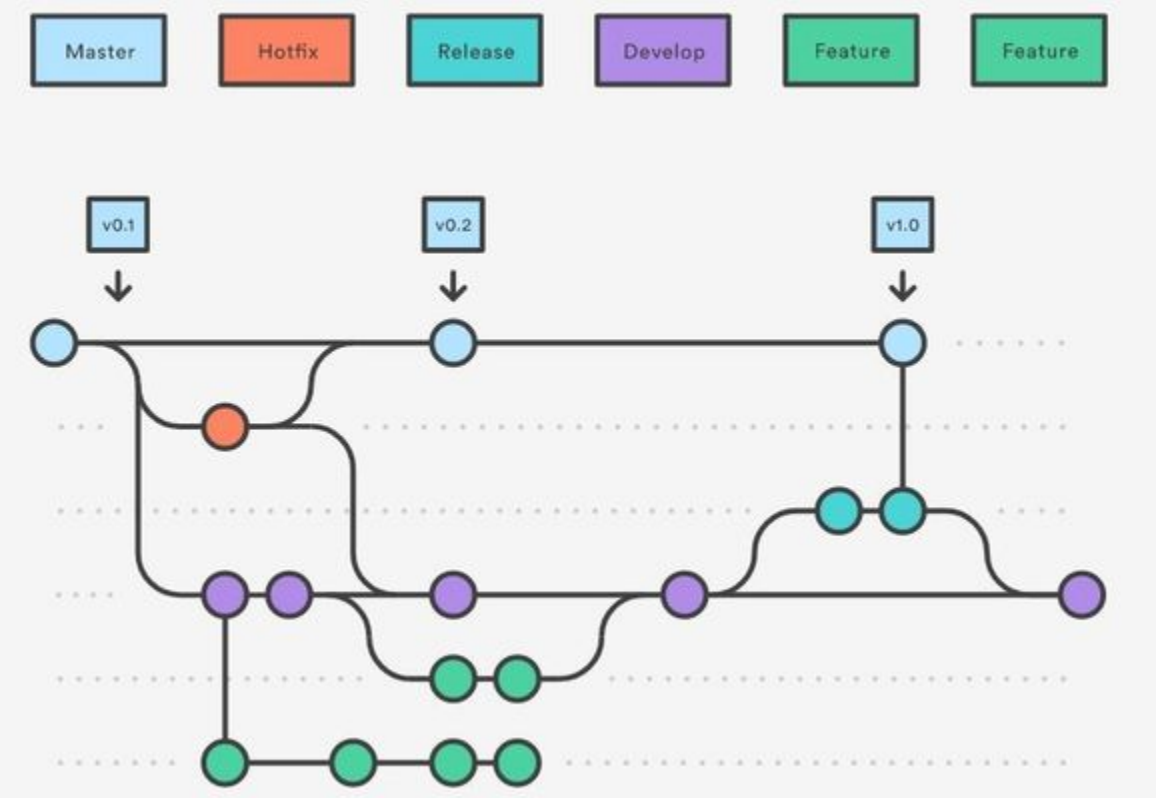
How To - BRANCHING MODEL & BEST PRACTICES

1. GITFLOW



1. The Gitflow Workflow defines a strict branching model designed around the project release.
2. Gitflow is ideally suited for projects that have a scheduled release cycle.
3. It assigns very specific roles to different branches and defines how and when they should interact.
4. It uses individual branches for preparing, maintaining, and recording releases.
5. Instead of a single `master` branch, this workflow uses two branches to record the history of the project.
6. The `master` branch stores the official release history, and the `develop` branch serves as an integration branch for features.
7. It's also convenient to tag all commits in the `master` branch with a version number.
8. Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration. But, instead of branching off of `master`, `feature` branches use `develop` as their parent branch.
9. When a feature is complete, it gets merged back into develop. Features should never interact directly with `master`.
10. Note that `feature` branches combined with the `develop` branch is, for all intents and purposes, the Feature Branch Workflow. But, the Gitflow Workflow doesn't stop there.
11. `Feature` branches are generally created off to the latest `develop` branch.
12. When you're done with the development work on the feature, the next step is to merge the `feature_branch` into `develop`.
13. Once `develop` has acquired enough features for a release (or a predetermined release date is approaching), you fork a `release` branch off of `develop`.
14. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch.
15. Once it's ready to ship, the `release` branch gets merged into `master` and tagged with a version number
16. In addition, it should be merged back into `develop`, which may have progressed since the release was initiated.
17. Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release.
18. Once the release is ready to ship, it will get merged it into `master` and `develop`, then the `release` branch will be deleted. It's important to merge back into `develop` because critical updates may have been added to the `release` branch and they need to be accessible to new features.
19. Maintenance or "`hotfix`" branches are used to quickly patch production releases. `Hotfix` branches are a lot like `release` branches and `feature` branches except they're based on `master` instead of `develop`.
20. This is the only branch that should fork directly off of `master`. As soon as the fix is complete, it should be merged into both `master` and `develop` (or the current `release` branch), and `master` should be tagged with an updated version number.

