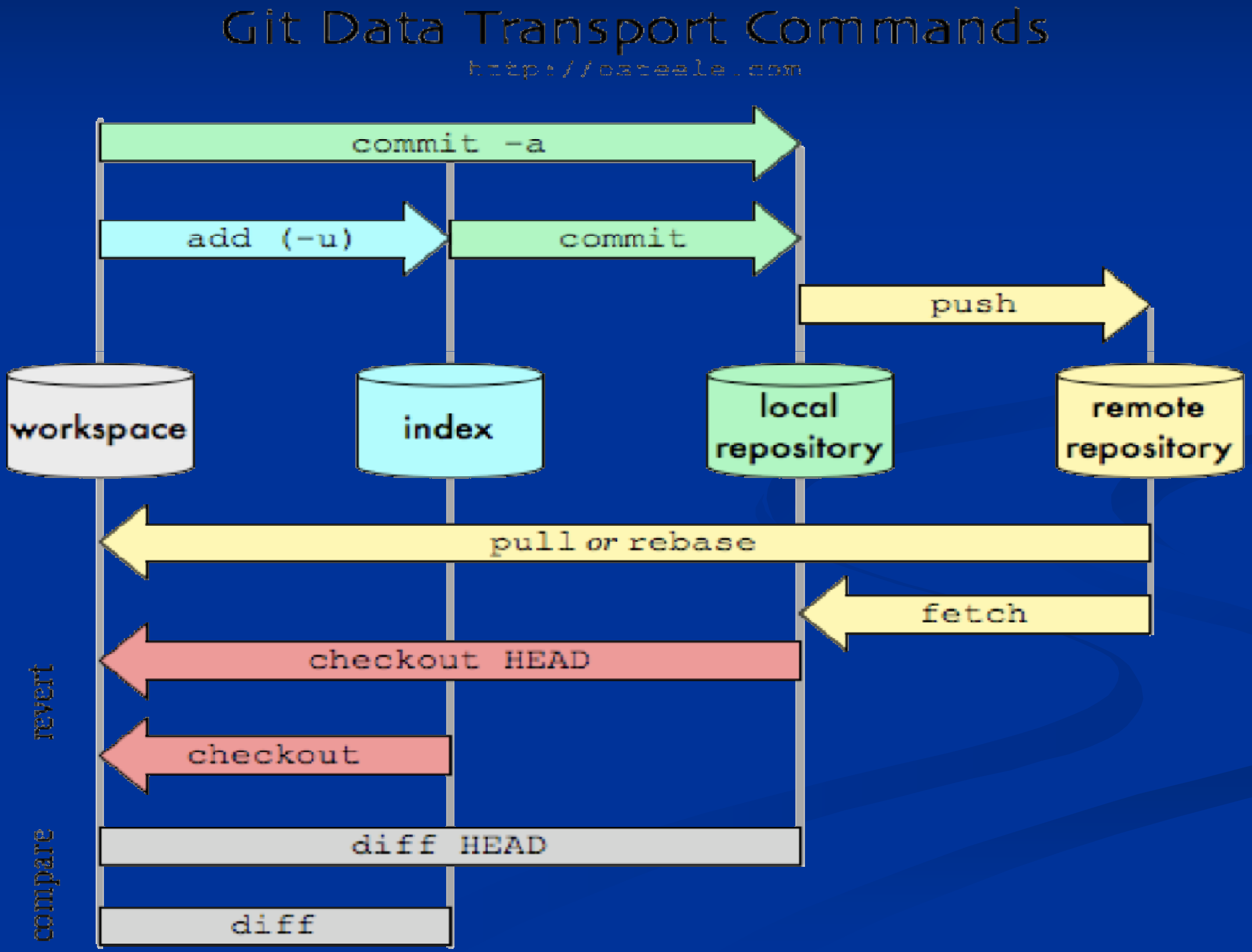
GIT Workflows

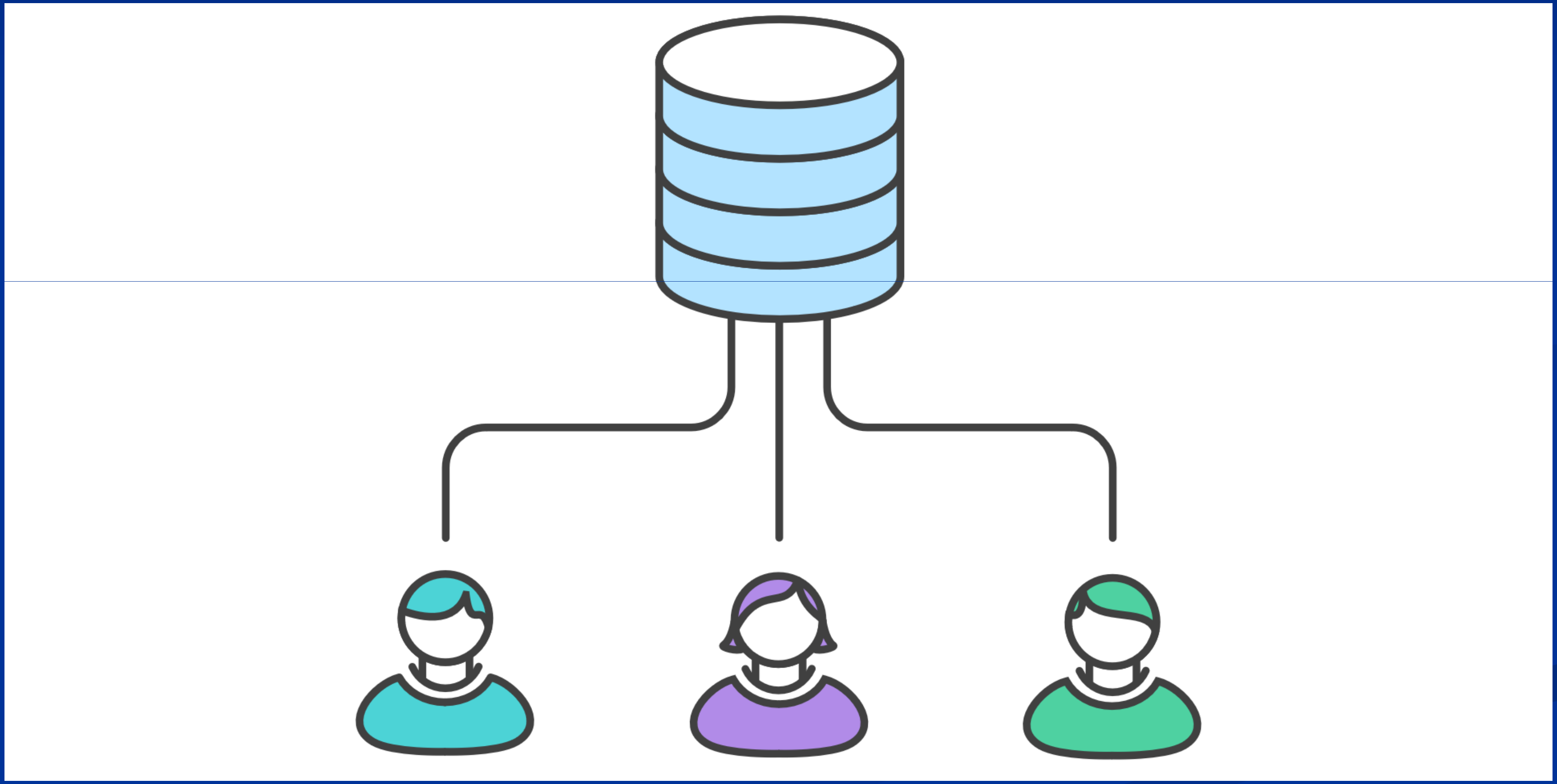
References (and sources of the following graphics):

- Workflows comparison by Atlassian (https://www.atlassian.com/git/tutorials/comparing-workflows)
- A successful Git branching model by Vincent Driessen (http://nvie.com/posts/a-successful-git-branching-model)
- GitFlow: safely merge develop changes to a feature branch (http://stackoverflow.com/questions/21661263/gitflow-safely-merge-develop-changes-to-a-feature-branch/21674420#21674420)
- Discussion on pros and cons of GitFlow (https://barro.github.io/2016/02/a-succesful-git-branching-model-considered-harmful)

Git


Git Data Transport Commands
http://osteele.com
commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull *or* rebase
fetch
checkout HEAD
checkout
diff HEAD
diff
revert
compare

