Git commit messaging

How to write a Git commit message:

1. Begin the commit message with a short title that summarizes the change.
2. Start the title with an imperative present active verb: `Add`, `Drop`, `Fix`, `Refactor`, `Optimize`, etc.
3. Format the title: start with a capital word, use up to `50 characters`, and end without a period.
4. Optionally follow the title by a blank line, then a more thorough description.

Title examples

Title examples of good commit messages:

- Add feature for ...
- Drop feature for ...
- Fix bug when ... is missing
- Start feature flag
- Stop feature flag
- Refactor ... for clarity
- Optimize ... for speed and memory

Keyword Strategy

We will use these keywords because they use the following:

- imperative moods
- present tense
- an active voice
- verbs

Keywords

- `Add` = Create a capability e.g. feature, test, dependency.
- `Drop` = Delete a capability e.g. feature, test, dependency.
- `Fix` = Fix an issue e.g. bug, typo, accident, misstatement.
- `Bump` = Increase the version of something e.g. a dependency.
- `Make` = Change the build process, or tools, or infrastructure.

- `Start` = Begin doing something; e.g. enable a toggle, feature flag, etc.
- `Stop` = End doing something; e.g. disable a toggle, feature flag, etc.
- `Refactor` = A change that MUST be just refactoring.
- `Reformat` = A change that MUST be just formatting, e.g. omit whitespace.
- `Rephrase` = A change that MUST be just textual, e.g. edit a comment, doc, etc.
- `Optimize` = A change that MUST be just about performance, e.g. speed up code.
- `Document` = A change that MUST be only in the documentation, e.g. help files.

Semantic versioning

We use semantic versioning for many of our projects:

- `Add` = Increment *SemVer* MINOR version.
- `Drop` = Increment *SemVer* MAJOR version.
- `Fix`, `Refactor`, `Reformat`, `Rephrase`, `Optimize`, etc. = Increment *SemVer* PATCH version.

Rules

Capitalize the title.

- Right: Add feature
- ~~Wrong: add feature~~

Do not end the title with a period:

- Right: Add feature
- ~~Wrong: Add feature.~~

Use imperative mood: present tense, active voice, and lead verb.

- Right: Add feature
- ~~Wrong: Adds feature (this is indicative mood, not imperative mood)~~
- ~~Wrong: Added feature (this is past tense, not present tense)~~
- ~~Wrong: Adding feature (this lead is a gerund, not a verb)~~
- ~~Wrong: Feature added (this is passive voice, not active voice)~~

It is best practice to keep the title within `50 characters`.

It is best practice use a blank line after the title.

It is best practice wrap the body at 72 characters. This is the same convention as writing an

email message.

For more about these see How to Write a Git Commit Message

Purpose and Reasoning

We primarily care that our team communicates effectively with an aligned understanding. The verbs above are helpful because they're they're easy to read, easy to type, align with best practice and are clear in many cultures.

Will will reject the following formats:

We will reject some kinds of git commit message formats using our linting commit hook

- [bug] ...
- (release) ...
- #12345 ...
- jira:// ...
- docs: ...

We reject the commit style of projets such as Angular, Commitizen defaults, etc.

- Because these use a leading tag that is sometimes a word, sometimes an abbreviation, sometimes a plural noun, etc.
- Examples are using "feat" for feature, "docs" for document, "perf" for performance improvement, etc.
- Instead we use "Add" for adding a feature, "Document" for documenting help, "Optimize" for performance improvement, etc.
- Active verbs are easier to skim, and easier to use for people from other cultures who may be less-comfortable using English.

We reject using an id number or URL in the title.

- We use fully-qualified URLs in the commit message body.
- This is because many of our projects use multiple tracking systems, and multiple ways of launching a URL.
- We want URL tracking to be easy to use by a wide range of systems, scripts, and teams.

Use task tracking links

We will connect a git commit to a task tracking system or web page that explains more.

