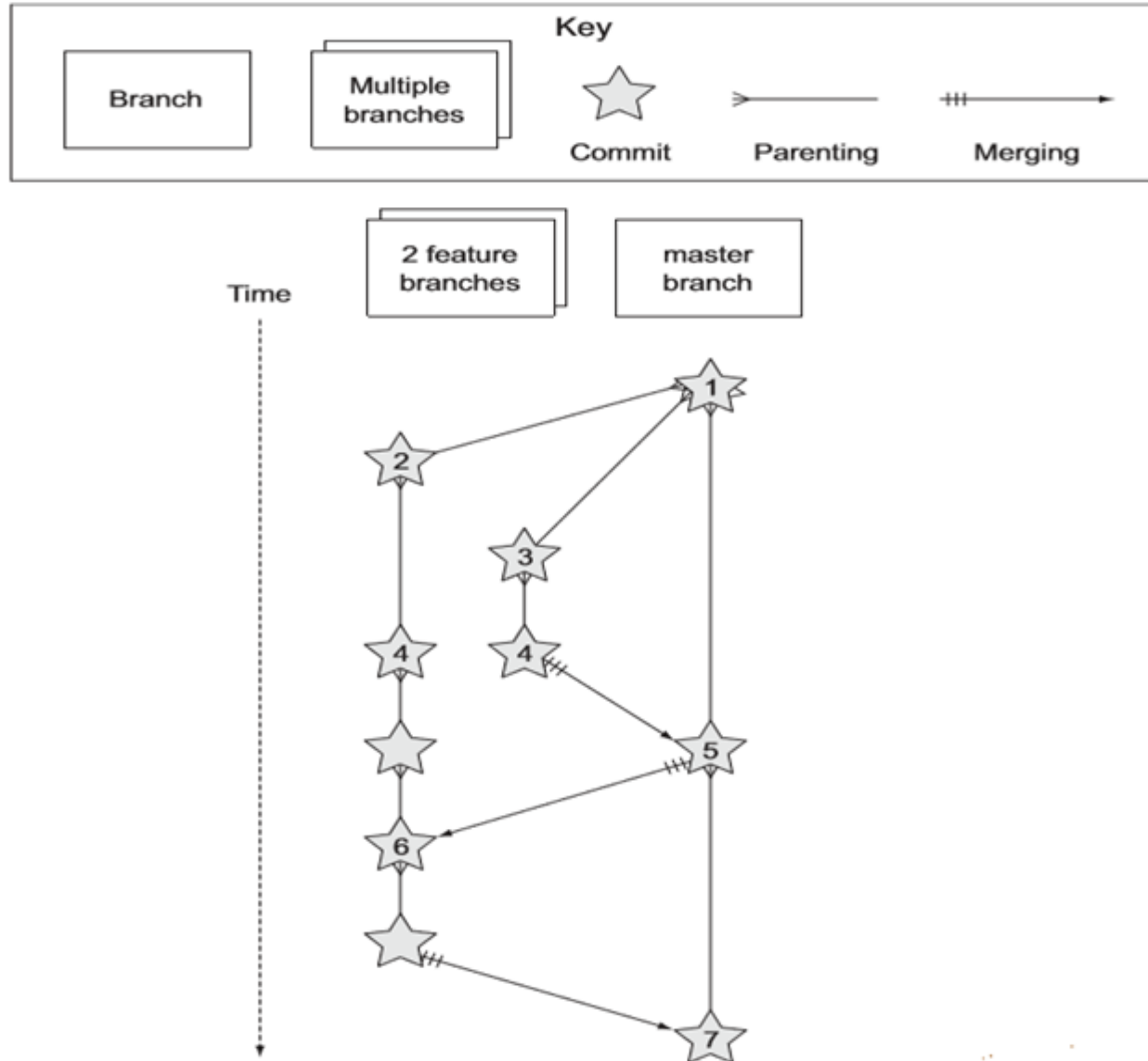


GitHub Flow



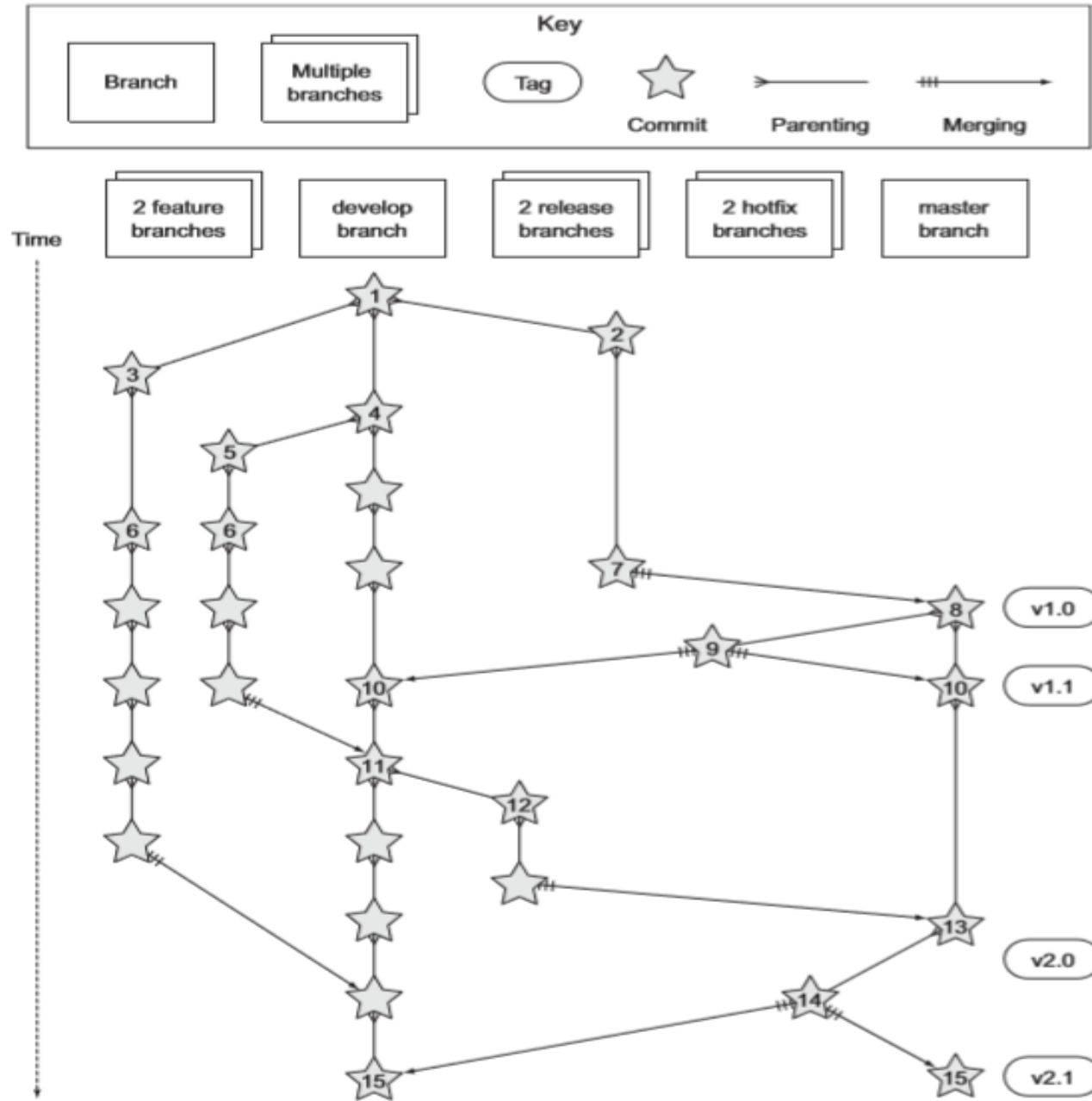
The pros of GitHub Flow are as follows:

- It's beautiful in its simplicity. As a result, it's easy to use with Git, GitHub's web interface, and any Git graphical tools that support branching and merging.
- Because everything ends up in the master branch, there's little concern about commits getting lost.
- A branch is created, committed to, reviewed in a pull request, committed to again if necessary, merged, and deleted. If a branch exists, that's because either it hasn't been merged to master or someone forgot to delete it.