Centralized Workflow



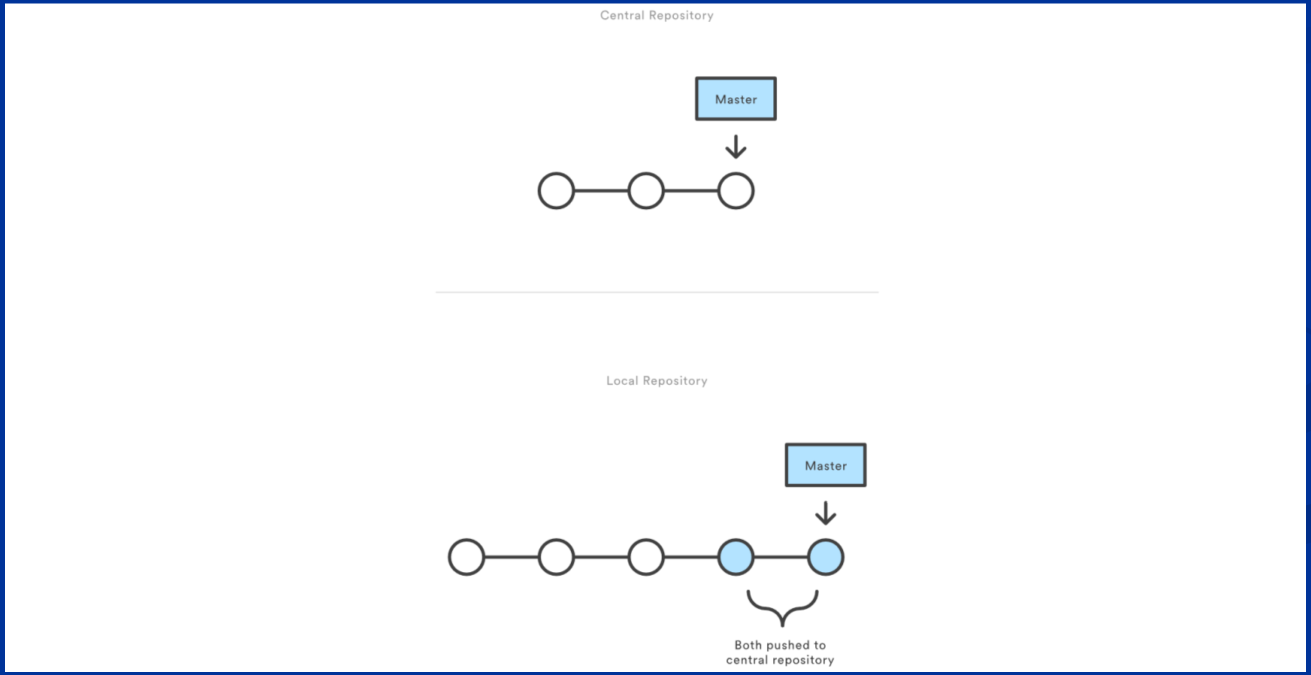
Bill Mary John

Centralized Workflow

- One central repository (**origin**), cloned by developers
- Only **master** branches are required
- Project development in the same way as with SVN
- Advantages comparing to SVN:
  - each developer has a local copy of the project
  - each developer can work in an isolated environment (local commits) – deferred synchronization

Centralized Workflow

- Publishing changes – **push** master to central repository (equiv. to svn commit, but adds all local commits to central master branch)



