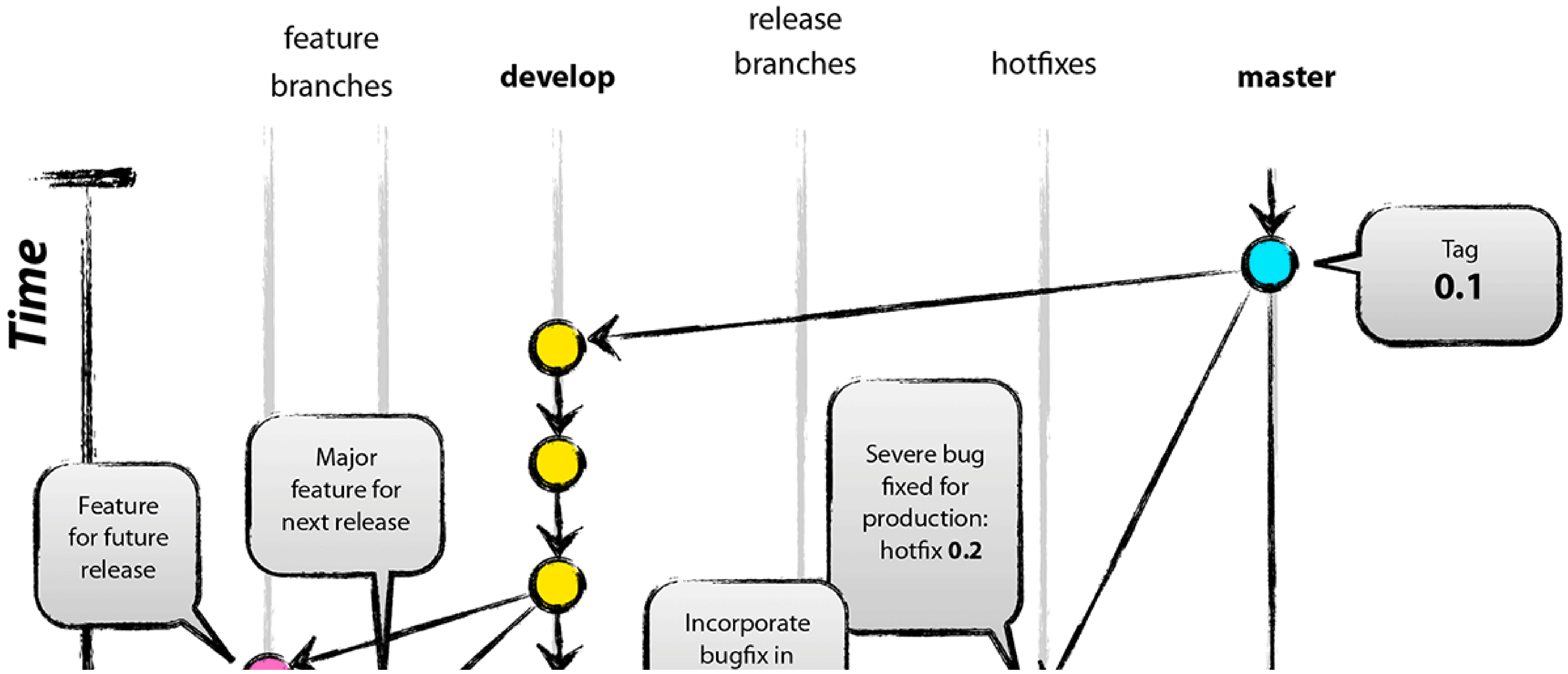
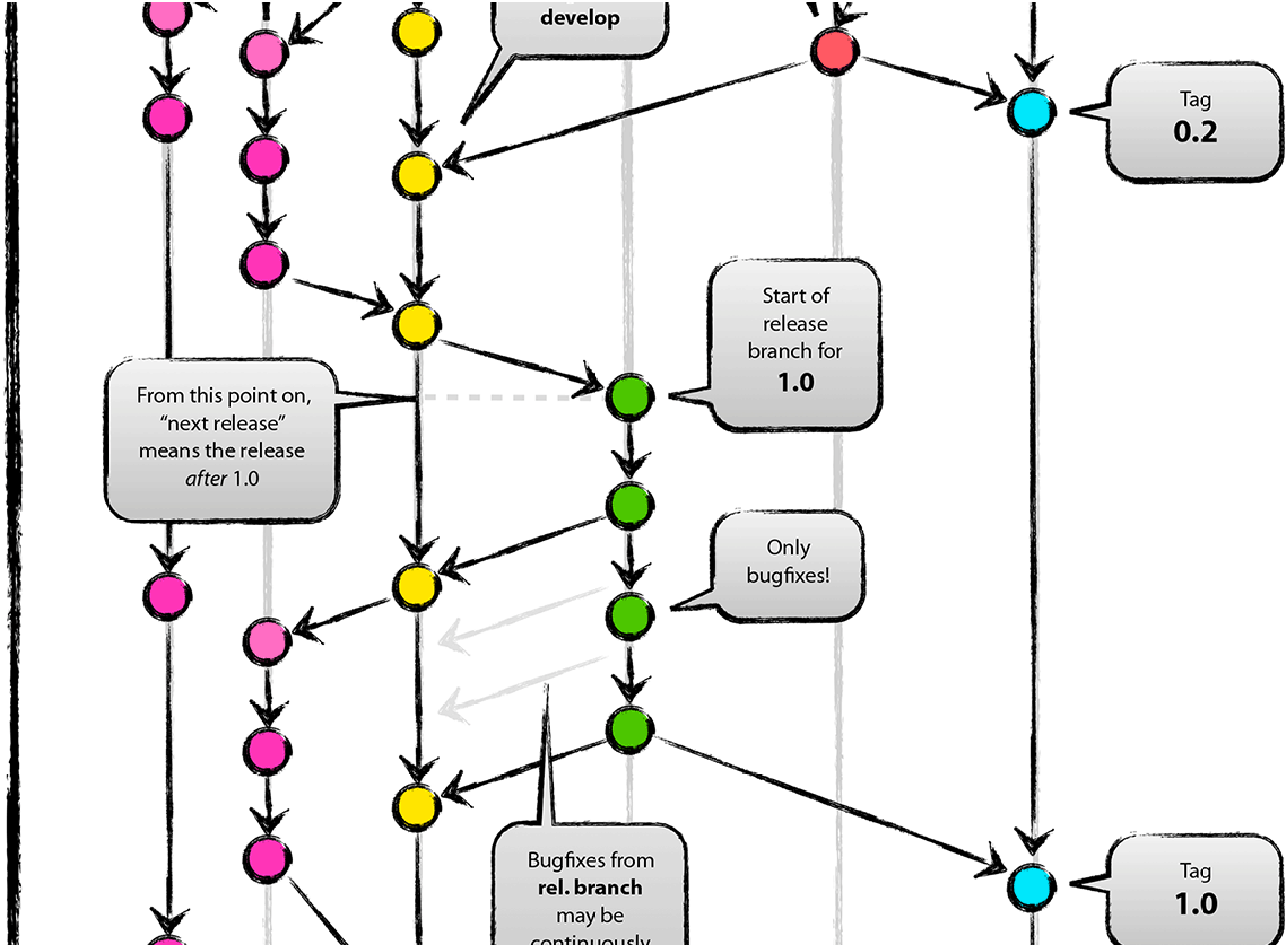