The cons of GitHub Flow are as follows:

- As a large web application, GitHub doesn't have versioned releases to customers. Every commit to the master branch is deployed to the production servers as soon as it's made. This approach is known as *continuous deployment*. It works well for web applications but doesn't work so well for things like desktop application software that needs to be released to users and requires them to restart their application to update it. In this case, because hotfixes can't be pushed to master without branching and review, it's important that feature branches be sufficiently tested before merging to master.
- Some pieces of software need to support multiple versions at once—for example, new stable versions of v1.0 and v2.0. GitHub Flow doesn't account for this case at all.

Git Flow



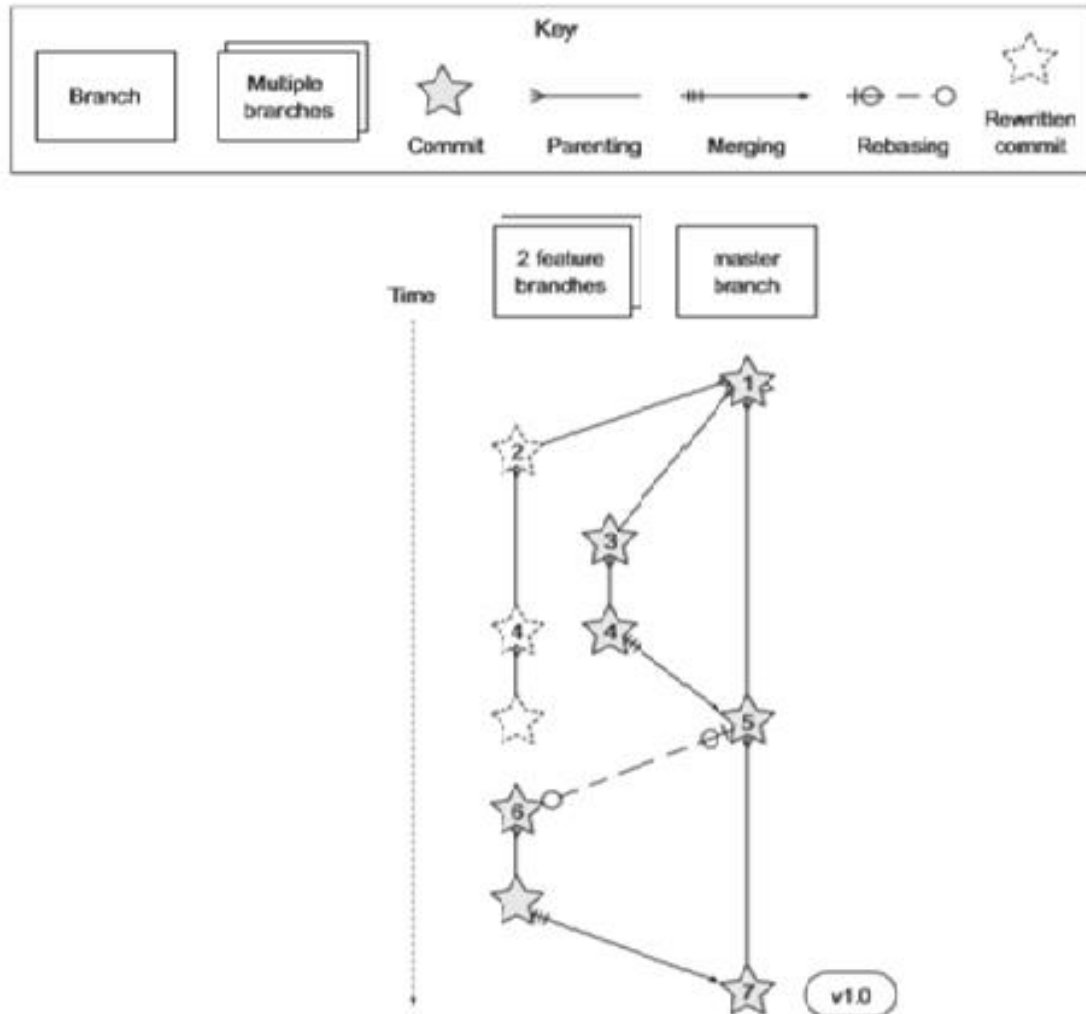
The pros of Git Flow are as follows:

- It allows you to keep track of released versions, features in development, and urgent and non-urgent bug fixes through branch naming.
- Having a formal flow through which branches are merged means a review process can ensure that things are reviewed multiple times before going into a release.

