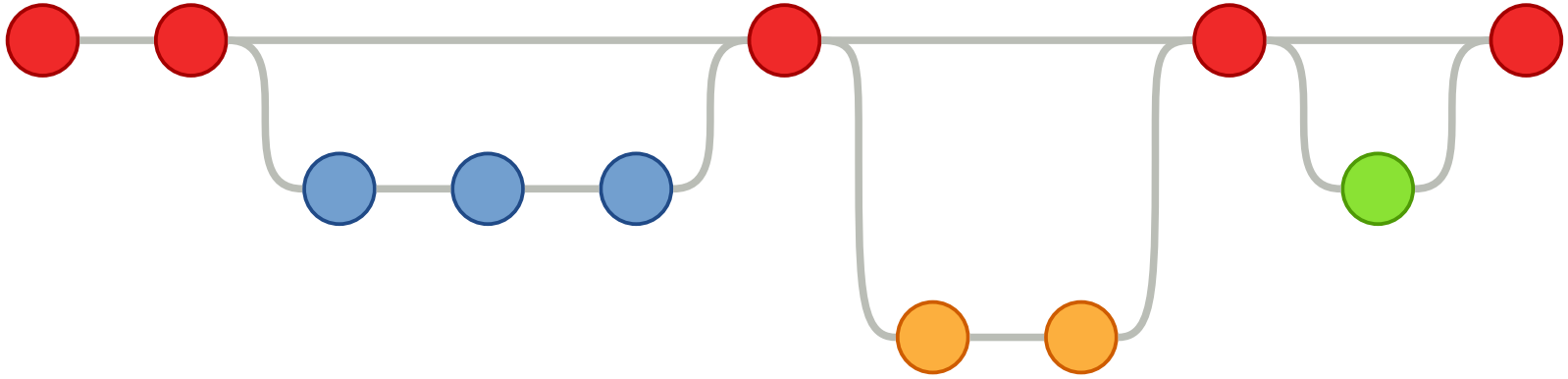
Using git and virtual machines

A more efficient workflow for the Data Science team

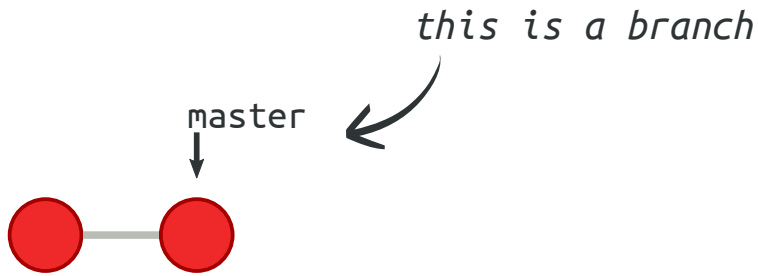


v1.0.0 | 2019.03.04 | Dr Étienne Boisseau-Sierra

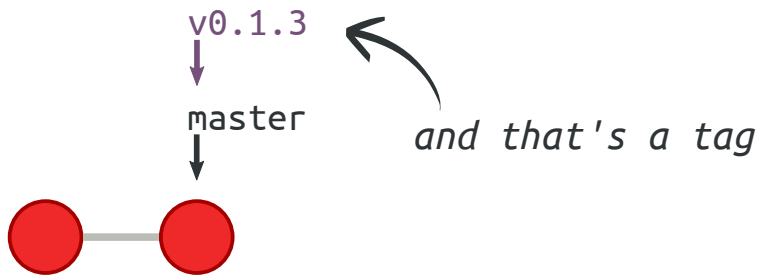
We are using these conventions

We are using these conventions

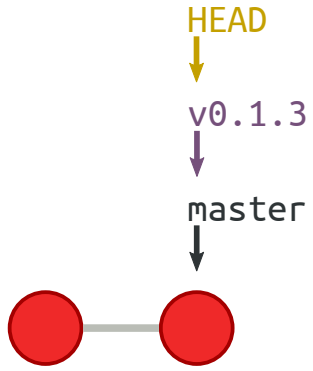
We are using these conventions



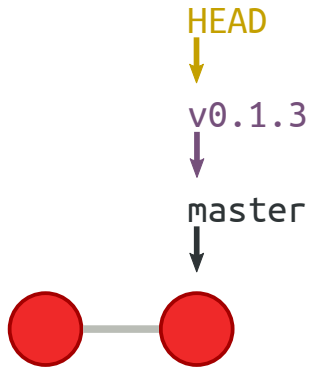
We are using these conventions



We usually won't show HEAD explicitely



HEAD, branches, and tags are just pointers

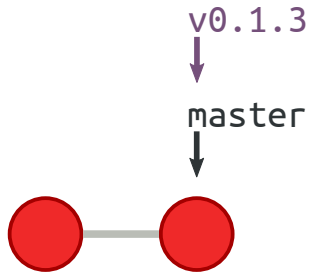


We are using these conventions

(virtual) machines

*production-**

*staging-**

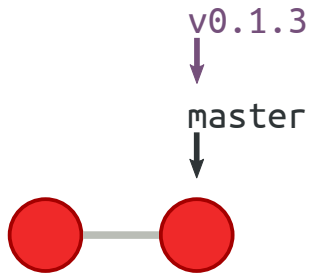


these are (virtual) machines

Two sets of machines: team-managed vs. yours

➔ 'sandbox-*' VM are created on ad hoc basis

(see wiki to know how to spin-up your own sandbox VM)



(virtual) machines

team
- *production-**
- *staging-**

personnal
- *sandbox-alice*
- *-bob*
- *-**
- *local*

remote machine (on the cloud)

local machine (on your computer)

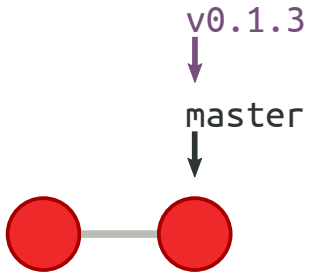
See e.g. git.unipart.io/unipart/douglas/wiki/Setting+up+Dash+on+OpenStack+VMs

We are using these conventions

(virtual) machines

team
- *production-**
- *staging-**

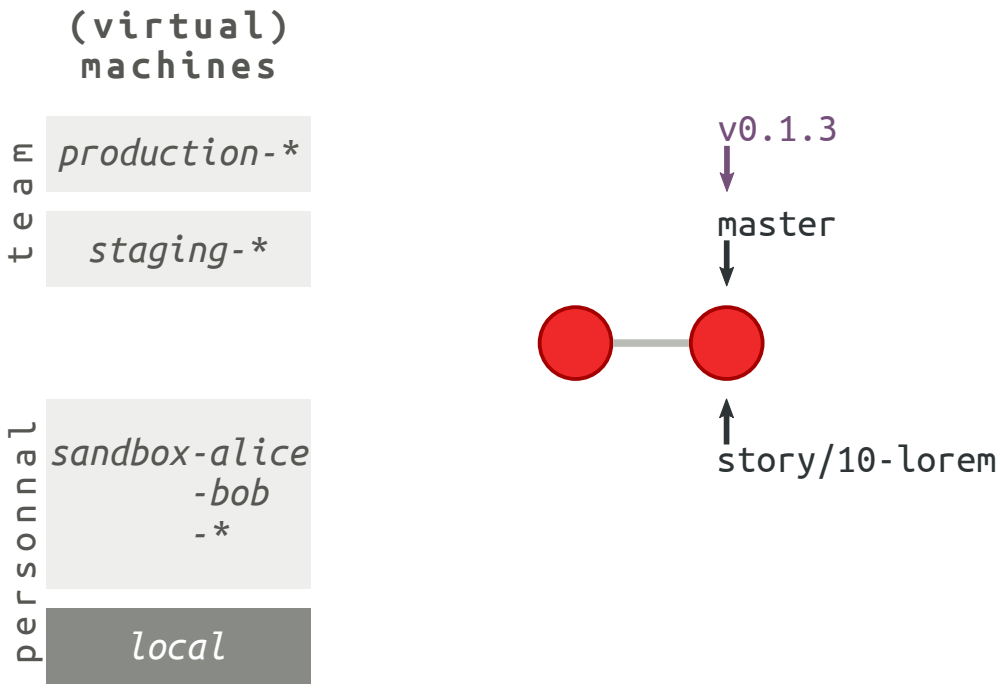
personnal
- *sandbox-alice*
- *-bob*
- *-**
- *local*



this is your terminal

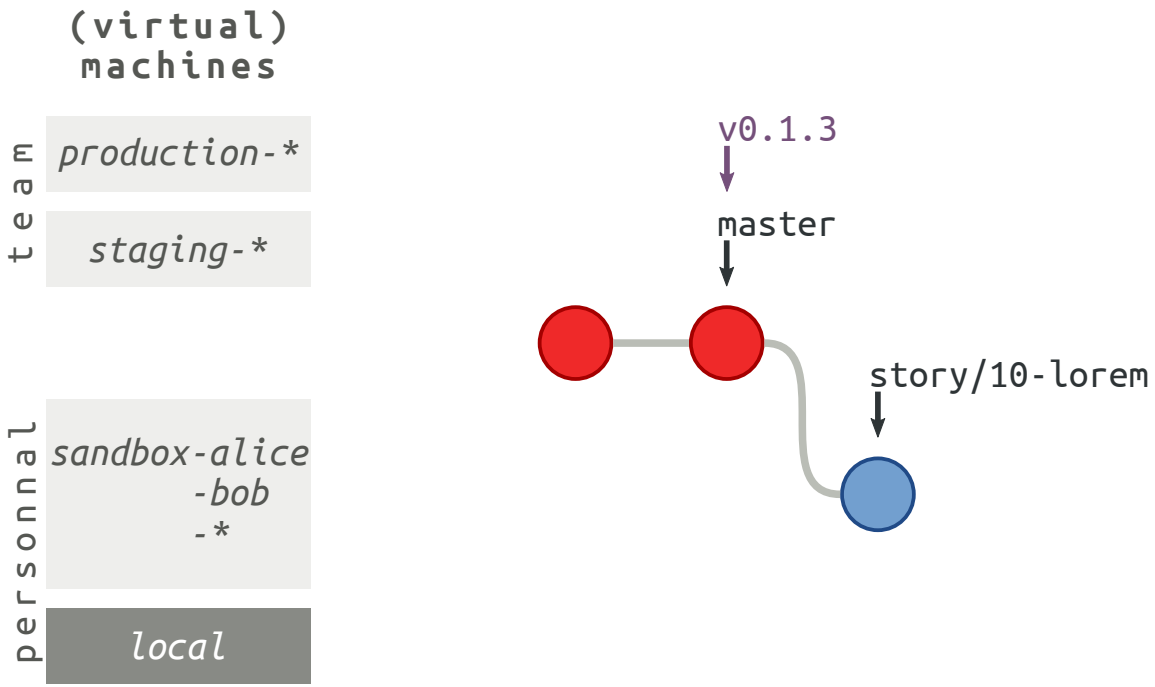
```
[machine] (git-pointer) $ command
```