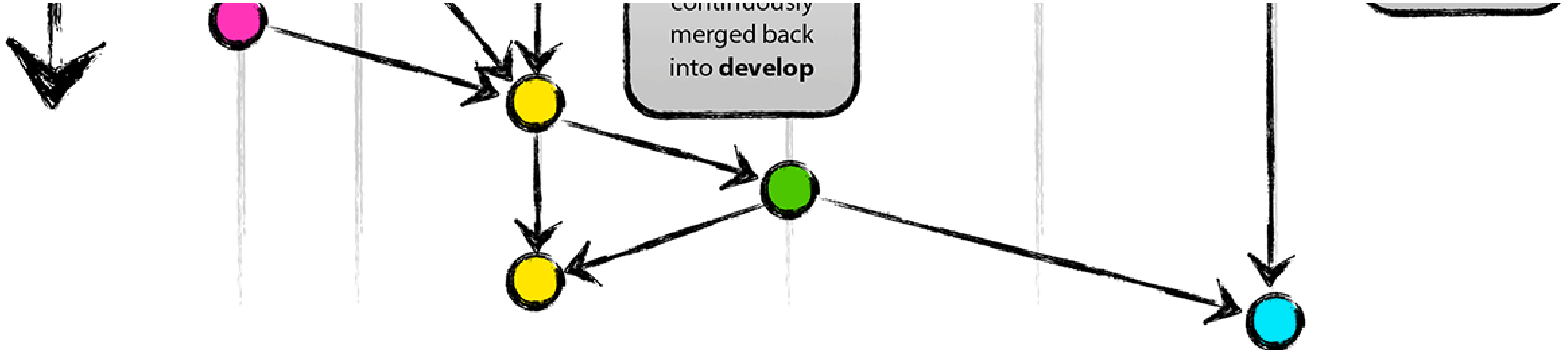
# Using git-flow to automate your git branching workflow

