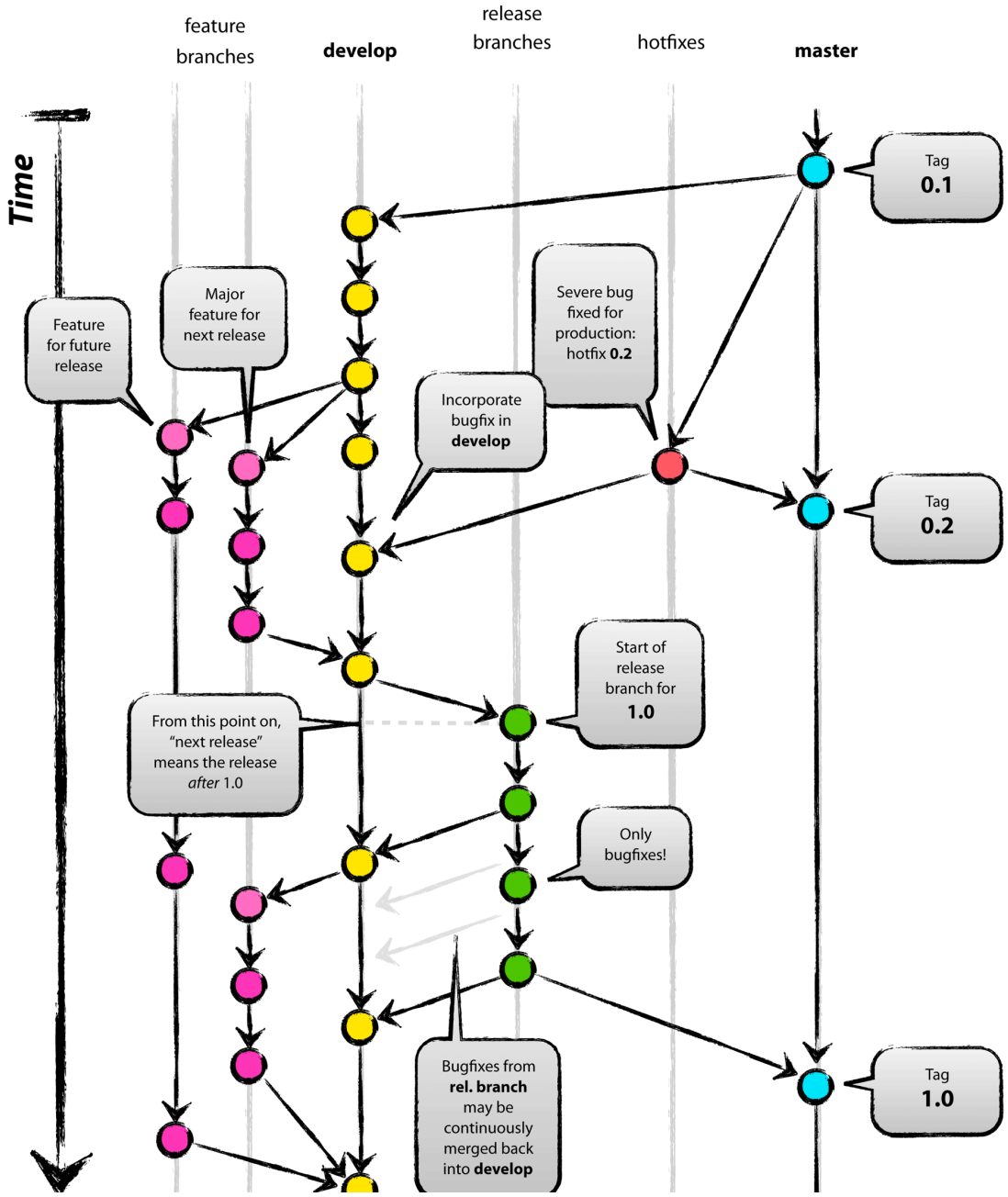
Git Flow Workflow

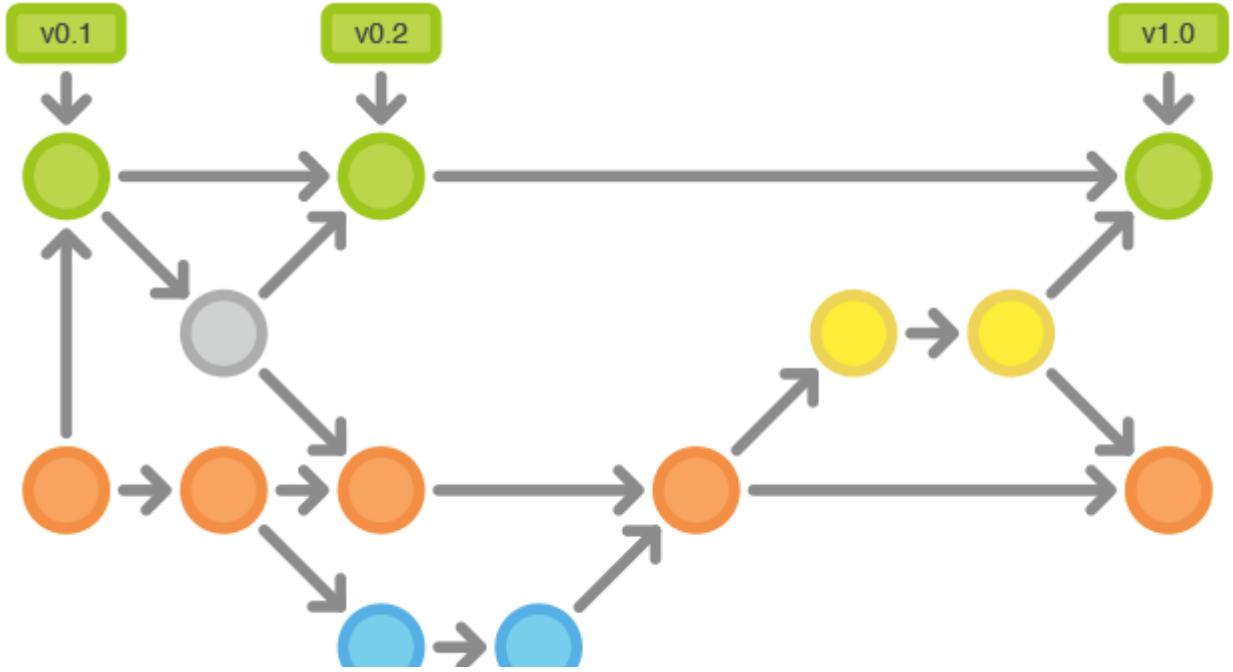

feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after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

This Git Flow mashup is derived from Vincent Driessen's A Successful Git Branching Model and Atlassian's Gitflow Workflow.

Credit is due to them. I've simply mashed up the two sources, edited some wording, and merged the two `git` command styles into one unified flow.

The pattern illustrated here uses `git` without the `git flow` module add-on; so just pure `git` commands.

How Git Flow Works



The Git Flow workflow uses a central repository as the communication hub for all developers. Developers work locally and push branches to the central repo.

Historical Branches

Instead of a single `master` branch, this workflow uses two branches to record the history of the project. The `master` branch stores the official release history, and the `develop` branch serves as an integration branch for features. It's also convenient to tag all commits in the `master` branch with a version number.

