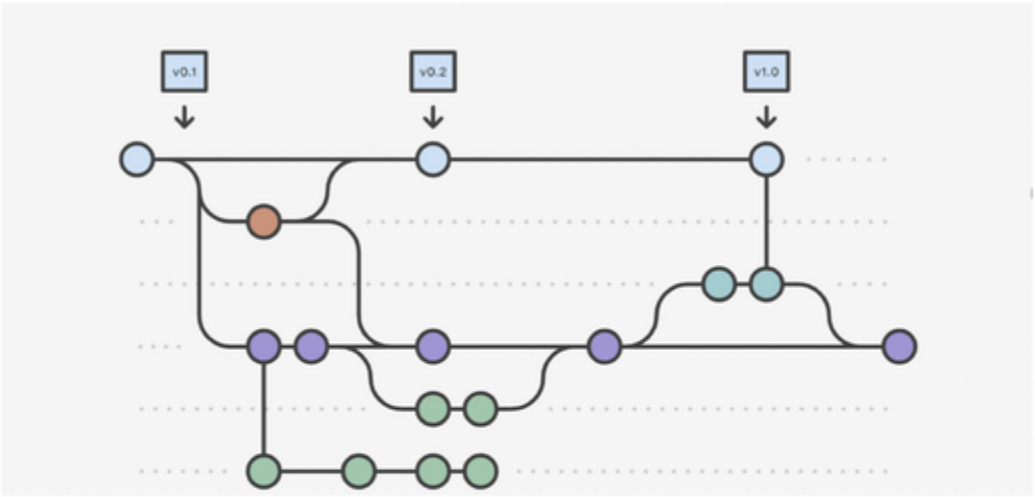


1. There is one branch called the "`Trunk`" where developers directly commit
2. A Release Manager can create release branches and no one can commit to them
3. Only if a defect **cannot** be reproduced on trunk, is permission given to fix it on the release branch, and cherry-pick back to `Trunk`
4. Developers commit small changes as often as they can
    a. Only Release Manager can commit to release branch
5. Code Review
    a. in `Trunk`-based workflow ideally should be done before commits integrate into `Trunk`.
    b. Manually, developers would push their commits to some temporary branch and, when approved, rebase those commits into `Trunk`
6. `Trunk`-Based Development
    a. You delete 'old' release branches, without merging them back to trunk



1. Planned release branch **v.1.0.1** Developers work on short lived feature branches that usually get merged back into `Trunk`
    a. Ideally multiple times a day which prevents possible merge conflicts
2. Bug fixed on `Trunk`
3. Cherry-Pick merge
4. Cherry-Pick bug fix
5. Once Bug fixed Cherry-Pick merged back into `Trunk` Cherry-Picked to release version get bumped up **v1.0.2**
6. Cherry-Pick from `Trunk` to release branch v.**2.0.0**

GitFlow



1. **Develop** branch is created from `master` **v.0.1**

2. **Preprod**, branch is created from **Develop**
3. **Feature** branches are created from **Develop**
4. When **Feature** is complete Merged to **Develop**
5. When **Preprod** branch is complete Merged into **Develop and** `master`
6. If there is an issue on `master` **Hotfix** is created from `master` Version updated **v.0.2**
7. When **Hotfix** is complete Merged to **Develop and** `master` Bumped up version number v.**1.0**

GitFlow vs Trunk-based

GitFlow

Fixes in Production

- If there is an issue on Master, Hotfix is created from master on a separate branch
- Then merged back into `Master` with updated version. (see GitFlow image)

Code Review

- Flexible with large or small commits
- Pull Request Peer Review Then approved to next phase

Testing

- Unit tests for a newly created method or function
- Unit tests should cover only the tested function
- Feature Flags Should tests run on all possible combinations of flags or just a few of them?
- Tests that cover the integration process
- For integration tests it is enough to check only two scenarios
    - Check if the toggles of all features expected to be in the next release are on
    - Check all toggles in unfinished features

Merging & Resolve

- Merge flexibility
- Using and mixing branches are easily accomplished BUT conflicts are common (more details below)