Remember that branches are pointers?




```
[local] (master) $ git checkout -b story/10-lorem
```

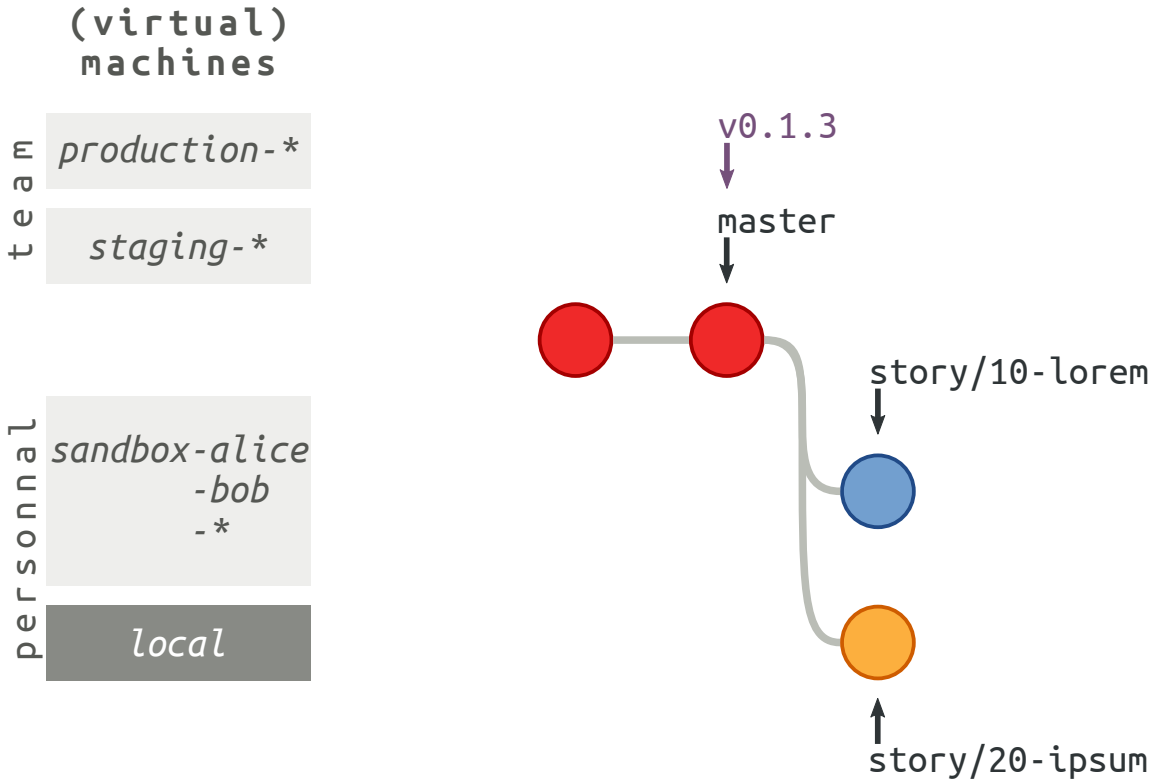
Commiting on the branch




```
[local] (story/10-lorem) $ git add <file>
[local] (story/10-lorem) $ git commit -s  # we explain the sign-off flag later
```

Using branches

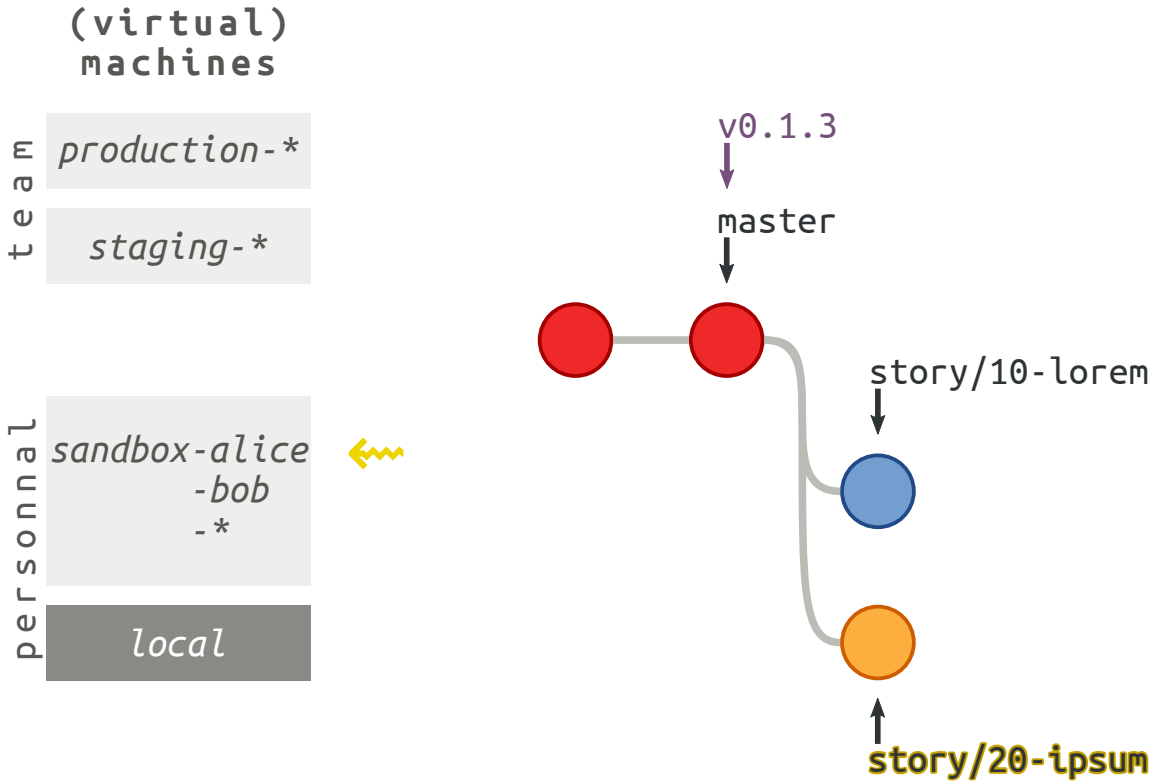
➔ One user story = one branch




```
[local] (master) $ git checkout -b story/20-ipsum
[local] (story/20-ipsum) $ git add <file>
[local] (story/20-ipsum) $ git commit -s
```

Deploy your story branches on a sandbox VM

➔ how to actually deploy depends on your project




```
[sandbox/alice] (story/20-ipsum) $ ./deploy
```

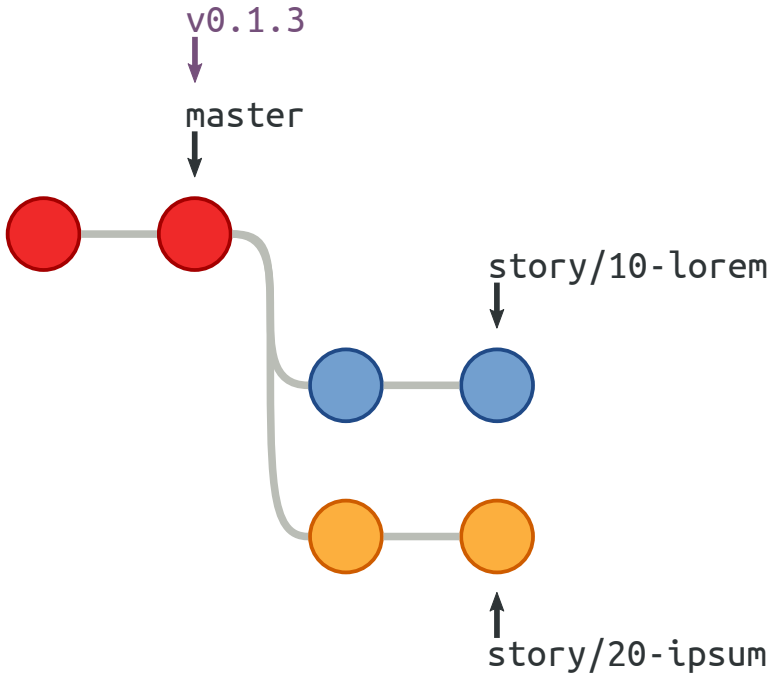
See e.g. wiki.unipart.io/wiki/Douglas_Production_release_checklist#Get_and_apply_changes