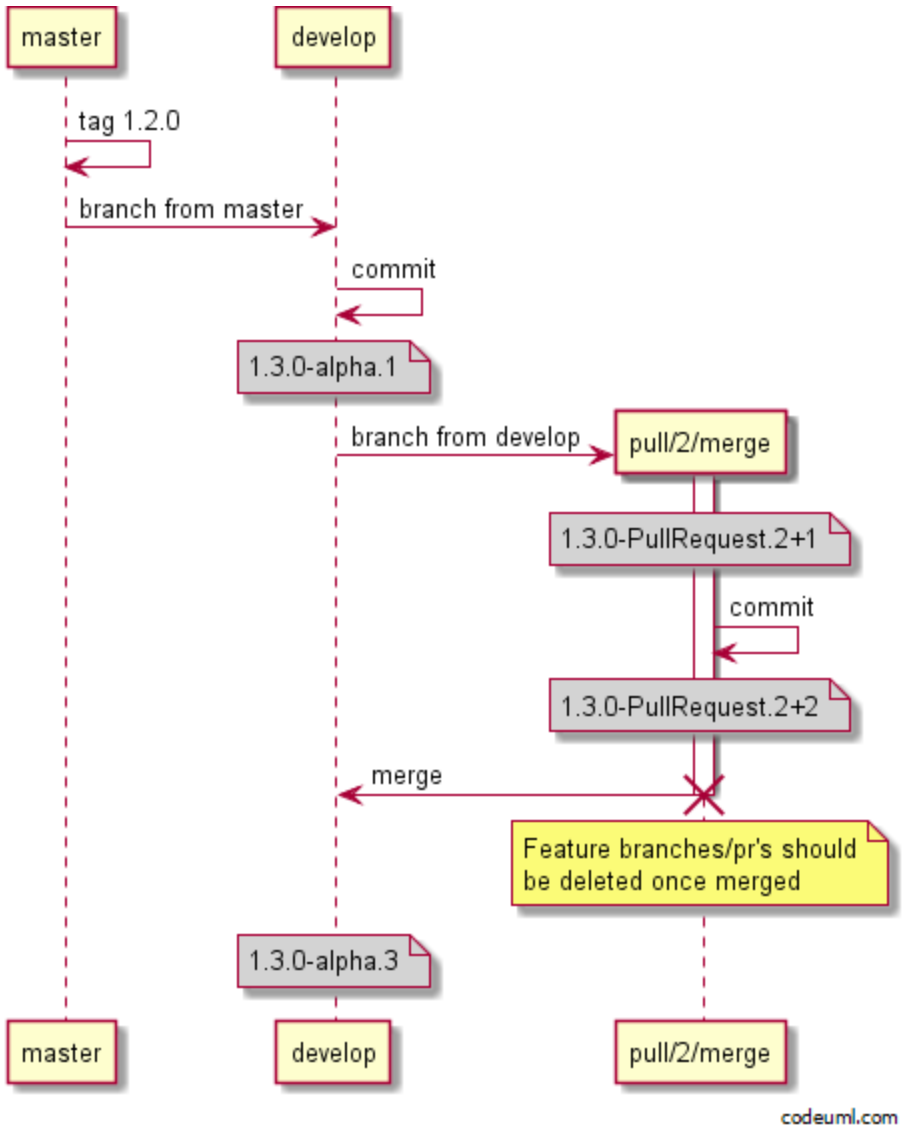
Some key takeaways to know about Gitflow are:

- The workflow is great for a release-based software workflow.
- Gitflow offers a dedicated channel for hotfixes to production.