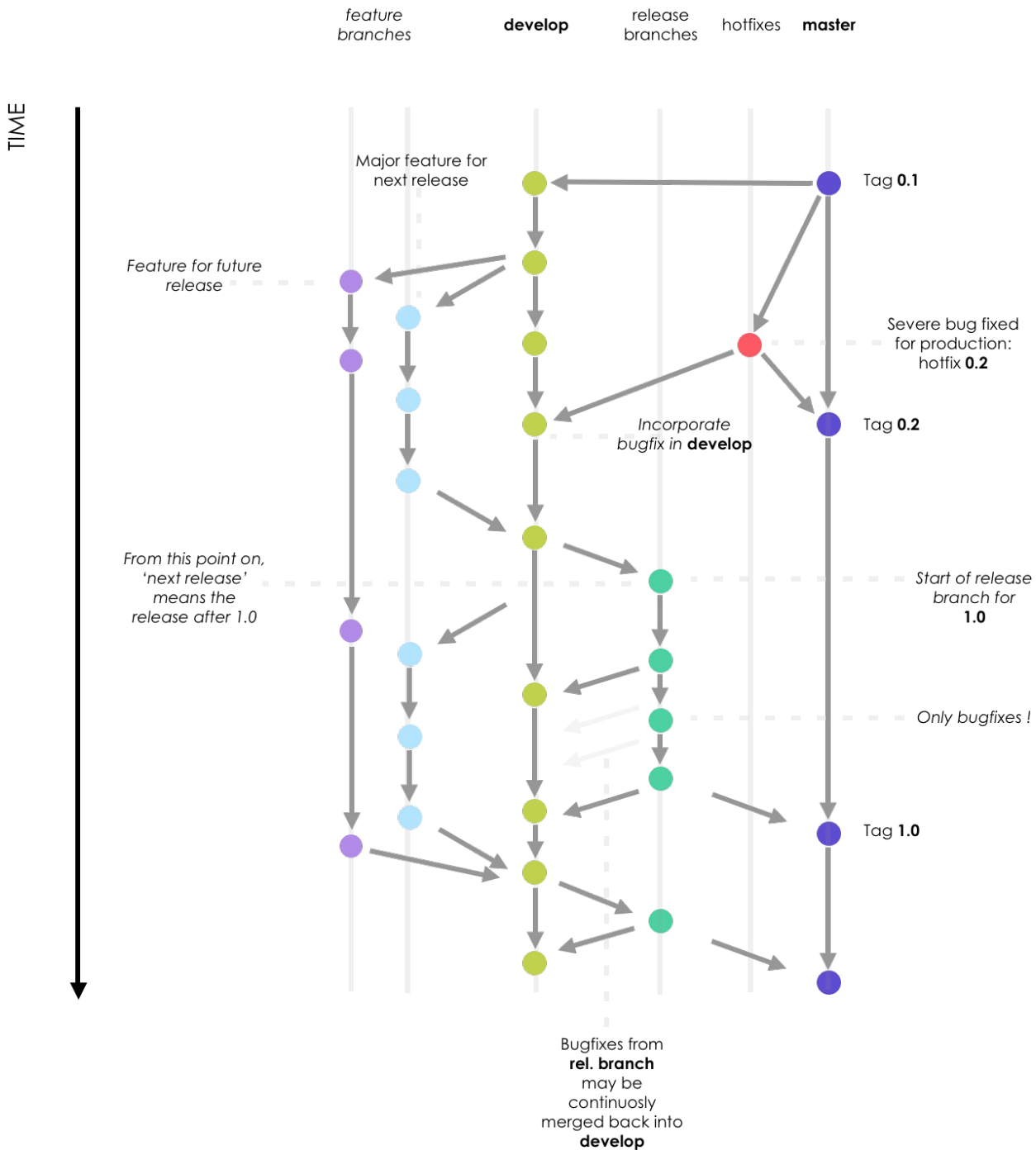
For example, we use Pivotal Tracker, Invision and Confluence. To keep track of these, we will write git commit body messages that lists each URL, one per line because this is easy to parse.