Sprint goes on…

(virtual) machines

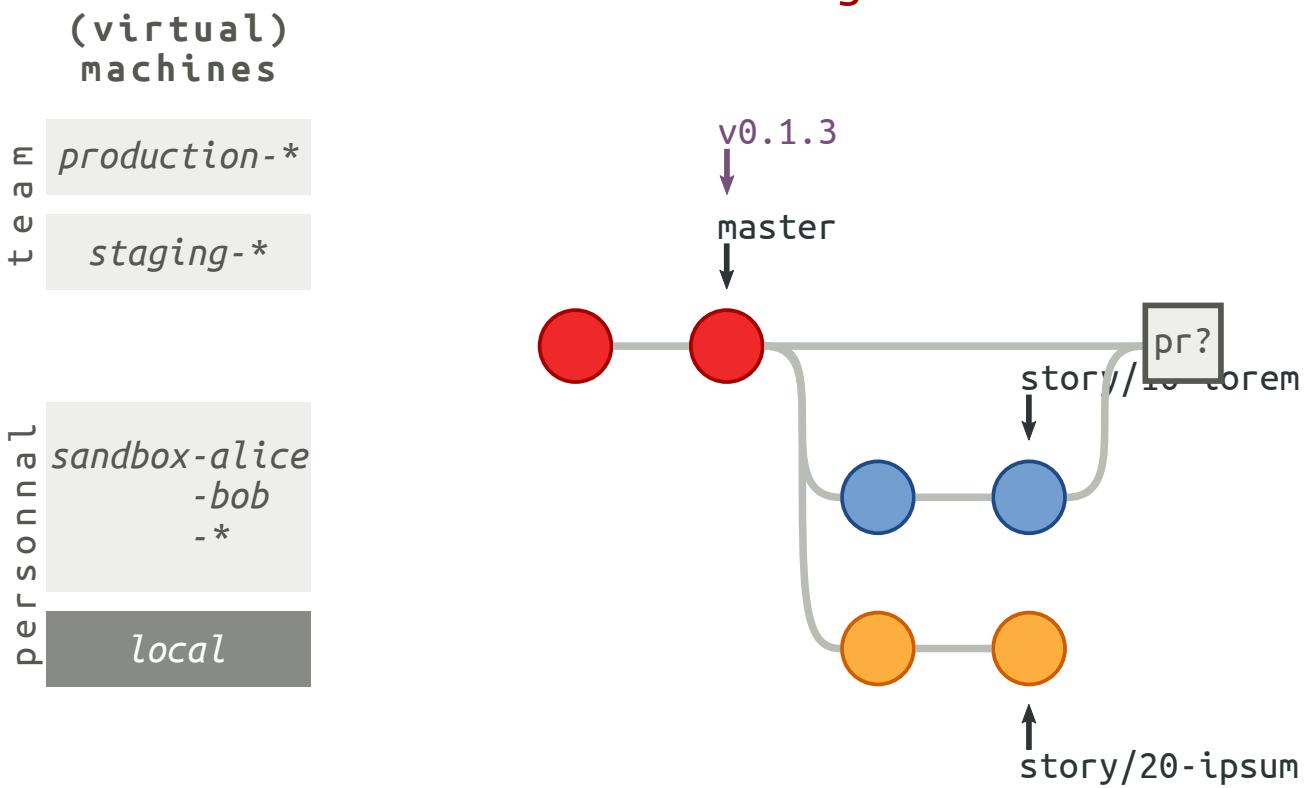
team
- *production-**
- *staging-**

personnal
- *sandbox-alice*
- *-bob*
- *-**
- *local*



When a branch is ready: create Pull Request

➔ Never merge to 'master': use Gogs' PR instead

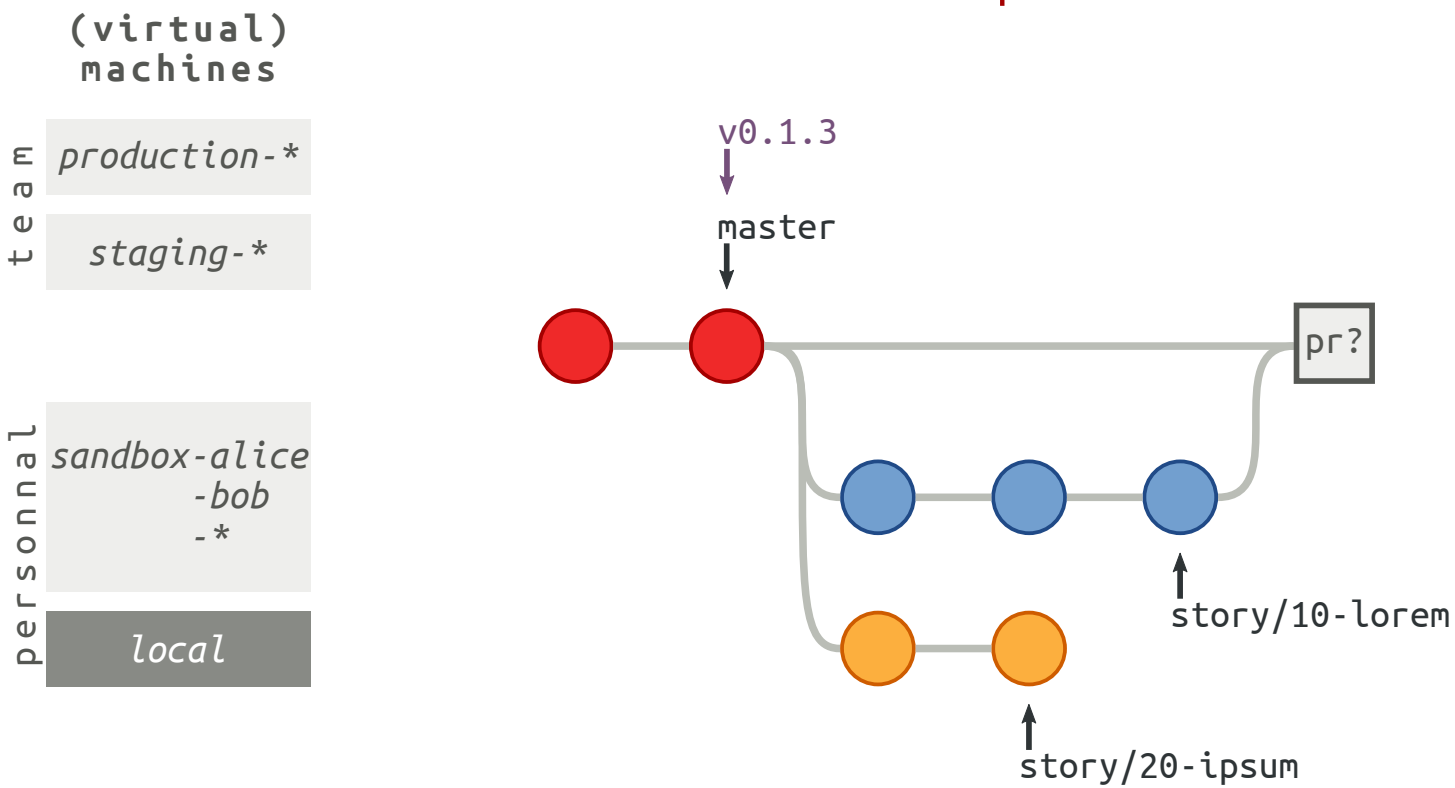



```
# this happens via Gogs GUI: you create a PR with "base: master … compare: story/10-lorem"
```

See e.g. wiki.unipart.io/wiki/Data_Science_LiveCoding#2019-01-22:_Gogs_features

Updating your (remote) branch updates the PR

➔ Don't close + open a new PR if there are still work to do




```
[local] (story/10-lorem) $ git add <file>
[local] (story/10-lorem) $ git commit
[local] (story/10-lorem) $ git push
```