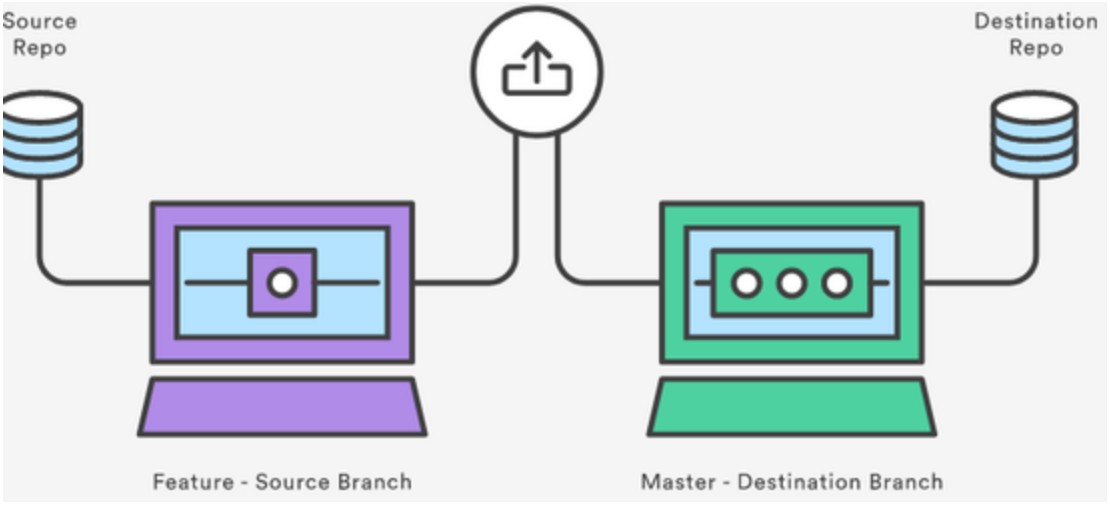
The overall flow of Gitflow is:

1. A `develop` branch is created from `master`
2. A `release` branch is created from `develop`
3. `Feature` branches are created from `develop`
4. When a `feature` is complete it is merged into the `develop` branch
5. When the `release` branch is done it is merged into `develop` and `master`
6. If an issue in `master` is detected a `hotfix` branch is created from `master`
7. Once the `hotfix` is complete it is merged to both `develop` and `master`

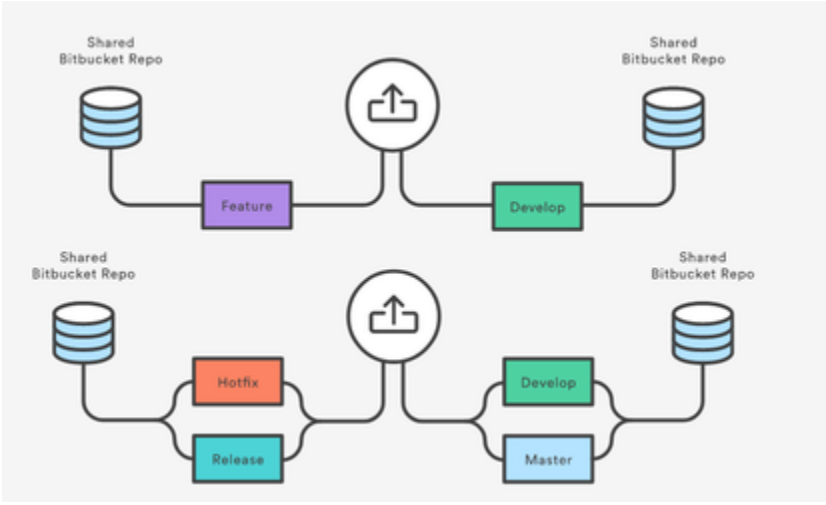
2. PULL REQUEST



1. In their simplest form, pull requests are a mechanism for a developer to notify team members that they have completed a feature.
2. But, the pull request is more than just a notification—it's a dedicated forum for discussing the proposed feature. If there are any problems with the changes, teammates can post feedback in the pull request and even tweak the feature by pushing follow-up commits. All of this activity is tracked directly inside of the pull request.
3. When you file a pull request, all you're doing is *requesting* that another developer (e.g., the project maintainer) *pulls* a branch from your repository into their repository. This means that you need to provide 4 pieces of information to file a pull request: the source repository, the source branch, the destination repository, and the destination branch.