Example:

```
Add feature foo

See: [Request for help with sign in](https://github.com/user/repo/issues/789)
See: [Add feature ...](https://jira.com/tasks/123)
See: [Invision ....](https://wikipedia/quicksort)
```

The Big Picture

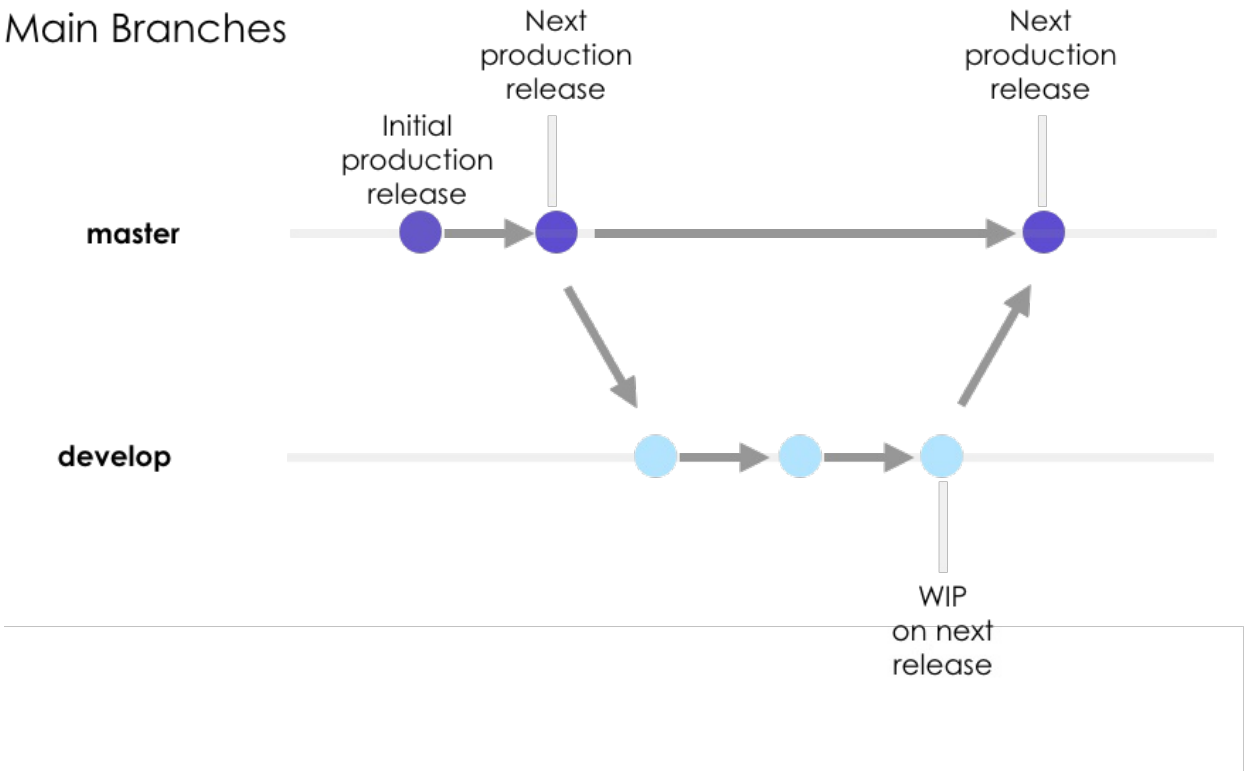


The Main Branches

At the core, the development model is greatly inspired by existing models out there. The central repo holds two main branches with an infinite lifetime:

- `master` at origin(should be familiar to every Git user)
- `develop` is parallel to the master branch.

We consider `origin/master` to be the main branch where the source code of *HEAD* always reflects a "production-ready" state.



We consider `origin/develop` to be the main branch where the source code of *HEAD* always reflects a state with the latest delivered development changes for the next release.

Some would call this the “integration branch”. This is where any automatic nightly builds are built from.

When the source code in the develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number.

How this is done in detail will be discussed further on.

Therefore, each time when changes are merged back into master, this is a new production release by definition.