The rest of this workflow revolves around the distinction between these two branches.

Feature Branches

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. When a feature is complete, it gets merged back into `develop`. Features should never interact directly with `master`.

Best Practices:

> May branch off: `develop`
>
> Must merge back into: `develop`
>
> Branch naming convention: anything except `master`, `develop`, `release-*`, or `hotfix-*`

Release Branches

Once `develop` has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of `develop`.

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.

Once it's ready to ship, the release gets merged into `master` and tagged with a version number. In addition, it should be merged back into `develop`, which may have progressed since the release was initiated.

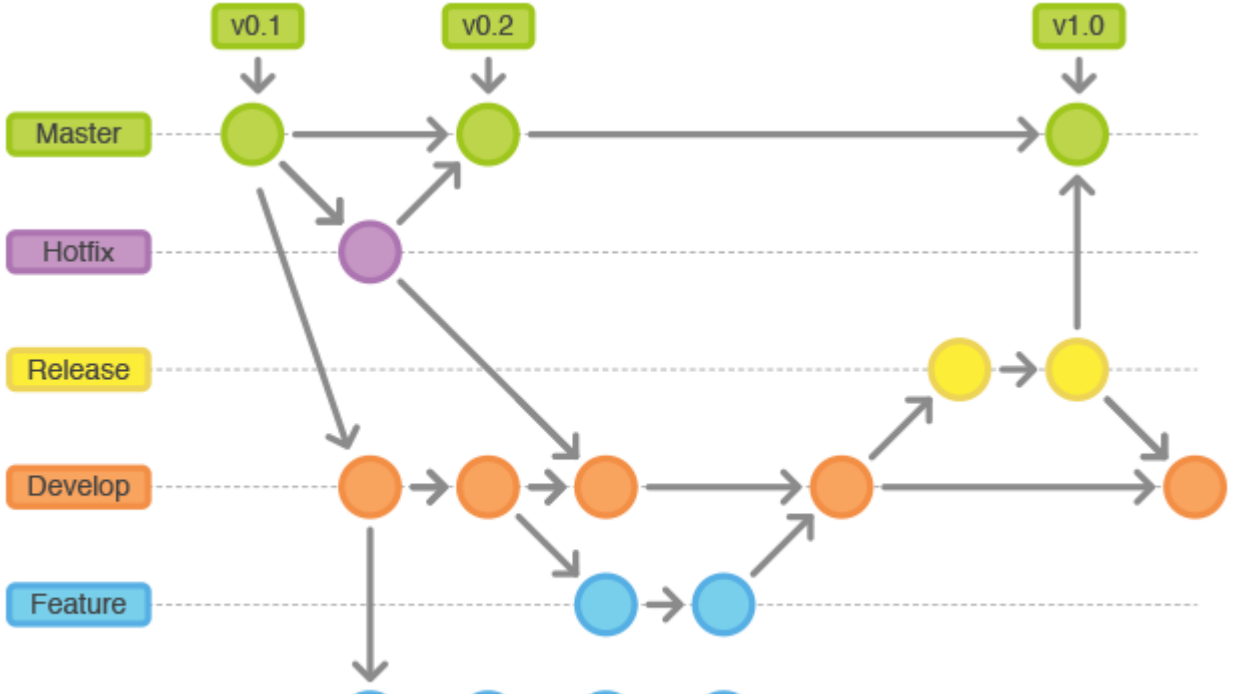
Using a dedicated branch to prepare releases makes it possible for one team to polish the current release while another team continues working on features for the next release. It also