Centralized Workflow

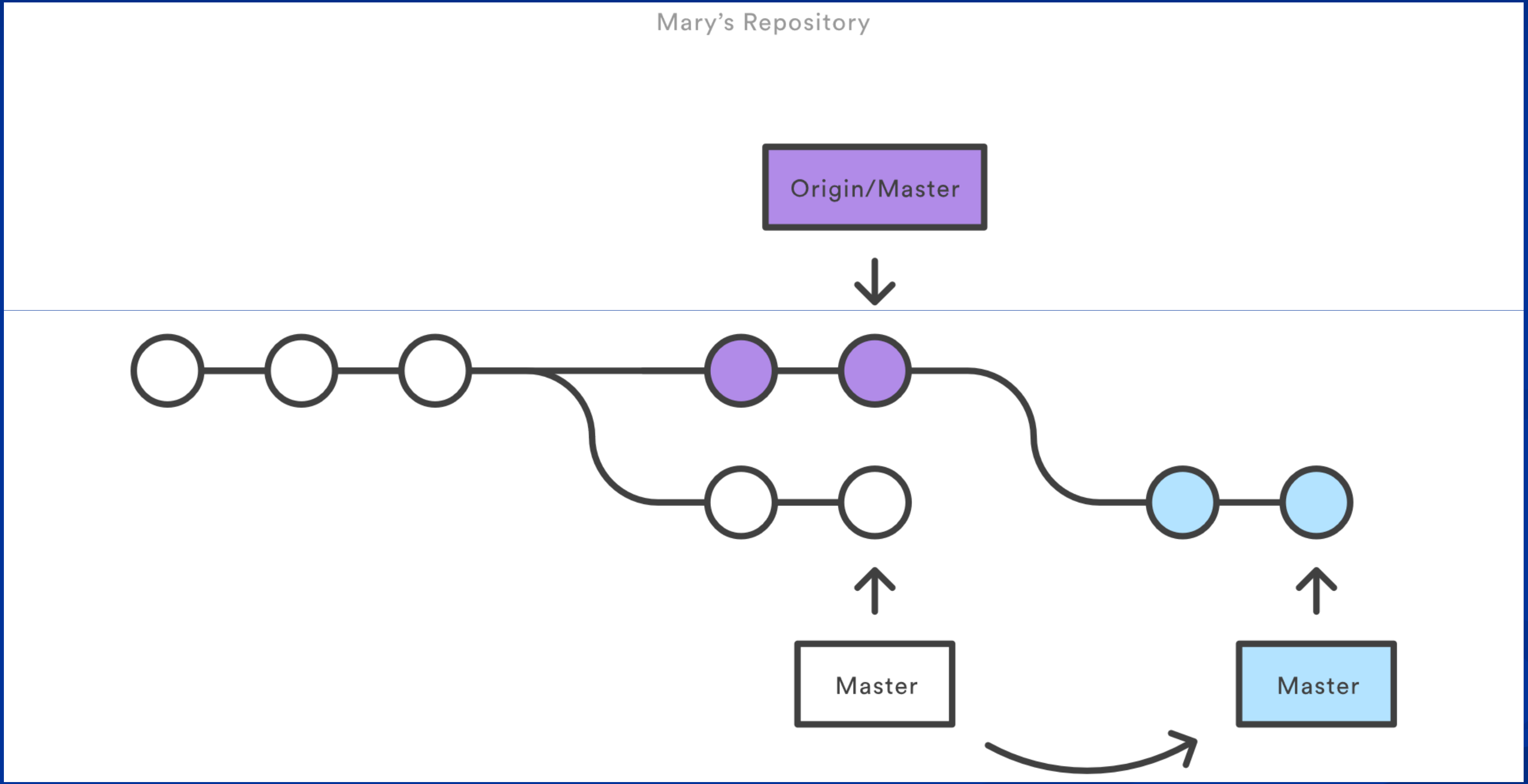
- If push fails – fetch from central repo and **rebase** local commits on top of them
- If local changes conflict with upstream commit, GIT pauses rebasing – **manual conflict resolution**

Centralized Workflow - example

- First, initialization of empty central repo or import of an existing repo
- Central repo must be **bare**

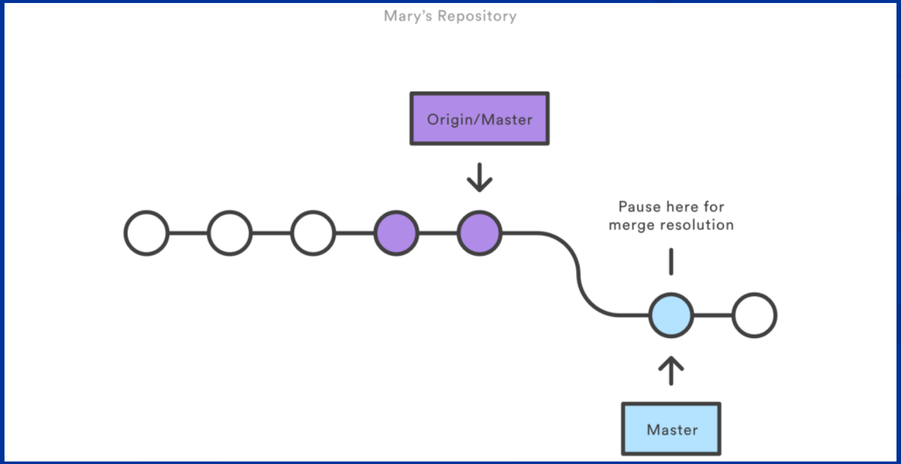
  ssh user@host git init --bare /path/to/repo.git
- Developers clone central repo (**origin**)
- Publishing of local commits:
  - git push origin master
  - git pull --rebase origin master (if push fails; avoids merge commit)