Some would call this the `release candidate`

We have to be very strict with this, so that theoretically, our pipeline scripts will automatically build and roll-out our software to our production servers every time there was a commit on master.

Supporting branches

Analogous to the main branches `master` and `develop`, our development should use a variety of supporting branches to act as support for the following business use cases.

- Parallel development between team members
- Ease tracking of features
- Prepare for production releases and to assist in quickly fixing live production problems.
- Unlike the main branches, these branches always have a limited life time, since they will be removed eventually.

The different types of branches we use are:

- `Feature`
- `Release`
- `Hotfix`

Each of these branches have a specific purpose that are bound to strict rules. These rules establish which branches may be their *origin* branch and which branches must be their *merge targets*.

The branch types are categorized by how we use them and are not “special” from a technical perspective.

Feature branches

Feature Branch	May branch off from	Must merge back into
155803215-feature	develop	develop

`155803215-feature` is an example of a branch name

May branch off from:

`develop`

Must merge back into:

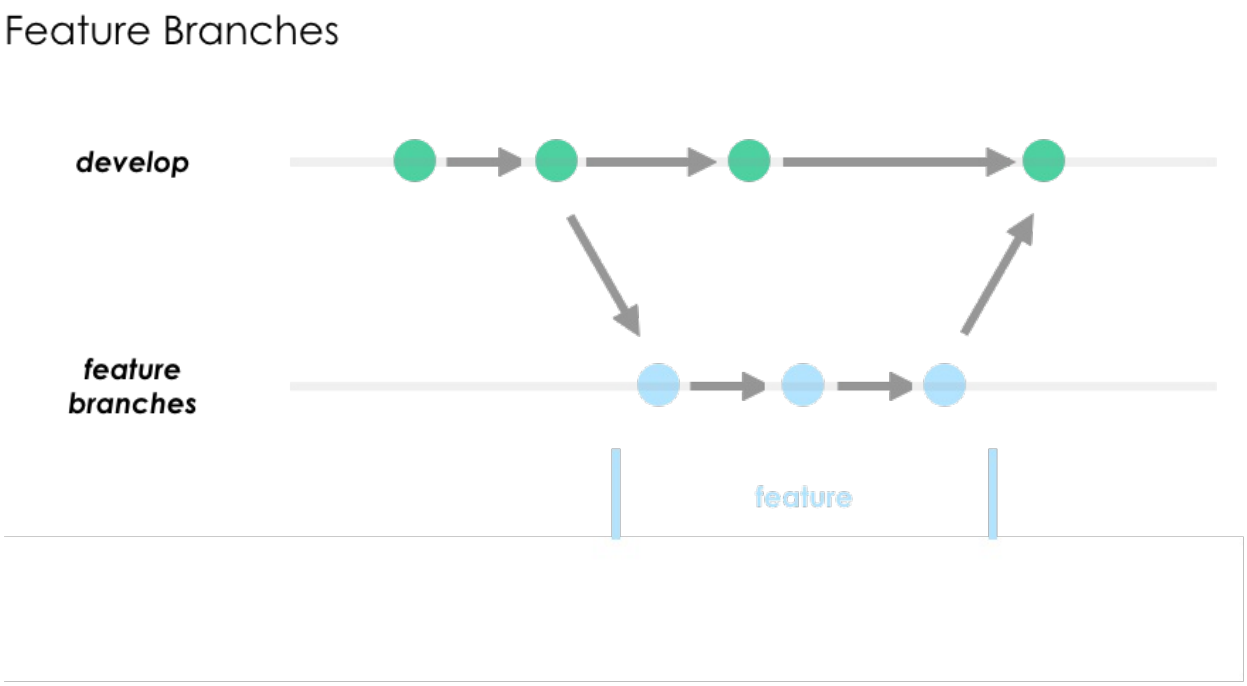
`develop`

Branch naming convention rules are anything except the following

- master
- develop
- release-*
- hotfix-*

The purpose of a feature branch

Feature branches are used to develop new features for the upcoming or a distant future release.
When starting development of a feature, the target release for the feature may be unknown at its creation time.
The purpose 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 add the new feature to the upcoming release.
The feature branch could also be discarded in case of the failure of an experiment.
Feature branches typically exist in developer repos only, not in origin.
Since this is not practiced today, this is a part of the paradigm that team will need to adjust to.