The cons of Git Flow are as follows:

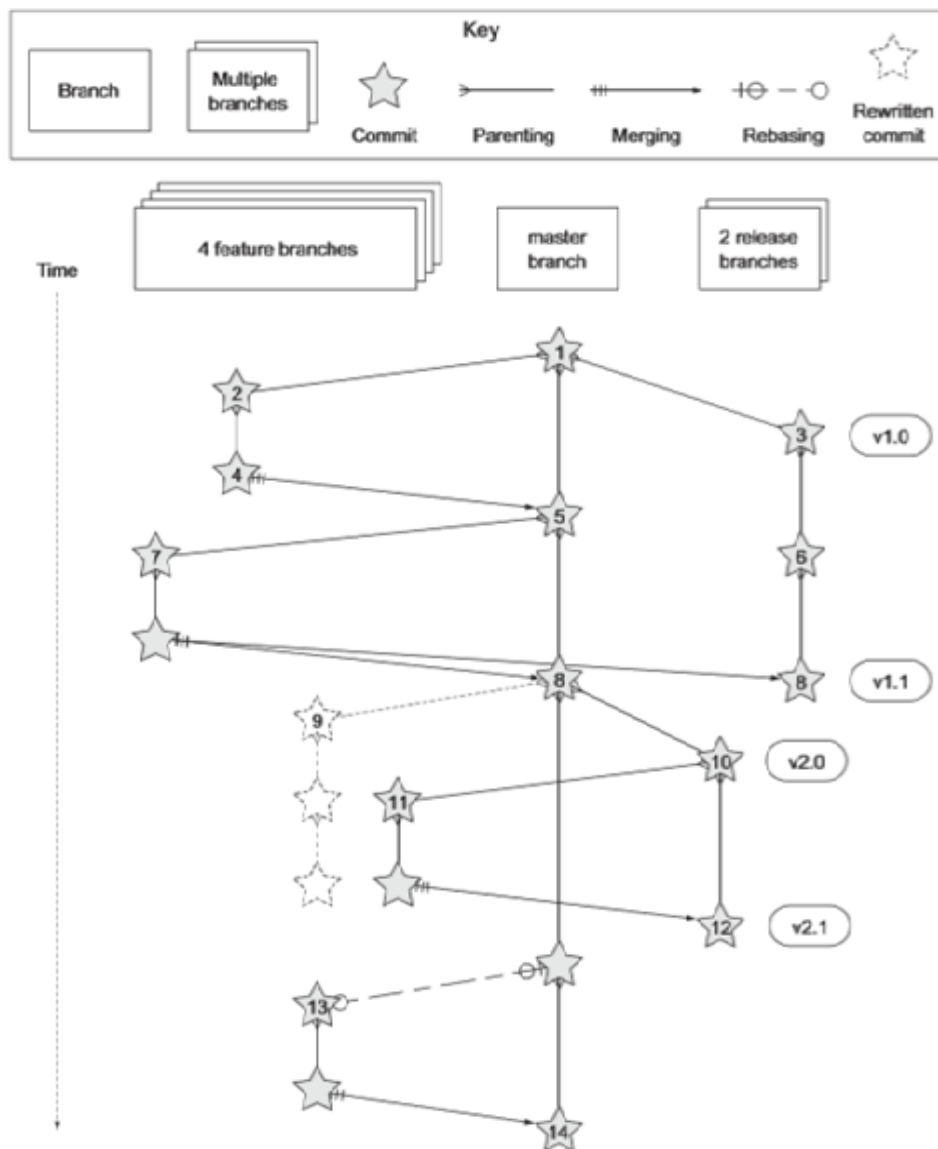
- It's complicated to come to grips with. It can work well for organizations where people can be trained and onboarded, but it's less suitable for short projects or open source projects that seek to attract many new contributors.
- If you're using continuous deployment, the number of merges required from feature branch to master branch can be excessive.