Centralized Workflow - example



Centralized Workflow - example

- Rebasing – transferring each local commit to the updated master branch one at a time
- This allows to catch merge conflicts on a commit-by-commit basis rather than resolving all of them in one massive merge commit



Centralized Workflow - example

- Once merge conflicts are resolved:

  ```
  git add <some-file> (staging)
  git rebase --continue
  ```

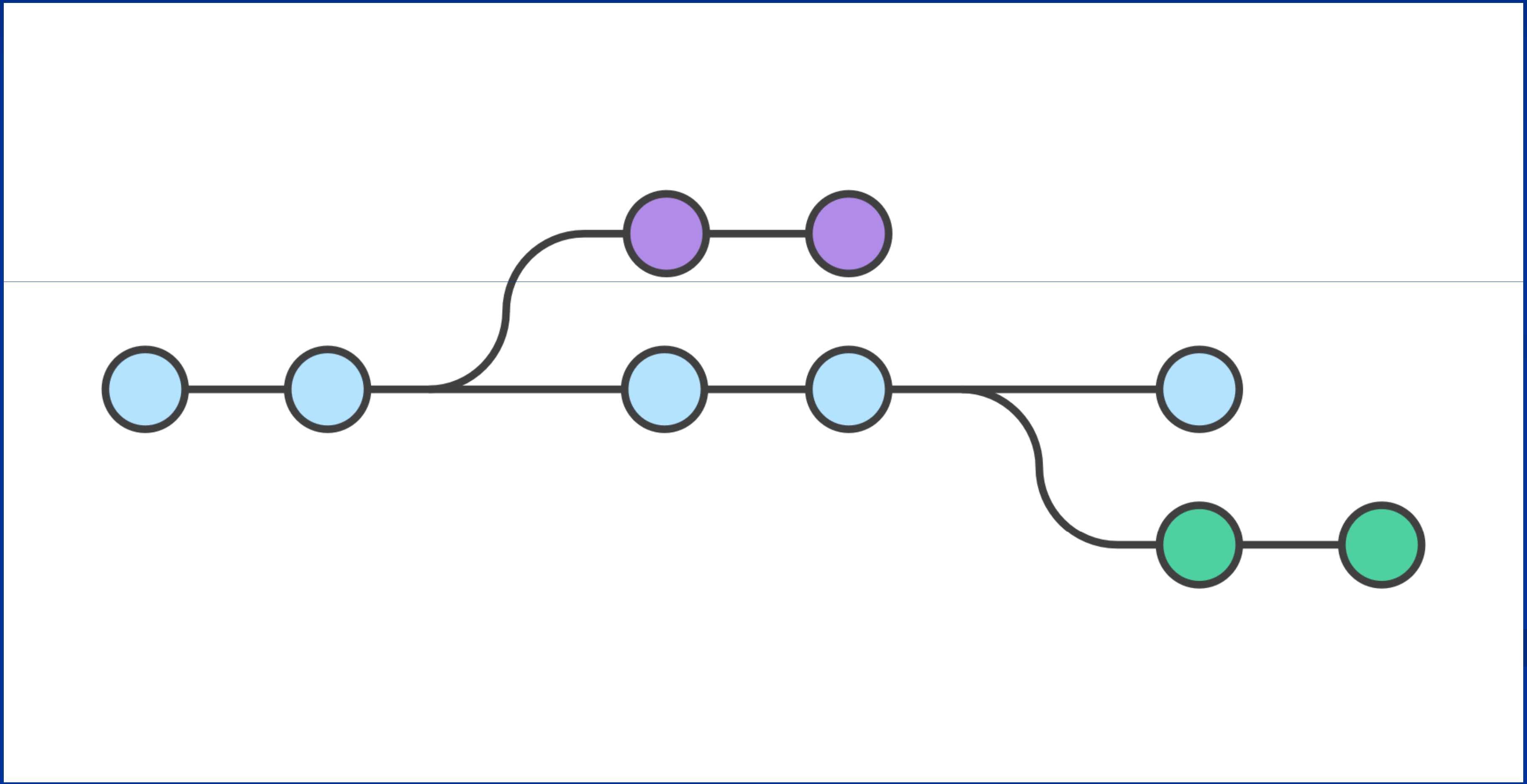
- Once something goes bad:

  ```
  git rebase --abort
  ```