Creating a feature branch

When starting work on a new feature, branch off from the develop branch.

```
$ git checkout -b <feature-branch-convention> develop
Switched to a new branch <feature-branch-convention>
```

Feature Branch Convention

`<STORYID-TYPE-SHORT_STATEMENT>`

Incorporating a finished feature on develop

Finished features are to be merged into the `develop` branch so that they are added to the upcoming release:

```
$ git checkout develop
```

`...Switched to branch 'develop'`

```
$ git merge --no-ff myfeature
```

`...Updating ea1b82a..05e9557,`
`(Summary of changes)`

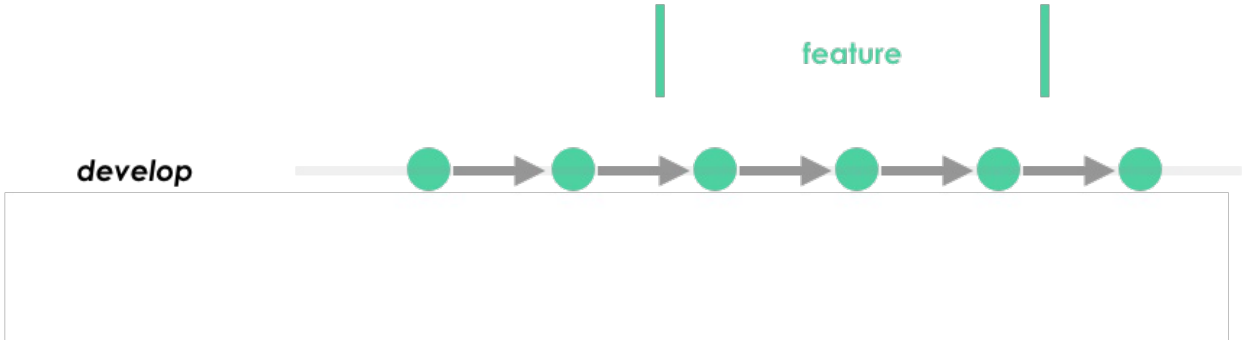
```
$ git branch -d myfeature
```

`...Deleted branch myfeature (was 05e9557)`

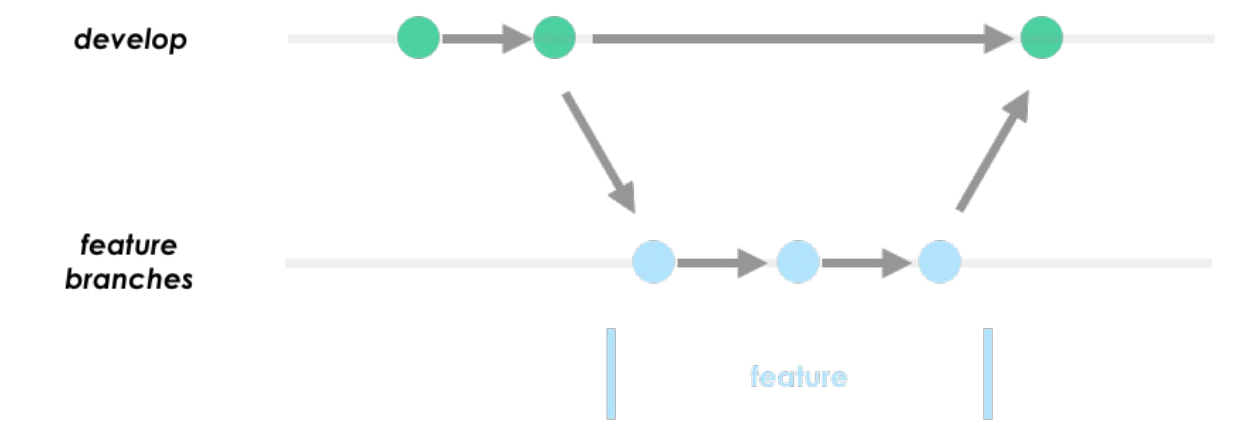
```
$ git push origin develop
```

The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature. Compare:

Merge Without —no-ff



Merge With —no-ff



In the latter case, it is impossible to see from the Git history which of the commit objects together have implemented a feature—you would have to manually read all the log messages. Reverting a whole feature (i.e. a group of commits), is a true headache in the latter situation, whereas it is easily done if the --no-ff flag was used.