2010-08-19 (last updated on 2017-01-07)

Vincent Driessen's branching model is a git branching and release management strategy that helps developers keep track of features, hotfixes and releases in bigger software projects. This workflow has lot of commands to type and remember, though, so there's also the git-flow library of git subcommands that helps automate some parts of the flow to make working with it a lot easier.


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
Tag 1.0



After installing git-flow (brew install git-flow), you can start using git-flow in your repository by using it's init command. You can use it in existing projects, but let's start a new repository:

```
$ git flow init
Initialized empty Git repository in ~/project/.git/
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
```

git-flow is just a wrapper around existing git commands, so the init command doesn't change anything in your repository other than creating branches for you. If you don't want to use git-flow anymore, there's nothing to change or remove, you just stop using the git-flow commands.

If you run git branch after setting up, you'll notice that you switched from the master branch to a new one named develop.

```
$ git branch
* develop
  master
```

The develop branch is intended to be the default branch where most of the work will happen, and the master branch keeps track of production-ready code. So instead of working in the master branch, you'll use git push origin develop to push new code to your repository from now on.

## Feature branches

git-flow makes it easy to work on multiple features at the same time by using feature branches. To start one, use feature start with the name of your new feature (in this case, "authentication"):

```
$ git flow feature start authentication
Switched to a new branch 'feature/authentication'

Summary of actions:
- A new branch 'feature/authentication' was created, based on 'develop'
- You are now on branch 'feature/authentication'

Now, start committing on your feature. When done, use:

     git flow feature finish authentication
```

As the output already explains, you’re now on a new branch you can use to work on your feature. Use git like you normally would, and finish the feature using `feature finish` when it’s done:

```
$ git flow feature finish authentication
Switched to branch 'develop'
Updating 9060376..00bafe4
Fast-forward
 authentication.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 authentication.txt
Deleted branch feature/authentication (was 00bafe4).

Summary of actions:
- The feature branch 'feature/authentication' was merged into 'develop'
- Feature branch 'feature/authentication' has been removed
- You are now on branch 'develop'
```

Your feature branch will be merged and you’re taken back to your `develop` branch. Internally, git-flow used `git merge --no-ff feature/authentication` to make sure you don’t lose any hostorical information about your feature branch before it is removed.

## Versioned releases

If you need tagged and versioned releases, you can use git-flow’s release branches to start a new branch when you’re ready to deploy a new version to production.

Like everything else in git-flow, you don't have to use release branches if you don't want to. Prefer to manually `git merge --no-ff develop` into master without tagging? No problem. However, if you're working on a versioned API or library, release branches might be really useful, and they work exactly like you'd expect:

```
$ git flow release start 0.1.0
Switched to a new branch 'release/0.1.0'

Summary of actions:
- A new branch 'release/0.1.0' was created, based on 'develop'
- You are now on branch 'release/0.1.0'

Follow-up actions:
- Bump the version number now!
- Start committing last-minute fixes in preparing your release
- When done, run:

    git flow release finish '0.1.0'
```

Bump the version number and do everything that's required to release your project in the release branch. I personally wouldn't do any last minute fixes, but if you do, git-flow will make sure everything is correctly merged into both `master` and `develop`. Then, finish the release:

```
$ git flow release finish 0.1.0
Switched to branch 'master'
Merge made by the 'recursive' strategy.
 authentication.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 authentication.txt
Deleted branch release/0.1.0 (was 1b26f7c)
Deleted branch release/0.1.0 (was 1b26f7c).

Summary of actions:
- Latest objects have been fetched from 'origin'
- Release branch has been merged into 'master'
- The release was tagged '0.1.0'
- Release branch has been back-merged into 'develop'
- Release branch 'release/0.1.0' has been deleted
```

Boom. git-flow pulls from origin, merges the release branch into master, tags the release and back-merges everything back into develop before removing the release branch.

You’re still on master, so you can deploy before going back to your `develop` branch, which git-flow made sure to update with the release changes in `master`.

## Hotfixing production code

Because you keep your `master` branch always in sync with the code that’s on production, you’ll be able to quickly fix any issues on production.

For example, if your assets aren’t loading on production, you’d roll back your deploy and start a hotfix branch:

```
$ git flow hotfix start assets
Switched to a new branch 'hotfix/assets'

Summary of actions:
- A new branch 'hotfix/assets' was created, based on 'master'
- You are now on branch 'hotfix/assets'
```

```
Follow-up actions:
- Bump the version number now!
- Start committing your hot fixes
- When done, run:

    git flow hotfix finish 'assets'
```

Hotfix branches are a lot like release branches, except they’re based on master instead of develop. You’re automatically switched to the new hotfix branch so you can start fixing the issue and bumping the minor version number. When you’re done, hotfix finish:

```
 $ git flow hotfix finish assets
Switched to branch 'master'
Merge made by the 'recursive' strategy.
 assets.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 assets.txt

Switched to branch 'develop'
Merge made by the 'recursive' strategy.
 assets.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 assets.txt
Deleted branch hotfix/assets (was 08edb94).

Summary of actions:
- Latest objects have been fetched from 'origin'
- Hotfix branch has been merged into 'master'
- The hotfix was tagged '0.1.1'
```

```
  - Hotfix branch has been back-merged into 'develop'
  - Hotfix branch 'hotfix/assets' has been deleted
```

Like when finishing a release branch, the hotfix branch gets merged into both `master` and `develop`. The release is tagged and the hotfix branch is removed.