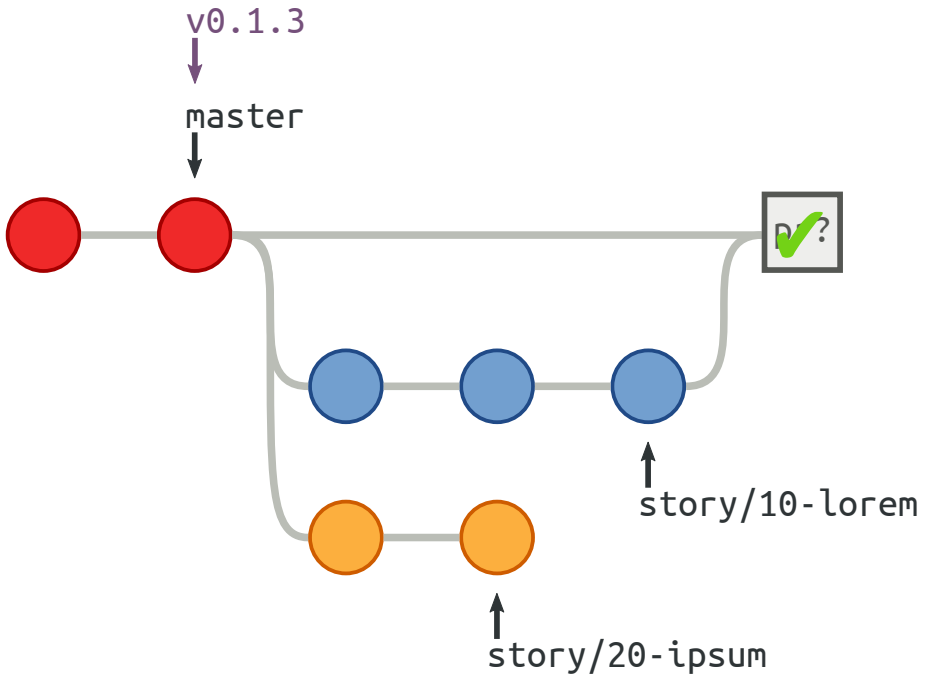
Accepting the PR: use Gogs

➔ Never merge your own PR

(virtual) machines

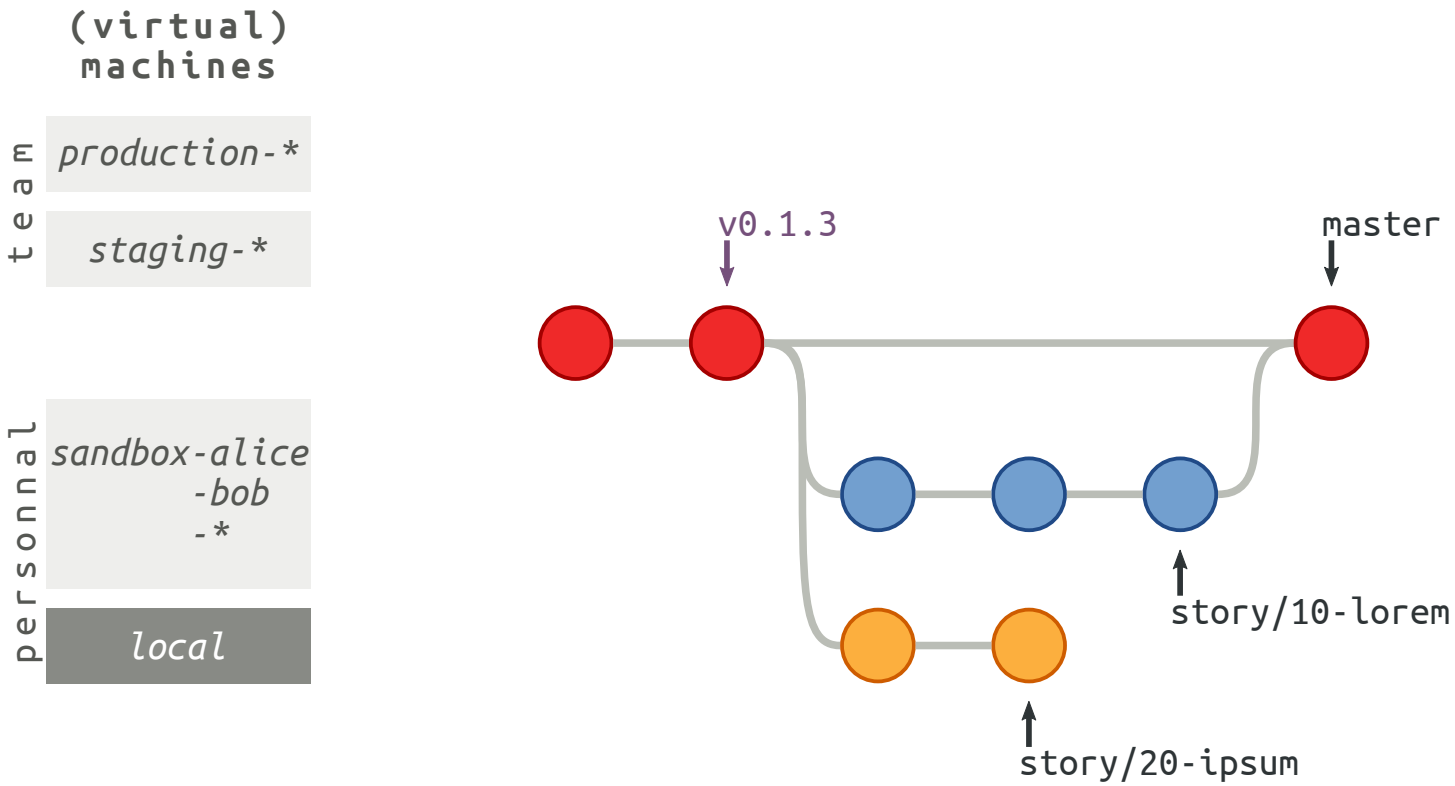
team
- *production-**
- *staging-**

personnal
- *sandbox-alice*
- *-bob*
- *-**
- *local*




```
# this happens via Gogs GUI
```

Accepting the PR creates a merge commit




```
# this happens via Gogs GUI
```

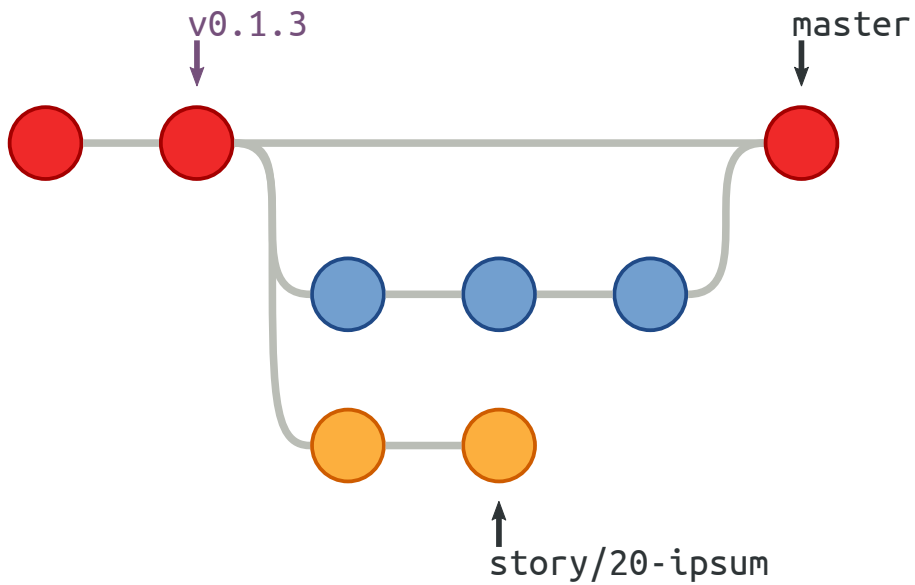
Once the PR is merged, delete the branch

➔ Don't work on a branch that has already been merged:
create a new one from 'master' instead

(virtual) machines

team
- *production-**
- *staging-**

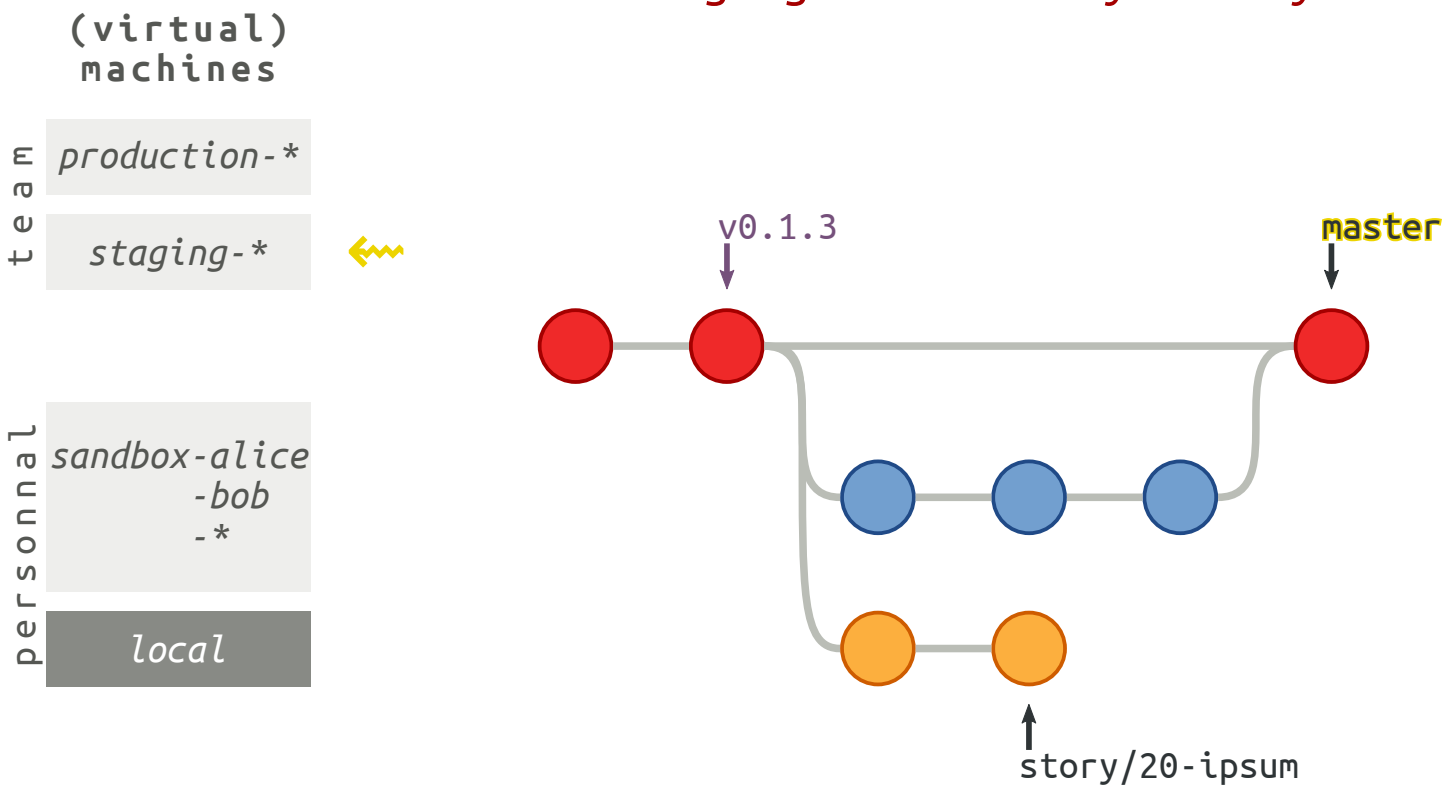
personnal
- *sandbox-alice*
- *-bob*
- *-**
- *local*




```
# deleting the branch on the shared repo is done via Gogs GUI. To do it on your local repo too:
[local] (master) $ git branch -d story/10-lorem
```

Once the PR is merged, deploy on 'staging'