Single Flow



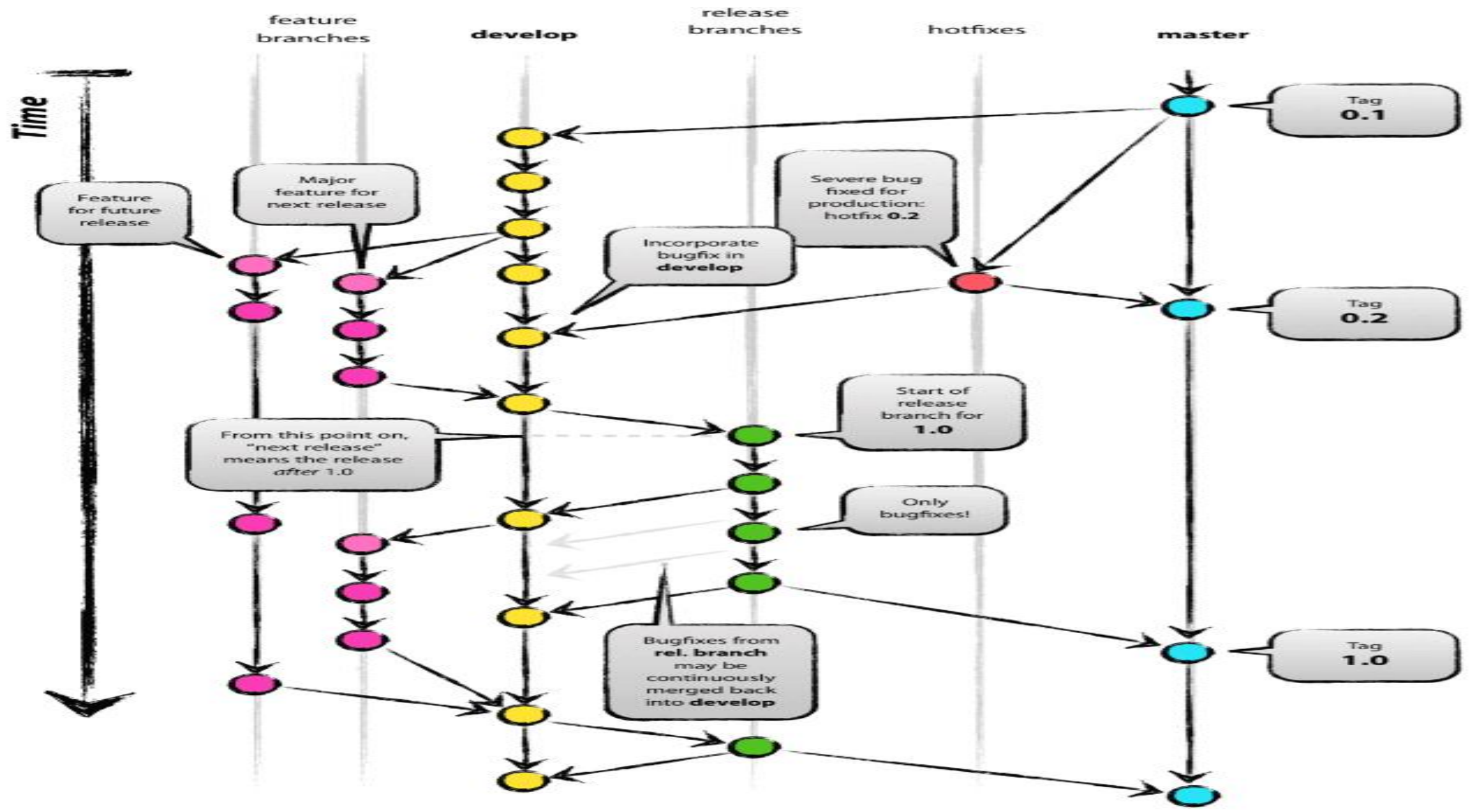
- 1 Branches can be (and should be) rebased, rewritten, and squashed where appropriate (to make history cleaner, but not if the branch is being used by multiple people).
- 2 Stable releases can be tagged on the master branch.