for version 4.0” and to actually see it in the structure of the repository).

Best Practices:

- May branch off: `develop`
- Must merge back into: `develop` and `master`
- Tag: increment `major` or `minor` number
- Branch naming convention: `release-*` or `release/*`

Maintenance Branches



Maintenance or “hotfix” branches are used to quickly patch production releases. 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.

Having a dedicated line of development for bug fixes lets your team address issues without interrupting the rest of the workflow or waiting for the next release cycle. You can think of maintenance branches as ad hoc release branches that work directly with `master`.

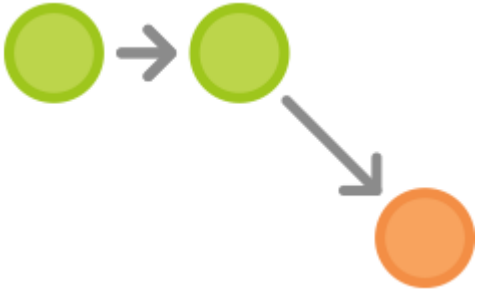
Best Practices:

> May branch off: `master`
>
> Must merge back into: `master` and `develop`
>
> Tag: increment `patch` number
>
> Branch naming convention: `hotfix-*` or `hotfix/*`

Git Flow Example

The example below demonstrates how this workflow can be used to manage a single release cycle. We’ll assume you have already created a central repository.

Create A Develop Branch



The first step is to complement the default `master` with a `develop` branch. A simple way to do this is for one developer to create an empty `develop` branch locally and push it to the server:

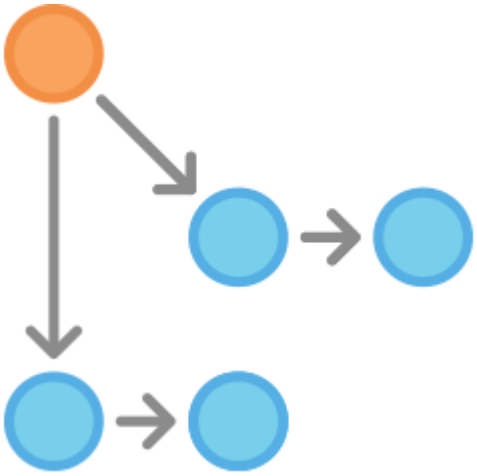
```
git branch develop
git push -u origin develop
```

This branch will contain the complete history of the project, whereas `master` will contain an abridged version. Other developers should now clone the central repository and create a tracking branch for `develop`:

```
git clone ssh://user@host/path/to/repo.git
git checkout -b develop origin/develop
```

Everybody now has a local copy of the historical branches set up.

Mary And John Begin New Features



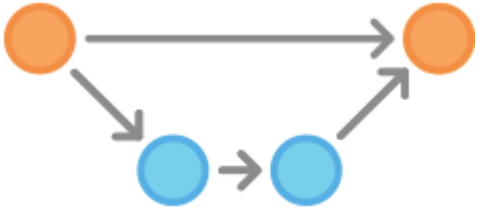
Our example starts with John and Mary working on separate features. They both need to create separate branches for their respective features. Instead of basing it on `master`, they should both base their feature branches on `develop`:

```
git checkout -b some-feature develop

# Optionally, push branch to origin:
git push -u origin some-feature
```

Both of them add commits to the feature branch in the usual fashion: edit, stage, commit:

```
git status
git add some-file
git commit
```