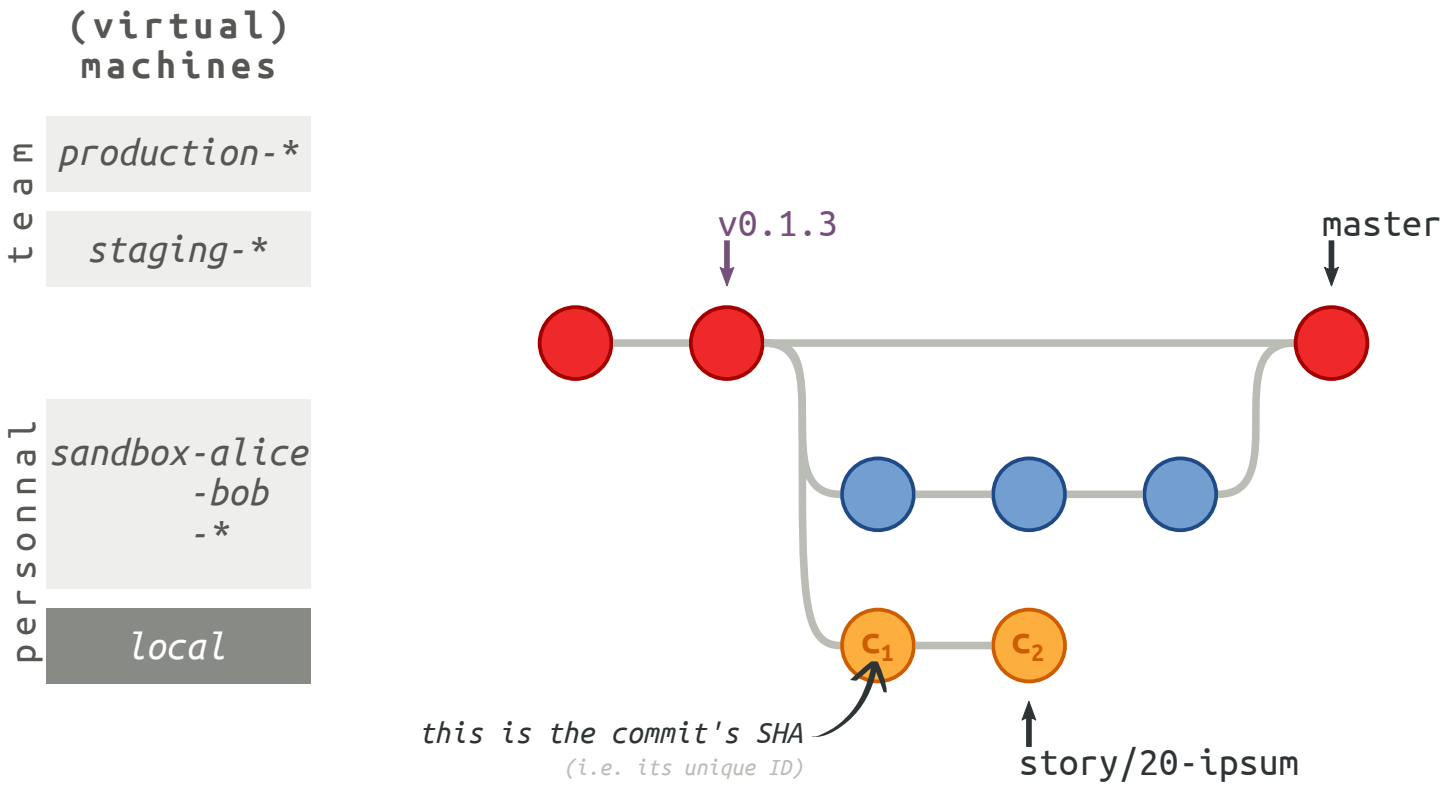
➔ 'staging' must always be sync with 'master'




```
[staging] (master) $ git pull --ff  # fast-forwarding ensure no merge commit is created, so repos stay synced
[staging] (master) $ ./deploy
```

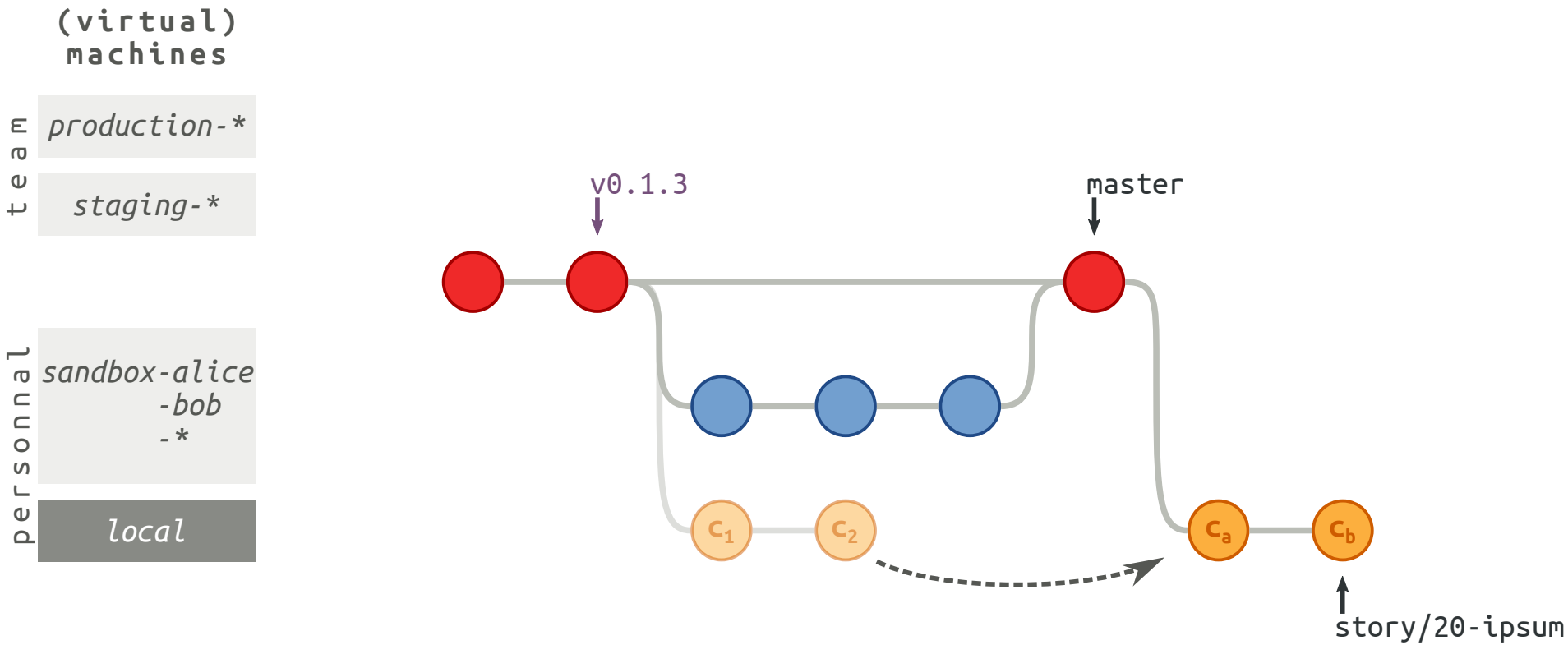
See e.g. wiki.unipart.io/wiki/Douglas_Production_release_checklist#Get_and_apply_changes

Merging an "outdated" branch: rebase first



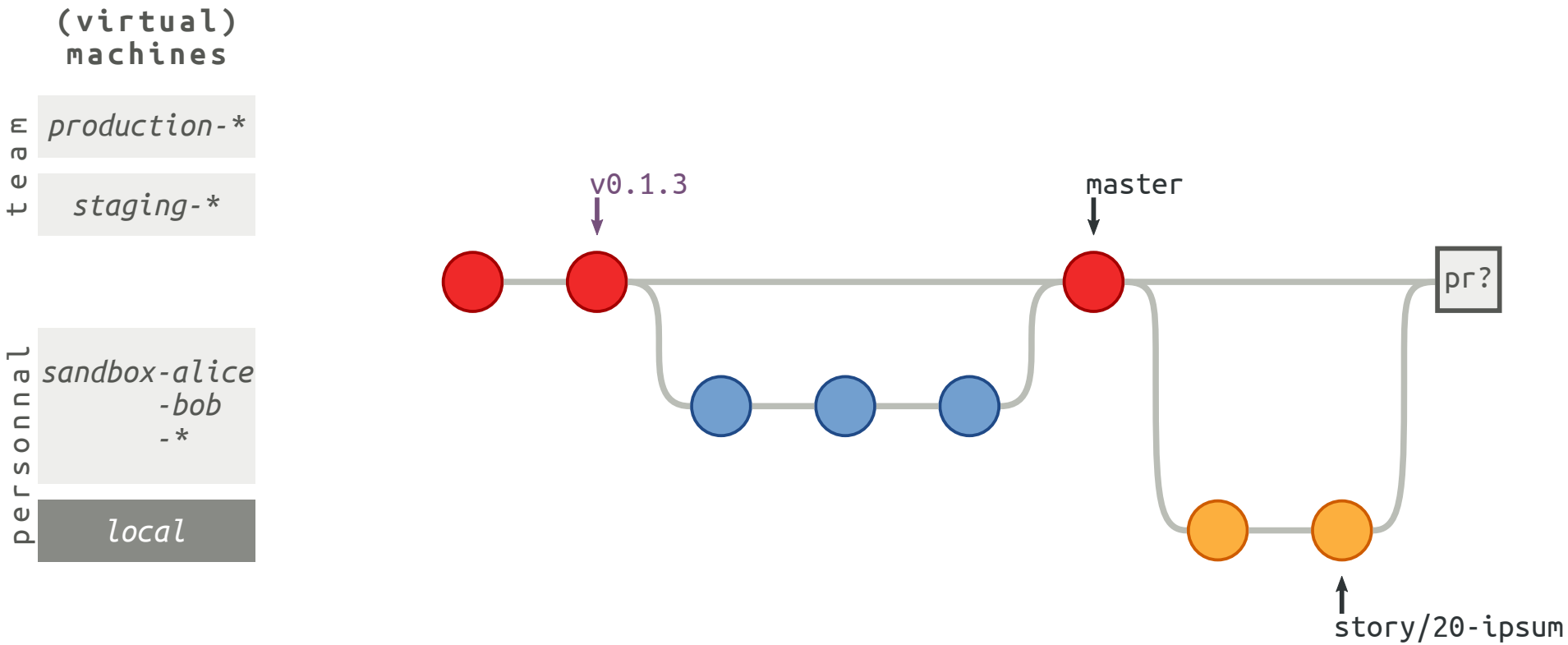

```
[local] (master) $ git pull --ff
[local] (master) $ git checkout story/20-ipsum
[local] (story/20-ipsum) $ git rebase master
```