Multiple Flow



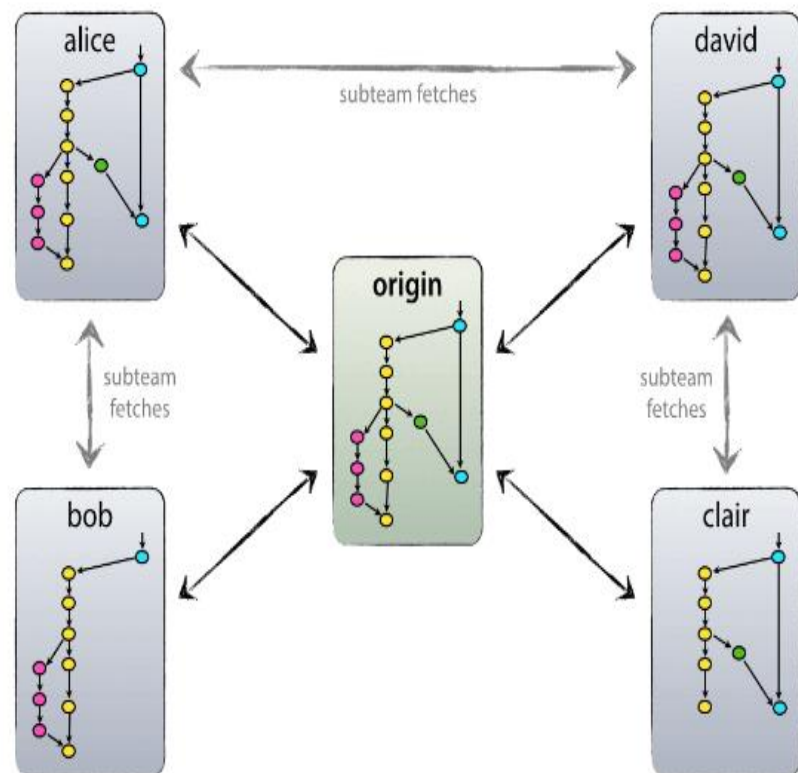
- Any developers not interacting with a release can behave as if they're using GitHub Flow.
- Any developers who are more experienced with Git are empowered by being able to use more advanced history rewriting on remote branches. This allows them to keep their work shared and backed up but still make changes before it's merged.
- Tags and multiple-release branches are optionally added because they're necessary with some forms of software development, such as desktop applications where multiple versions need to be supported.
- Its flexibility in history rewriting and branching may lead to more mistakes.

A successful Git branching model



Decentralized but centralized

The repository setup that we use and that works well with this branching model, is that with a central "truth" repo. Note that this repo is only *considered* to be the central one (since Git is a DVCS, there is no such thing as a central repo at a technical level). We will refer to this repo as `origin`, since this name is familiar to all Git users.



Each developer pulls and pushes to `origin`. But besides the centralized push-pull relationships, each developer may also pull changes from other peers to form sub teams.

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`
- `develop`

The `master` branch at `origin` should be familiar to every Git user. Parallel to the `master` branch, another branch exists called `develop`.

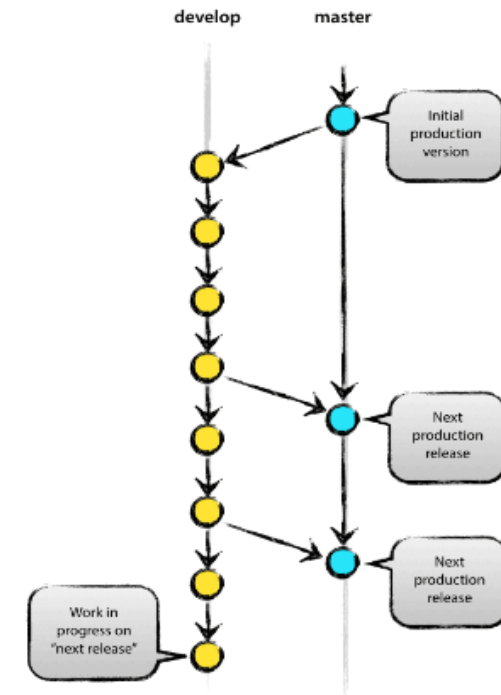
We consider `origin/master` to be the main branch where the source code of HEAD always reflects a *production-ready* state.