- Once `git pull --rebase origin master` done:

  ```
  git push origin master
  ```

Feature Branch Workflow



Feature Branch Workflow

- All feature development should take place in a dedicated branch instead of the master branch (encapsulation)
- Feature branches should have descriptive names, like animated-menu-items or issue-#1061
- Pull requests – enable discussion around a branch (feature) before it gets integrated into the official project (master branch); see, e.g., Gerrit
- Thus, master should not contain broken code

Feature Branch Workflow

- Feature branches can (and should) be pushed to the central repository.
- This makes it possible to share a feature with other developers without touching official code.
- This is also a convenient way to back up everybody’s local commits.
- Publishing changes: synchronizing local master with origin’s master, merging feature branch into master, pushing master back to central repo

Feature Branch Workflow - example

- Mary begins a new feature:

```
git checkout -b marys-feature master
```

- Mary edits, stages, and commits changes in the usual fashion, building up her feature with as many commits as necessary:

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

Feature Branch Workflow - example

- Before lunch, Mary pushes her feature branch to central repo (backup, access for other collaborators)

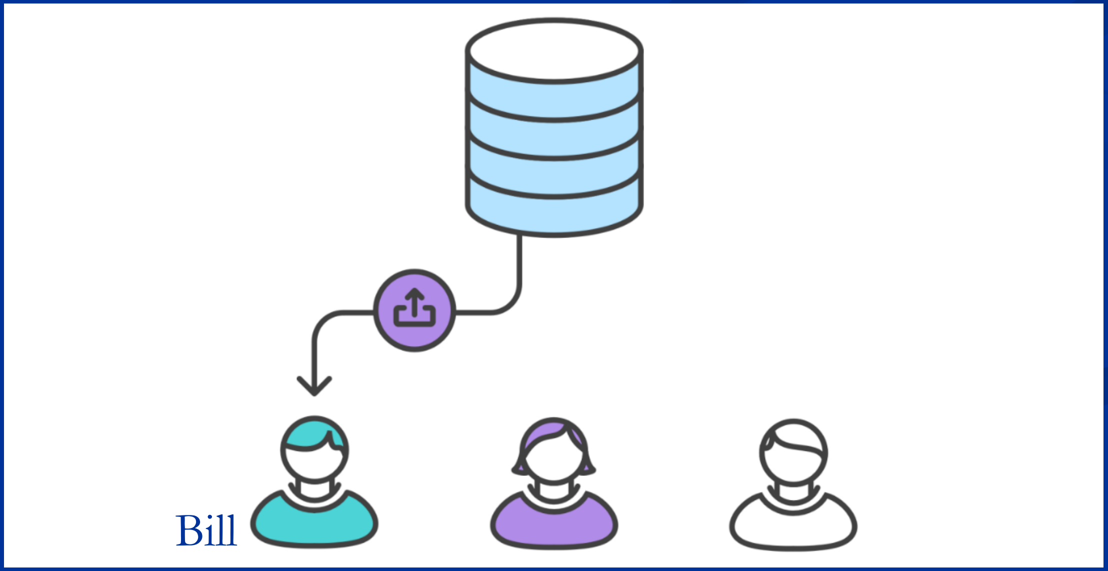
  ```
  git push -u origin marys-feature
  ```

  (-u flag adds it as a remote tracking branch)
- After lunch, Mary completes her feature (all commits) and publishes it:

  ```
  git push
  ```

Feature Branch Workflow - example

- Then, she fires the **pull request** (merge request) in her Git GUI asking to **merge marys-feature into master**, and team members will be notified automatically



Feature Branch Workflow - example

- Bill gets the pull request and takes a look at marys-feature. He decides he wants to make a few changes before integrating it into the official project, and he and Mary have some back-and-forth via the pull request.
- To make the changes, Mary uses the exact same process as she did to create the first iteration of her feature. Her activity shows up in the pull request, and Bill can still make comments along the way.