4. The above diagram shows a pull request that asks to merge a feature branch into the official master branch, but there are many other ways to use pull requests.
5. The general process for PR is as follows:
    a. A developer creates the feature in a dedicated branch in their local repo.
    b. The developer pushes the branch to repository.
    c. The developer files a pull request .
    d. The rest of the team reviews the code, discusses it, and alters it.
    e. The project maintainer merges the feature into the official repository and closes the pull request.
6. Adding pull requests to the Gitflow Workflow gives developers a convenient place to talk about a release branch or a maintenance branch while they're working on it.



7. The mechanics of pull requests in the Gitflow Workflow are:a developer simply files a pull request when a feature, release, or hotfix branch needs to be reviewed, and the rest of the team will be notified.
8. Features are generally merged into the `develop` branch, while release and hotfix branches are merged into both `develop` and `master`. Pull requests can be used to formally manage all of these merges.

3. Versioning

The different types of branches we may use are:

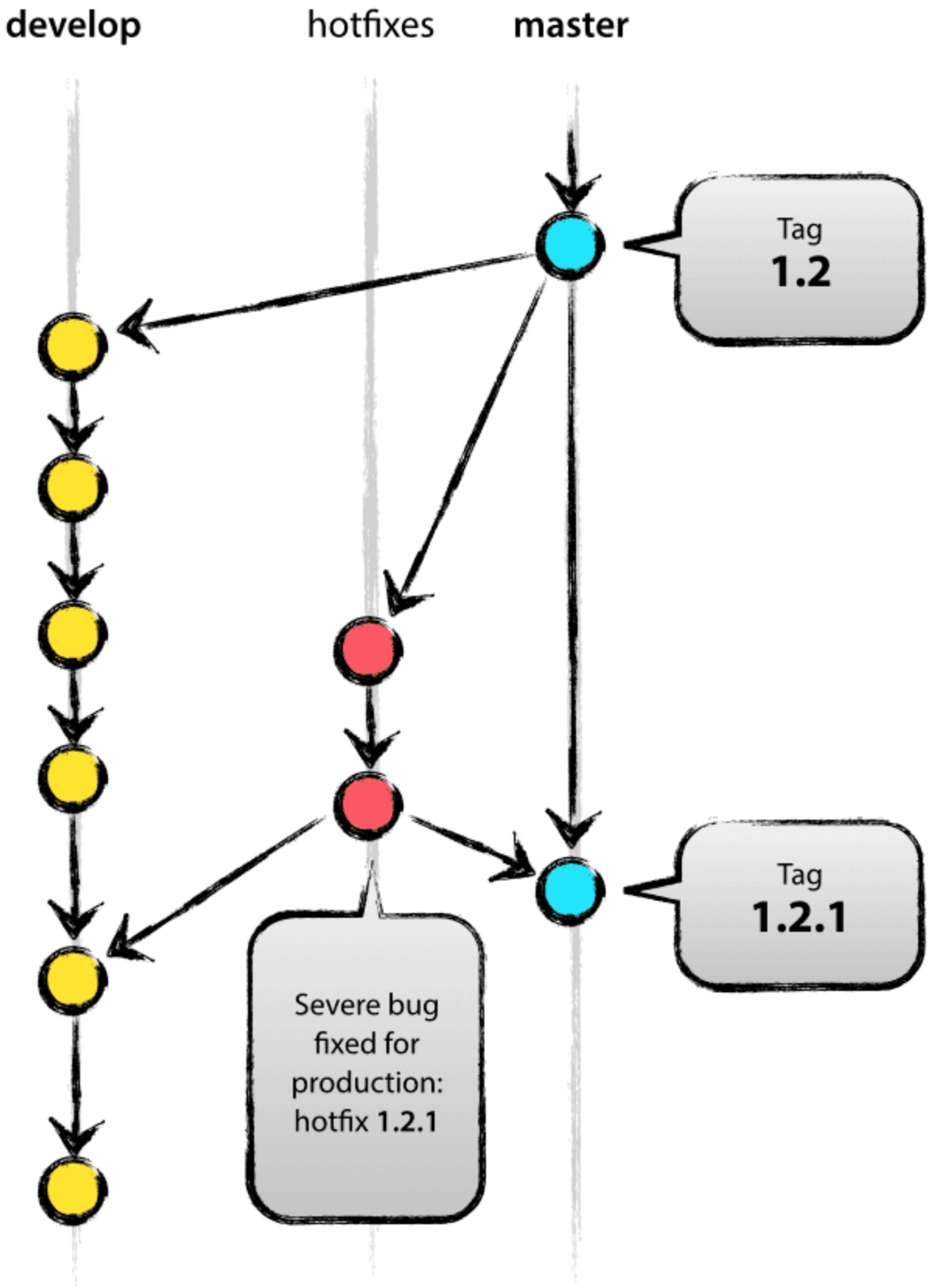
- `master`
    - production releases
- `develop`
    - AKA integration branch
- Release branches (`release-*`)
    - for finalizing a major/minor release, branched from `develop`
- Hotfix branches (`hotfix-*`)
    - for applying patches, branched from master (or support-* for older releases)
- Support branches (`support-*`)
    - for applying patches to old release versions, branched from `master`
- Feature branches (`feature-*`)
    - for developing features or wild speculation, branched from `develop`