Supporting branches

Next to the main branches `master` and `develop`, our development model uses a variety of supporting branches to aid 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 may use are:

- Feature branches
- Release branches
- Hotfix branches



Feature branches

May branch off from:

develop

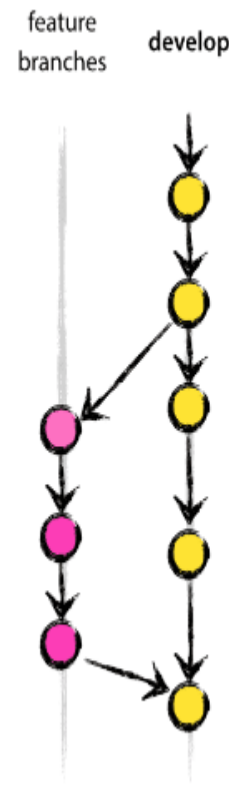
Must merge back into:

develop

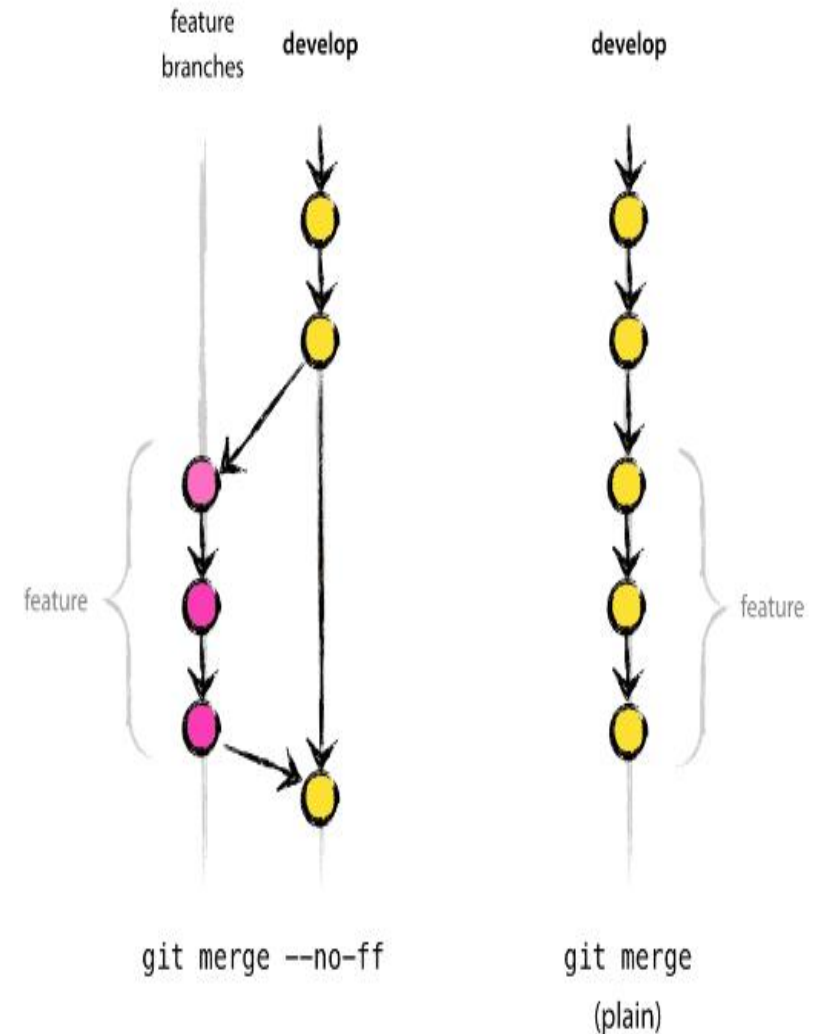
Branch naming convention:

anything except master, develop, release-*, or hotfix-*

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).



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:



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.

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.