Pros

1. More CI compatible
2. Parallel Development One of the great things about GitFlow is that it makes parallel development very easy, by isolating new development from finished work
    a. New development (such as features and non-emergency bug fixes) is done in feature branches, and is only merged back into main body of code when the developer(s) is happy that the code is ready for release
    b. Although interruptions are a not fun to deal with, if you are asked to switch from one task to another, all you need to do is commit your changes and then create a new feature branch for your new task
    c. When that task is done, just checkout your original feature branch and you can continue where you left off
3. Collaboration Feature branches also make it easier for two or more developers to collaborate on the same feature, because each feature branch is a sandbox where the only changes are the changes necessary to get the new feature working
    a. Much easier to follow and track
4. Release Staging Area As new development is completed, it gets merged back into the develop branch, which is a staging area for all completed features that haven't yet been released
    a. When the next release is branched off of develop, it will automatically contain all of the new stuff that has been finished
5. Support For Emergency Fixes GitFlow supports hotfix branches branches made from a tagged release. You can use these to make an emergency change, safe in the knowledge that the hotfix will only contain your emergency fix
    a. Less risk for developers to accidentally merge in new development at the same time

Cons

1. Large commits can cause issue with tons of features/changes

    a. Merge conflicts
    b. Problematic for testing
2. You cannot test the combination of two features until they are merged into one branch
    a. When you develop features in separate branches for multiple days or weeks then problems which arise from the interaction of two features become visible too late
3. Unable to know how much time you will need for a release if the feature branches are not merged yet
4. Merge conflicts are very common when you work with Git Flow
    a. If you have multiple parallel feature branches which live for a long time then it is very likely that the same part of the code base is changed in two different branches
5. Code Freeze is strongly recommended which can slow things down

Trunk-based

Fixes in Production

- Issue is located on `Trunk` branch and try to fix it on the mainline with an additional commit
- Be aware that the mainline and the release branch contain the same code (or similar) so it should not be a problem to reproduce any failure from the release branch on the `Trunk`
- After committing, we will send the commit id to a Release Manager and he or she will cherry-pick it to the release branch

Code Review

- Smaller commits Less is more Makes review process much easier
- Pull Request Peer Review Then approved to next phase
- The code-review in trunk-based workflow ideally should be done before commits integrate into master.
- Manually, developers would push their commits to some temporary feature branch and, when approved, rebase those commits into master and push them (*optionally squashing them into a single commit*)
- There are tools that can automate such as Gerrit
    - For example
        - When pushing a commit to Gerrit, it creates an (almost invisible) temporary set of branches to hold the commit under review.
        - During review, any corrections made are amended to the commit under review and pushed again to Gerrit
        - Once approved, the commits are integrated into master atomically (user can chose how among options like rebase, cherry-pick and merge)

Testing

- Unit tests for a newly created method or function
    - Unit tests should cover only the tested function
- Tests that cover the integration process
- Feature Flags
    - Should tests run on all possible combinations of flags or just a few of them?
- Tests that cover the integration process
- For integration tests it is enough to check only two scenarios
    - Check if the toggles of all features expected to be in the next release are on
    - Check all toggles in unfinished features
- Automation Tests
    - Comprehensive automated tests creates more team confidence
    - Quicker feedback if something fails Quicker to fix an issue
    - If you are always checking in small incremental changes, test failures are easy to fix

Merging & Resolve

- There is only one branch, there are no other branches, so there is no merging
- No merging equals no merge conflicts

Cherry-Pick

- One of the most important rules of `Trunk`-Based Development
    - If there is a bug on release, you cannot push changes into the release branch

    - Commit directly to the mainline

- Best way to fix the bug is to reproduce it on the `Trunk`, then perform a fix also on the main branch, after which the Release Manager can pick this commit into release Defines Cherry-Picking
- Why not commit directly into the release branch and then merge it to the `Trunk`?
    - First, advantages of `Trunk`-Based Development is that there are usually no Merge related issues If we introduce merging from release branches into the mainline We would have merge conflicts
    - Second, there is a chance you might forget to merge it down, and then there is going to be a regression at the next release

Pros

1. Smaller conflict resolutions, mainly in case of major refactoring
2. Supports the best developing practices, including feature planning, committing small changes, and writing backward compatible code
3. More linear history, which is easier to understand and to make Cherry-Picks and reverts
4. Creates more opportunity to deploy new features faster than using feature branching
5. Enables CI
6. No code freeze needed
7. Build is always release ready One of the best practices for Continuous Delivery
8. Business is able to make really late yet low-cost decisions
    a. Scrapping part of a release
    b. Un-releasing features in production

Cons

1. Team gets larger Lots of commits for Release Manager to monitor
2. `Trunk`-Based Development does not encourage developing into separate feature branches
3. When build fails Everyone gets blocked from deploying and everyone needs to be informed
4. Feature toggles To be able to release trunk more often under Trunk Based Development, you have to add a lot of feature toggles.
    a. Feature toggles cost time and effort to add and they also increase the complexity.
    b. Then you have to remember to remove them later
5. Refactoring Large refactoring tasks, where you have to push code to trunk every day, requiring green tests all along and keeping trunk in a releasable state can be quite hard