Original commits are "replayed" on new base



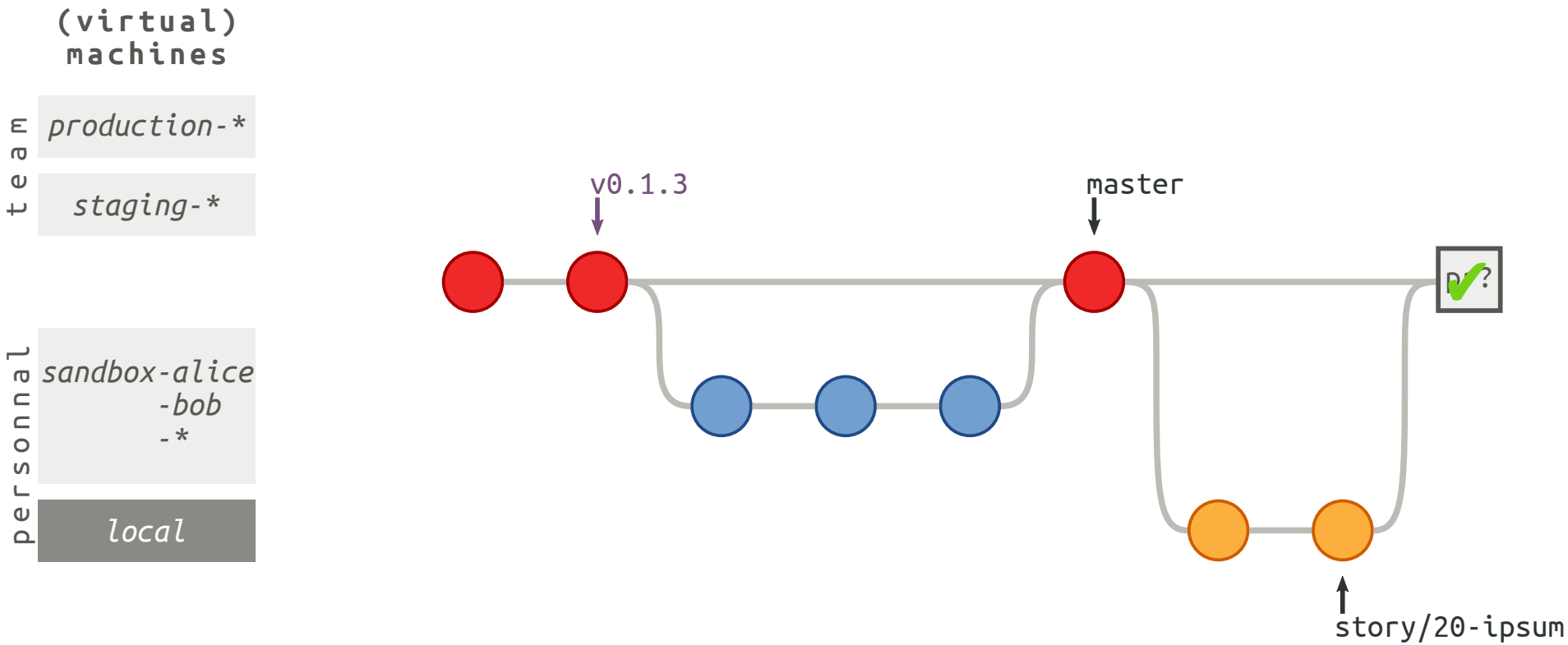

```
[local] (master) $ git pull --ff
[local] (master) $ git checkout story/20-ipsum
[local] (story/20-ipsum) $ git rebase master
```

The PR is then guaranteed conflict-less



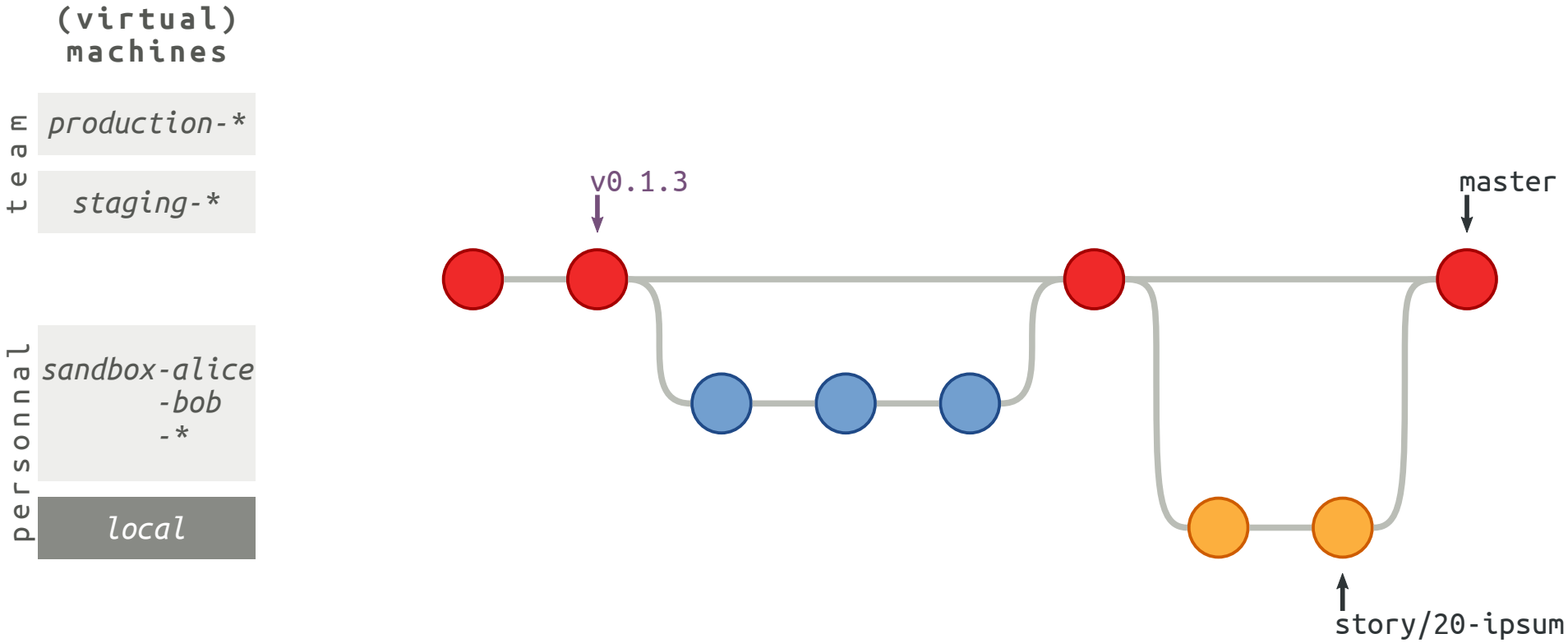

```
# this happens via Gogs GUI
```

Always ask someone else to merge your PR



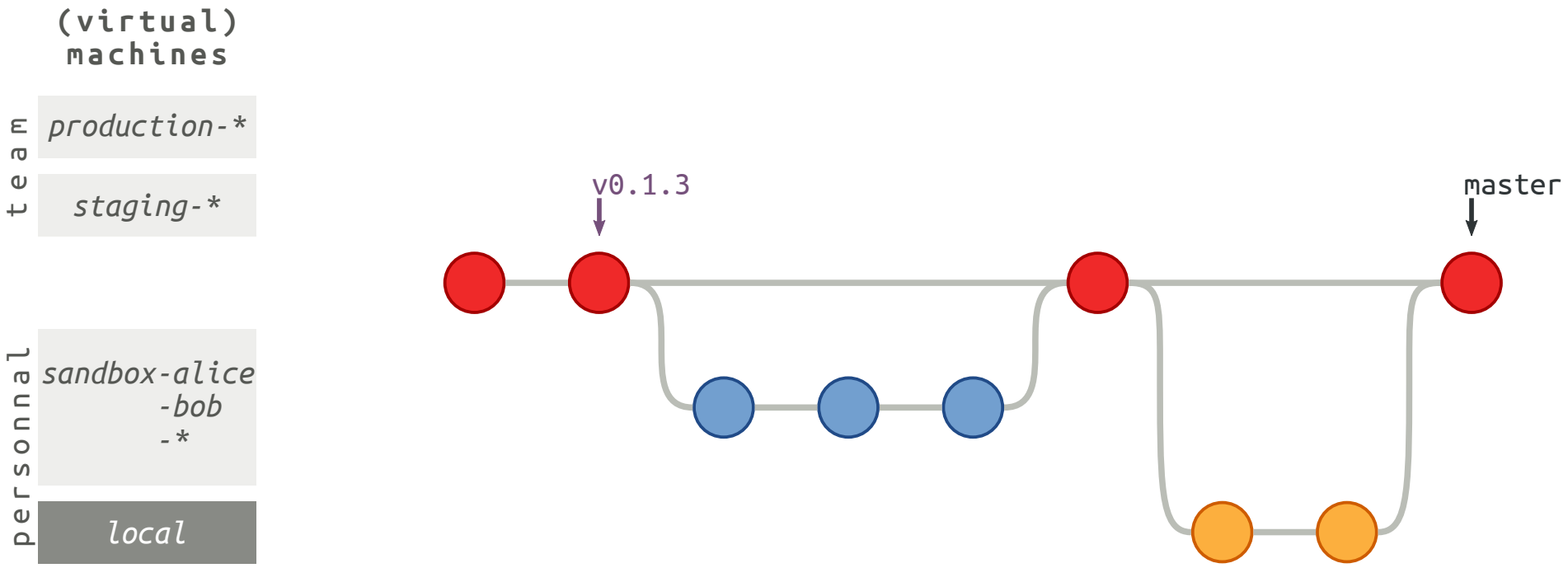

```
# this happens via Gogs GUI
```

What should you do after merging the PR?