1. **master branch:** Initially this branch will be manually tagged with the first pre-release version say 0.0.0. Branches should only be merged onto this branch if they are complete and a production candidate. On branch build the CI system will:
    - compile and run tests, terminating on failure
    - find the previous latest version tag on the branch, lets call this *x.y.z*
    - tag the commit with the tag *x.(y+1).0* (i.e. the first version will be 0.1.0)
2. **feature branches:** Feature branches (or sometimes called topic branches) are used to develop new features for the upcoming or a distant future release.When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point. The essence of a feature branch is that it exists as long as the feature is in development, but will eventually be merged back into `develop` (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment).Feature branches typically exist in developer repos only, not in `origin`.
3. **hotfix branches:** These could be branches prefixed 'hotfix-'. Where a problem is discovered with a historical version x.y.0 and a decision has been made to patch that version as opposed to upgrade to the latest version a branch will be created from the version tagged x.y.0 and match the hotfix naming convention (e.g. 'hotfix-crash-on-empty-friend-list'). On branch build the CI system will:
    - compile and run tests, terminating on failure
    - find the previous latest version tag on the branch, lets call this *x.y.z*
    - tag the commit with the tag *x.y.(z+1)*
4. **release branches:**
    a. Release branches support preparation of a new production release. They allow for last-minute dotting of i's and crossing t's. Furthermore, they allow for minor bug fixes and preparing meta-data for a release (version number, build dates, etc.). By doing all of this work on a release branch, the `develop` branch is cleared to receive features for the next big release
    b. .Release branches are created from the `develop` branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of `develop` is ready for the "next release" and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0).
    c. This new branch may exist there for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the `develop` branch). Adding large new features here is strictly prohibited. They must be merged into `develop`, and therefore, wait for the next big release.
5. **Support branch:** It is essential if you need to maintain multiple major versions at the same time. You could use support branches for supporting minor releases as well. If you are just supporting the majors, then name your branch `support-<major>.x`(i.e`support-1.x`), to support minors use `support-<major>.<minor>.x` or `support-<major>.<minor>.0`. (i.e`support-1.3.x` or `support-1.3.0`)

    If `master` has moved on a point release (1.0, 1.1, 2.0, etc) and a hotfix must be applied to a older version (e.g 1.x):
    - create a `support-1.x` branch (if none exists) based on the newest 1.x tag in `master`
    - create a branch (e.g. `hotfix-1.1.1`), based on `support-1.x`
    - fix the bug and merge `hotfix-1.1.1` back into `support-1.x`
    - Do this for other older major releases as well

    The support branch effectively becomes a master branch for a past version.

4. Best Practices for Hotfix



1. Hotfix branches are created from the `master` branch.
2. When finished, the bugfix needs to be merged back into `master`
3. It also needs to be merged back into `develop`, in order to safeguard that the bugfix is included in the next release as well.
4. The one exception to the rule here is that, **when a release branch currently exists, the hotfix changes need to be merged into that**

release branch, instead of `develop`.
5. Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into `develop` too, when the release branch is finished.