Yes, it will create a few more (empty) commit objects, but the gain is much bigger than the cost.

Release branches

May branch off from:
`develop`

Must merge back into:
`develop and master`
Branch naming convention:
`release-*`

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.

The key moment to branch off a new release branch from develop is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to develop at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the develop branch reflected changes for the "next release", but it is unclear whether that "next release" will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project's rules on version number bumping.

Creating a release branch

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). So we branch off and give the release branch a name reflecting the new version number:

```
git checkout -b release-1.2 develop
```

`Switched to a new branch "release-1.2`

```
./bump-version.sh 1.2
```

`Files modified successfully, version bumped to 1.2.`

```
git commit -a -m "Bumped version number to 1.2"
```

```
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

After creating a new branch and switching to it, we bump the version number. Here, bump-version.sh is a fictional shell script that changes some files in the working copy to reflect the new version. (This can of course be a manual change—the point being that some files change.) Then, the bumped version number is committed.

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.

Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into master (since every commit on master is a new release by definition, remember). Next, that commit on master must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into develop, so that future releases also contain these bug fixes.

The first two steps in Git:

```
git checkout master
```

```
Switched to branch 'master'
```

```
git merge --no-ff release-1.2
```

```
Merge made by recursive.
(Summary of changes)
```

```
git tag -a 1.2
```

The release is now done, and tagged for future reference.

You might want to use the -s or -u flags to sign your tag cryptographically.

To keep the changes made in the release branch, we need to merge those back into develop, though. In Git:

```
git checkout develop
```

```
Switched to branch 'develop'
```

```
git merge --no-ff release-1.2
```

```
Merge made by recursive. (Summary of changes)
```

This step may well lead to a merge conflict (probably even, since we have changed the version number). If so, fix it and commit.

Now we are really done and the release branch may be removed, since we don't need it anymore:

```
$ git branch -d release-1.2
```

```
Deleted branch release-1.2 (was ff452fe).
```

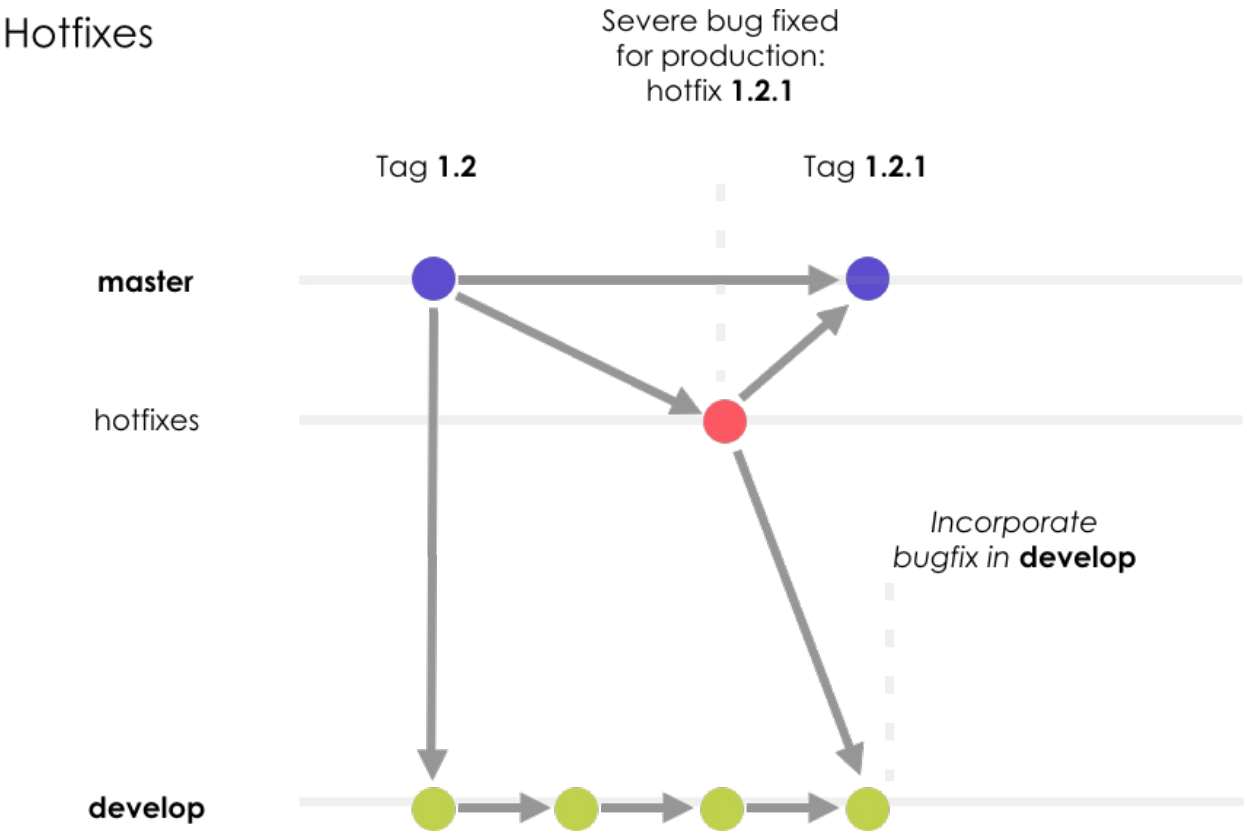
Hotfix branches

May branch off from:
`master`
Must merge back into:
`develop and master`
Branch naming convention:
`hotfix-*`

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

The essence is that work of team members (on the develop branch) can continue, while

another person is preparing a quick production fix.



Creating the hotfix branch

Hotfix branches are created from the master branch. For example, say version 1.2 is the current production release running live and causing troubles due to a severe bug. But changes on develop are yet unstable. We may then branch off a hotfix branch and start fixing the problem:

```
git checkout -b hotfix-1.2.1 master
```

```
Switched to a new branch "hotfix-1.2.1"
```

```
./bump-version.sh 1.2.1
```

```
Files modified successfully, version bumped to 1.2.1.
```

```
git commit -a -m "Bumped version number to 1.2.1"
```

```
[hotfix-1.2.1 41e61bb] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
```

Don't forget to bump the version number after branching off!
Then, fix the bug and commit the fix in one or more separate commits.

```
git commit -m "Fixed severe production problem"
```

```
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
```

```
5 files changed, 32 insertions(+), 17 deletions(-)
```

Finishing a hotfix branch

When finished, the bugfix needs to be merged back into master, but also needs to be merged back into develop, in order to safeguard that the bugfix is included in the next release as well. This is completely similar to how release branches are finished.

First, update master and tag the release.

```
git checkout master
```

```
Switched to branch 'master'
```

```
git merge --no-ff hotfix-1.2.1
```

```
Merge made by recursive.
```

```
(Summary of changes)
```

```
git tag -a 1.2.1
```

```
You might as well want to use the -s or -u <key> flags to sign your tag
cryptographically.
```

Next, include the bugfix in develop, too:

```
git checkout develop
```

```
Switched to branch 'develop'
```

```
git merge --no-ff hotfix-1.2.1
```

```
Merge made by recursive. (Summary of changes)
```

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`. 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. (If work in develop immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into develop now already as well.)

Finally, remove the temporary branch:

```
$ git branch -d hotfix-1.2.1
```

```
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

Summary

Whether we are all familiar or not, this branching model and strategy is nothing new. The main thing to take from this is that this project's purpose is to provide an elegant mental model that is easy to comprehend and allows team members to develop with a shared understanding of the branching and releasing processes.