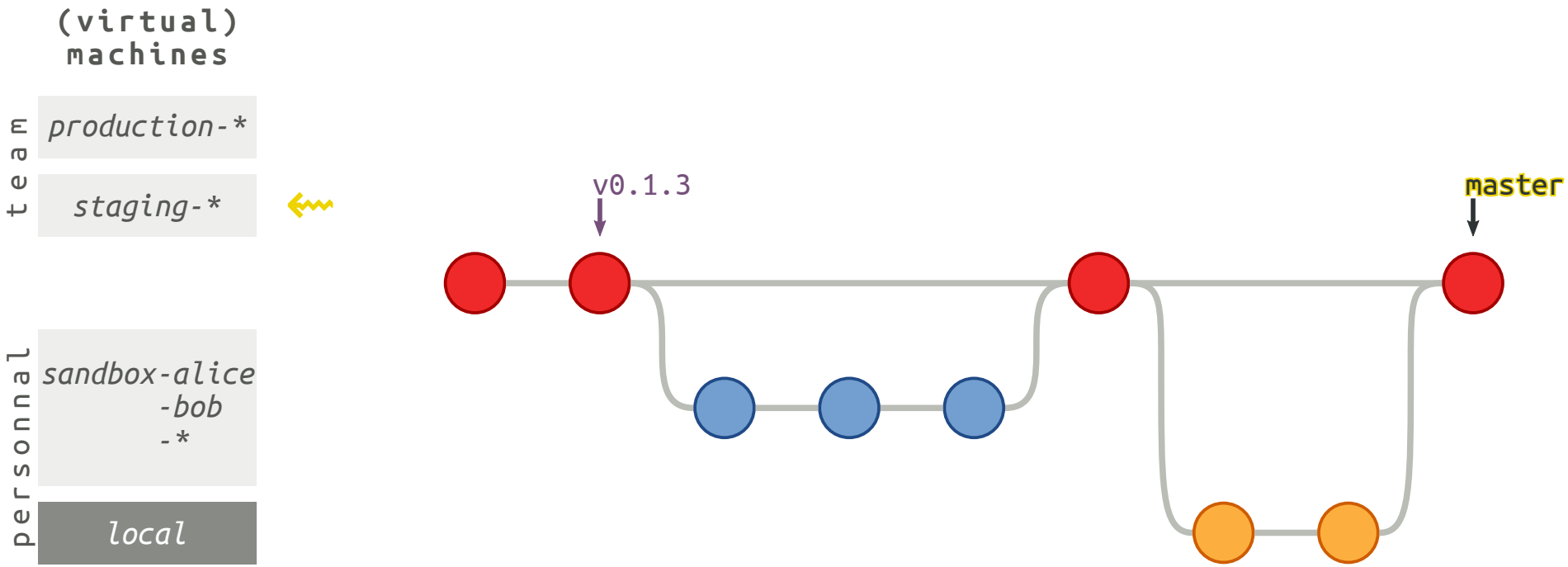

```
# this happens via Gogs GUI
```

1) delete branch (both on 'origin' and locally)




```
# delete remote branch via Gogs GUI, and the local one via:
[local] (master) $ git branch -d story/20-ipsum
```

2) deploy 'master' on 'staging'




```
[staging] (master) $ git pull --ff
[staging] (master) $ ./deploy
```

See e.g. wiki.unipart.io/wiki/Douglas_Production_release_checklist#Get_and_apply_changes

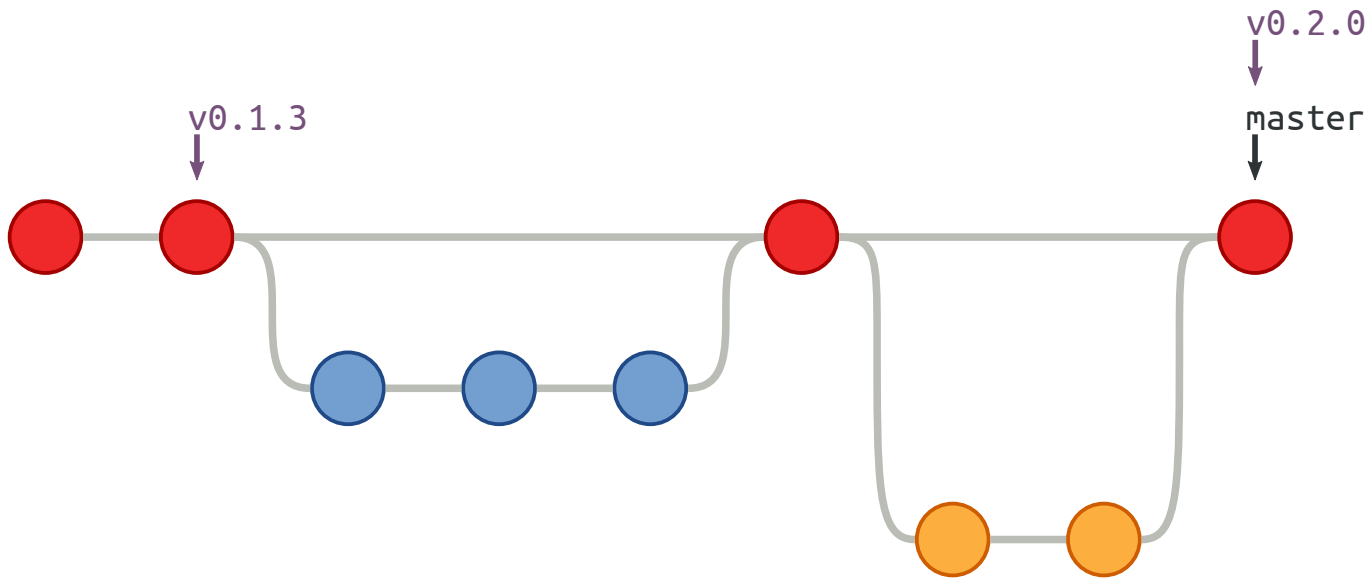
Create a new release via Gogs GUI…

➔ We follow "Semantic Versioning" specs – see semver.org

(virtual) machines

team
- *production-**
- *staging-**

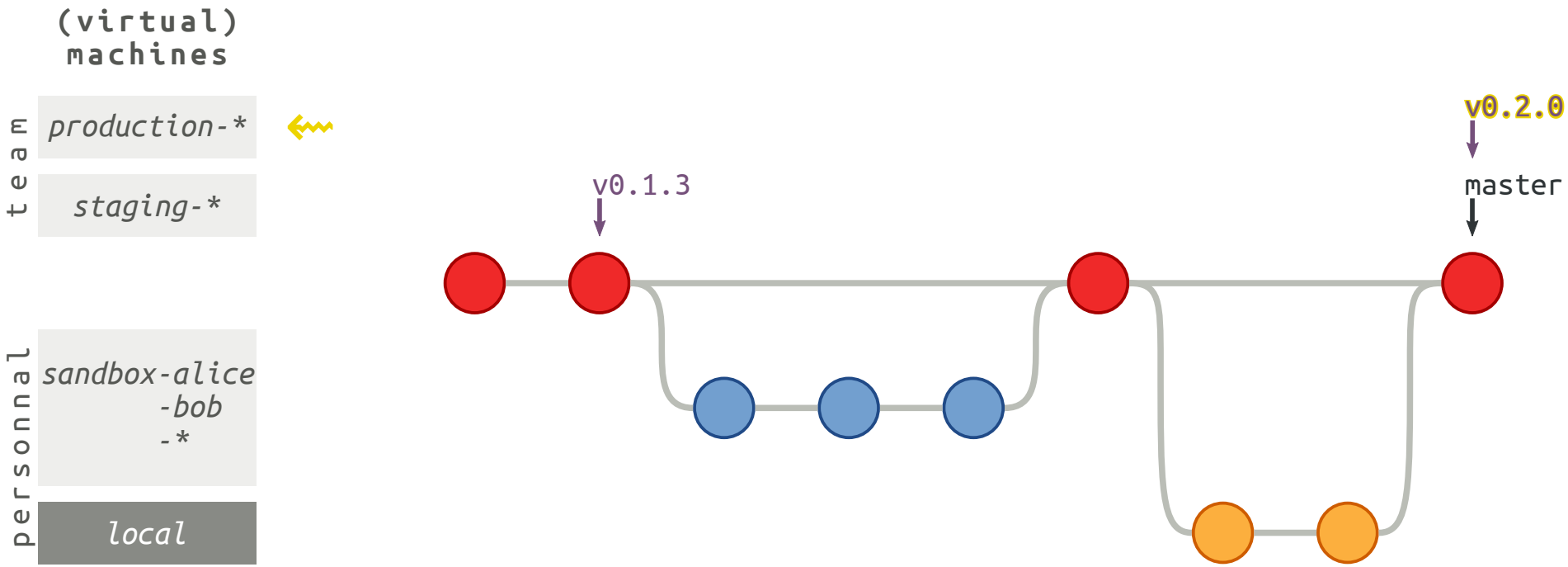
personnal
- *sandbox-alice*
- *-bob*
- *-**
- *local*




```
# this happens via Gogs GUI
```

See wiki.unipart.io/wiki/Douglas_Production_release_checklist

…and deploy it on 'production'