Feature Branch Workflow - example

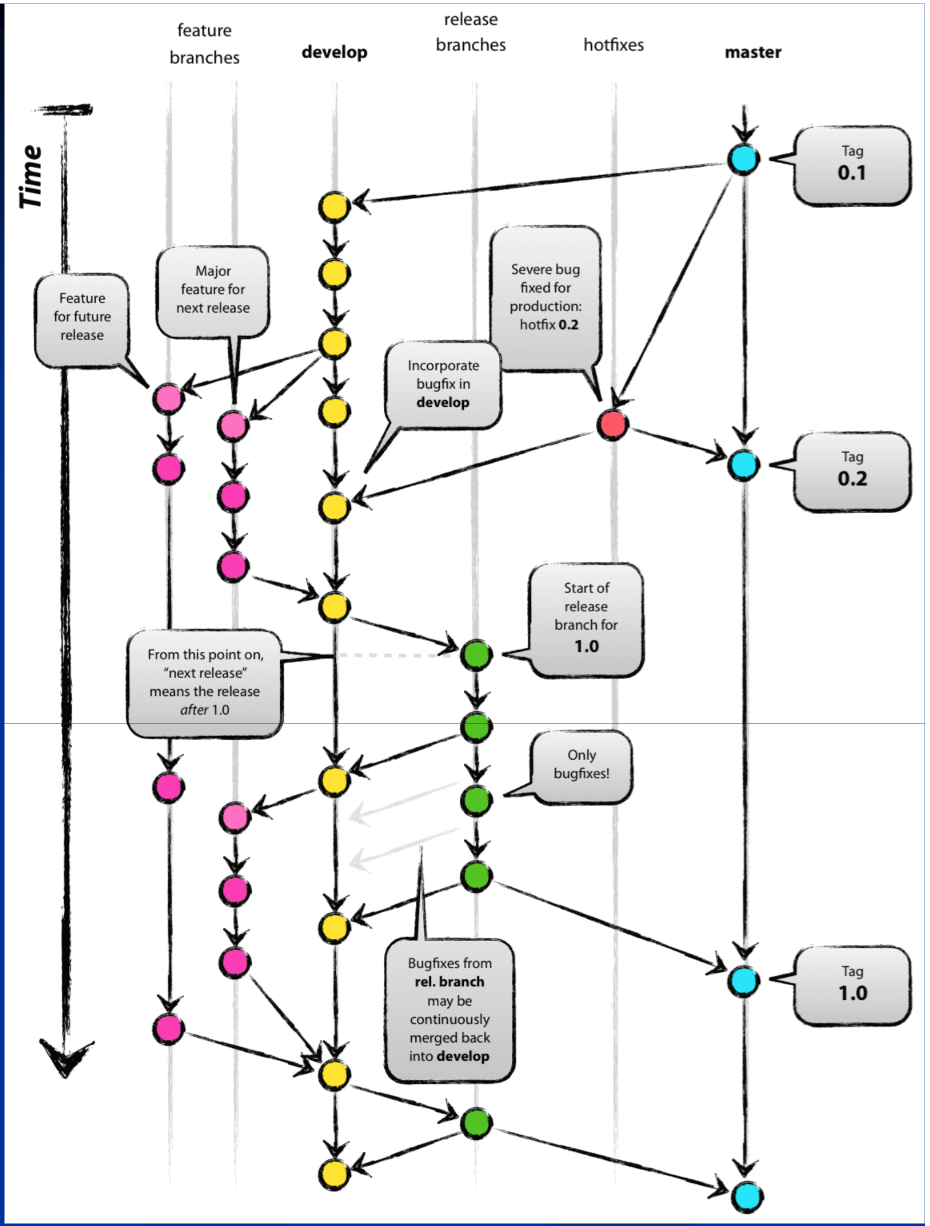
- Bill could pull marys-feature into his local repo and work on it on his own. Any commits he added would also show up in the pull request.
- Once Bill is ready to accept the pull request, someone needs to merge the feature:

```
git checkout master
git pull
git pull origin marys-feature
git push
```