Mary Finishes Her Feature

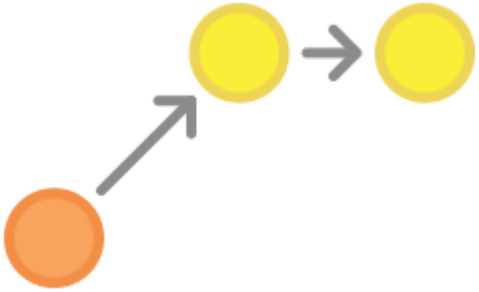


After adding a few commits, Mary decides her feature is ready. If her team is using pull requests, this would be an appropriate time to open one asking to merge her feature into `develop`. Otherwise, she can merge it into her local `develop` and push it to the central repository, like so:

```
git pull origin develop
git checkout develop
git merge --no-ff some-feature
git push origin develop

git branch -d some-feature

# If you pushed branch to origin:
git push origin --delete some-feature
```

The first command makes sure the `develop` branch is up to date before trying to merge in the feature. Note that features should never be merged directly into `master`.

Mary Begins To Prepare A Release



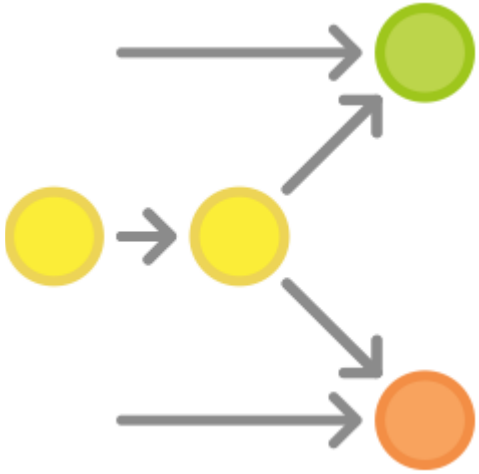
While John is still working on his feature, Mary starts to prepare the first official release of the project. She uses a new branch to encapsulate the release preparations. This step is also where the release's version number is established, and she uses the SemVer initial release recommendation of `v0.1.0`:

```
git checkout -b release-0.1.0 develop

# Optional: Bump version number, commit
# Prepare release, commit
```

This branch is a place to clean up the release, test everything, update the documentation, and do any other kind of preparation for the upcoming release. It's like a `feature` branch dedicated to polishing the release.

As soon as Mary creates this branch and pushes it to the central repository, the release is **feature-frozen**. Any functionality that isn't already in `develop` is postponed until the next release cycle.

Mary Finishes The Release



Once the release is ready to ship, Mary merges it into `master` and `develop`, then deletes the `release` branch. 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. Again, if Mary's organization stresses code review, this would be an ideal place for a pull request.

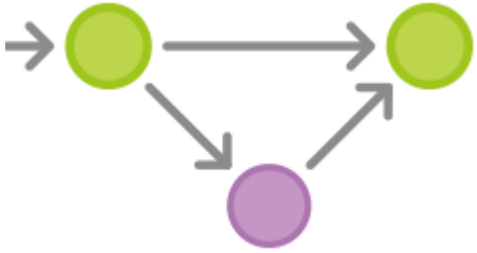
```
git checkout master
git merge --no-ff release-0.1.0
git push

git checkout develop
git merge --no-ff release-0.1.0
git push

git branch -d release-0.1.0

# If you pushed branch to origin:
git push origin --delete release-0.1.0
```

commit for easy reference:

```
git tag -a v0.1.0 master
git push --tags
```

> **Pro Tip**: Git comes with several hooks, which are scripts that execute whenever a particular event occurs within a repository. It's possible to configure a hook to automatically build a public release whenever you push the `master` branch to the central repository or push a tag.

End-user Discovers A Bug



After shipping the release, Mary goes back to developing features for the next release with John. That is, until an end-user opens a ticket complaining about a bug in the current release. To address the bug, Mary (or John) creates a maintenance branch off of `master`, fixes the issue with as many commits as necessary, then merges it directly back into `master`.

```
git checkout -b hotfix-0.1.1 master
```

```
git checkout master
git merge --no-ff hotfix-0.1.1
git push
```

Like release branches, maintenance branches contain important updates that need to be included in `develop`, so Mary needs to perform that merge as well. Then, she's free to delete the branch:

```
git checkout develop
git merge --no-ff hotfix-0.1.1
git push

git branch -d hotfix-0.1.1
```

Just like in the `release` branch, Mary has merged into `master`, so she needs to tag the commit on the `master` branch. By incrementing the `patch` number, she indicates this is a non-breaking maintenance release:

```
git tag -a v0.1.1 master
git push --tags
```

Now go forth and enjoy the "flow" in your own Git repos!