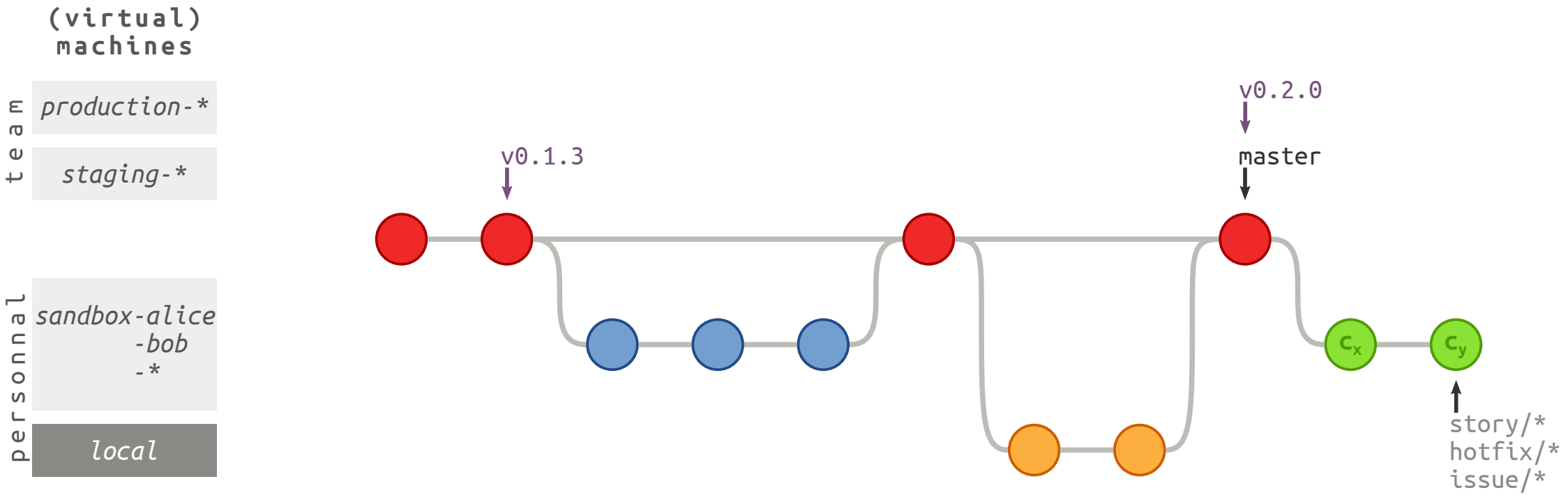



```
[production] (master) $ git pull --ff && git fetch --tags
[production] (master) $ git checkout v0.2
[production] (v0.2) $ ./deploy
```

See wiki.unipart.io/wiki/Douglas_Production_release_checklist

Create branches for everything

➔ Never commit while on branch 'master'




```
[local] (master) $ git checkout -b <prefix/##-short_description>
[local] (<prefix/##-short_description>) $ git add <file> && git commit
[local] (<prefix/##-short_description>) $ git push -u origin/<prefix/##-short_description>
```

Last but not least

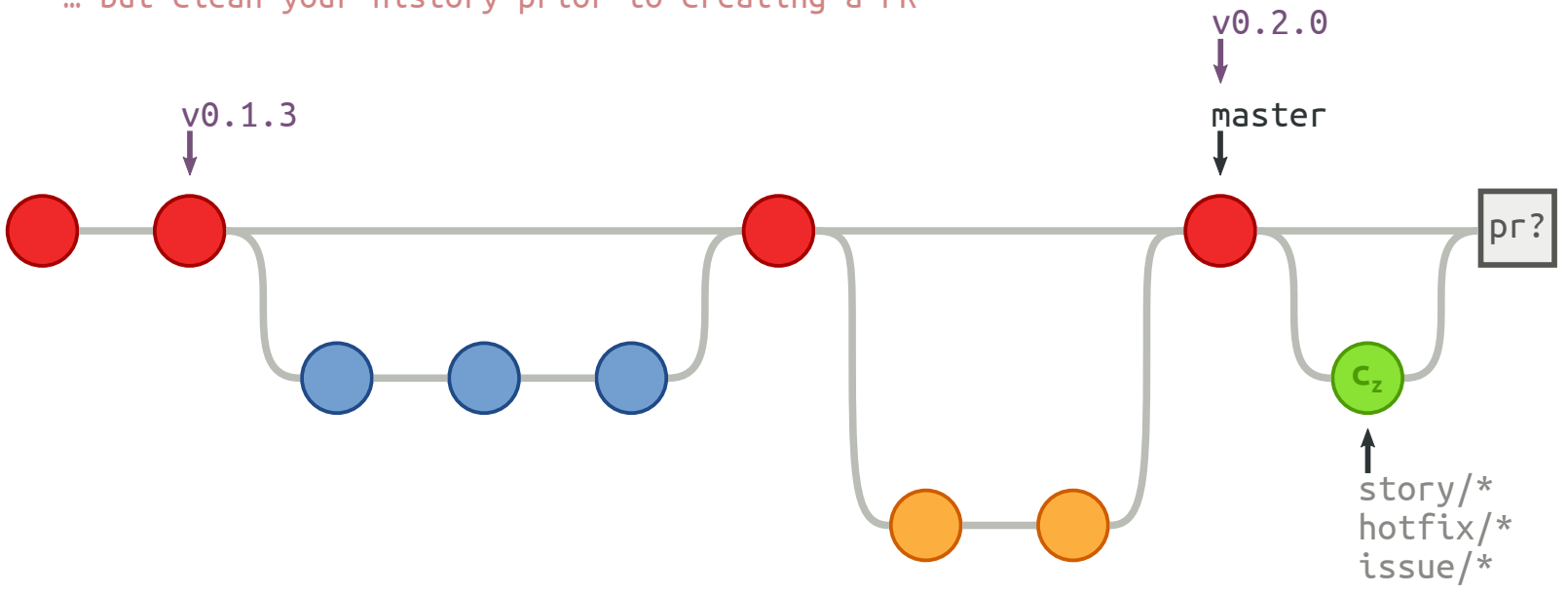
➔ Never create a PR with commits leading to a broken state

Commit early & often is important
Do what you want on your branch…
… but clean your history prior to creating a PR

(virtual) machines

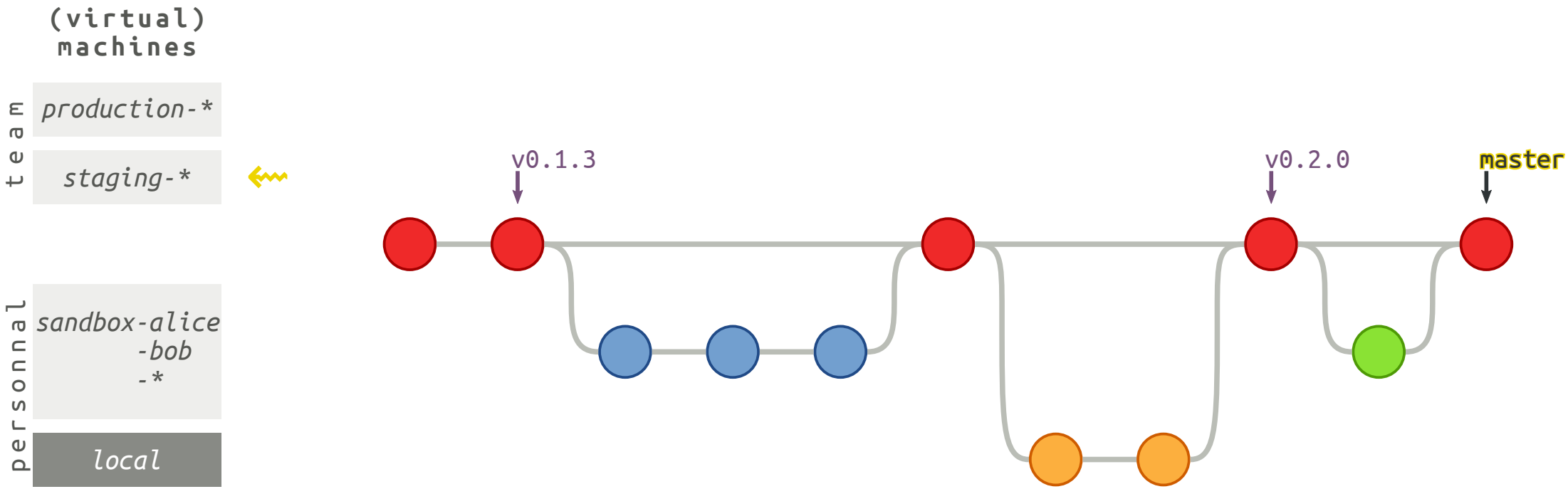
team
- *production-**
- *staging-**

personnal
- *sandbox-alice*
- *-bob*
- *-**
- *local*




```
[local] (<prefix/##-short_description>) $ git fetch
[local] (<prefix/##-short_description>) $ git rebase -i master
```

Last but not least




```
[staging] (master) $ git pull --ff
[staging] (master) $ ./deploy
```

One last thing: writing a good commit message

Limit the subject line to 50 characters

Capitalize the subject line

Do not end the subject line with a period

Use the imperative mood

You can use prefixes like "prefix: Foo bar"

Wrap the body at 72 characters

Use the body to explain what and why (vs. how)

Link to relevant Taiga's task (or user story)

See tree.taiga.io/support/integrations/gogs-integration/

```
[local] (story/00-branch) $ git commit --signoff

Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like `log`, `shortlog`
and `rebase` can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here's the place to explain them.

Further paragraphs come after blank lines.
 - Bullet points are okay, too
 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: TG-123
Signed-off-by: Jane Doe <jane.doe@unipart.io>
```

Separate subject from body with a blank line

Signoff your commits (shorthand: -s)

Adapted from chris.beams.io/posts/git-commit/

Example of good commit messages

Prefixes should describe parts/modules of your code

Keep quite generic prefixes

Don't use prefixes for User-story/Task or type of modif (fix,imp, …): use instead the "Resolve" footer, or the first verb of the subject, respectively.

```
[local] (story/00-branch) $ git commit --signoff

db loader: Refactor data processing to improve speed

When fetching data for a large period (~3 months+), processing data
has been reported too long by client (meeting 2019-02-29).

The new processing limits the amount of nested loops. Additional
unit tests are added.

New error status:
 - 427: when start_date is later than end_date
 - 804: when trying to fetch more than 12 months

Resolves: TG-42
See-also: TG-210
Signed-off-by: Jane Doe <jane.doe@unipart.io>
```

The firt word of the subject should be a common verb detailing the type of changes, like: Add, Remove, Update, Refactor, Fix, …

If there is an "and" in the subject, you should consider splitting it in smaller ones

One-line messages are fine too… when you commit trivial modifs

```
[local] (story/00-branch) $ git commit --signoff

Bump version number in documentation

Signed-off-by: Jane Doe <jane.doe@unipart.io>
```