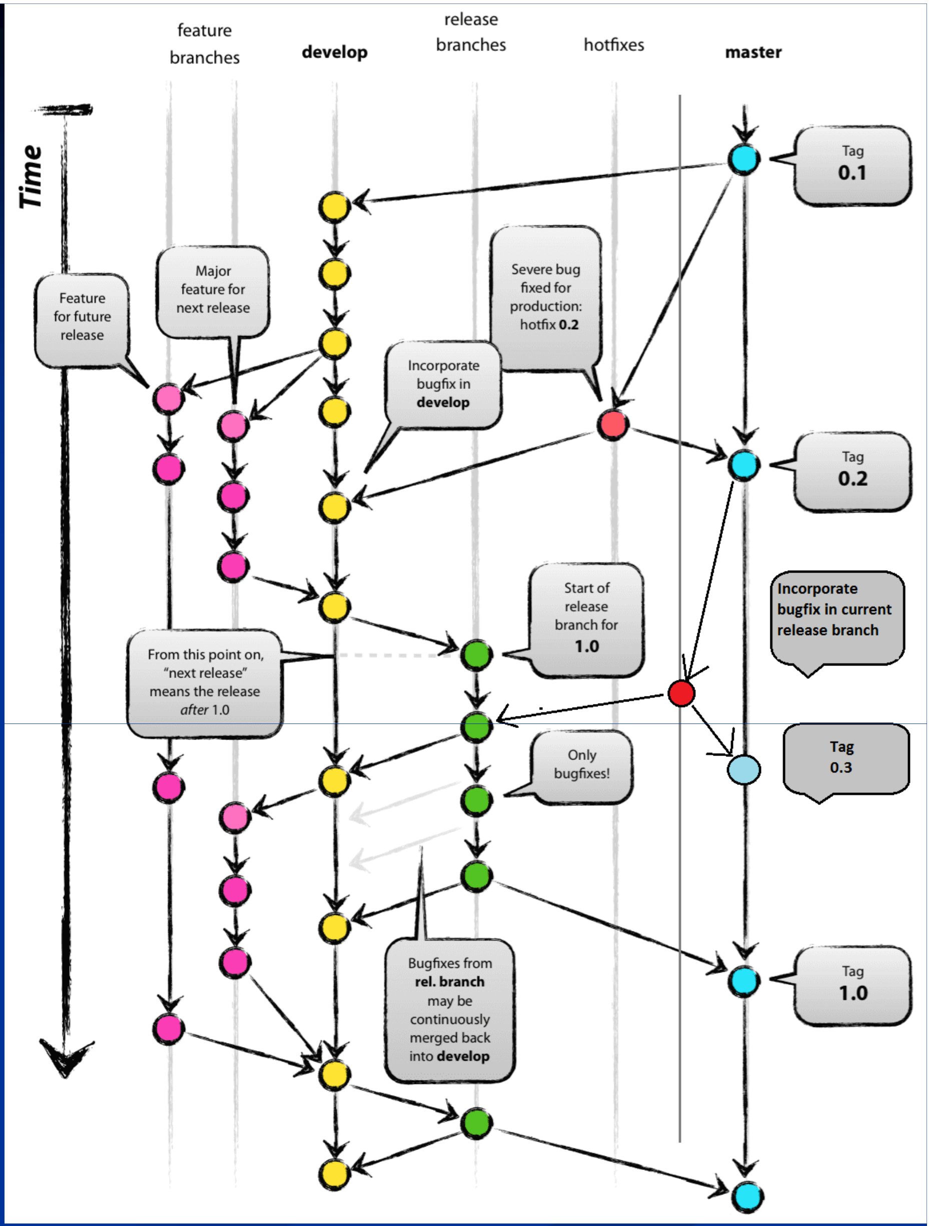
Feature Branch Workflow - example

- This process often results in a **merge commit** (except for fast forward merge, if master did not changed since branching marys-feature)
- Instead, it’s also possible to **rebase** the feature onto the tip of master before executing the merge, resulting in a fast-forward merge (and linear history)

Gitflow Workflow


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Tag 0.2
Tag 1.0
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
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

Gitflow Workflow



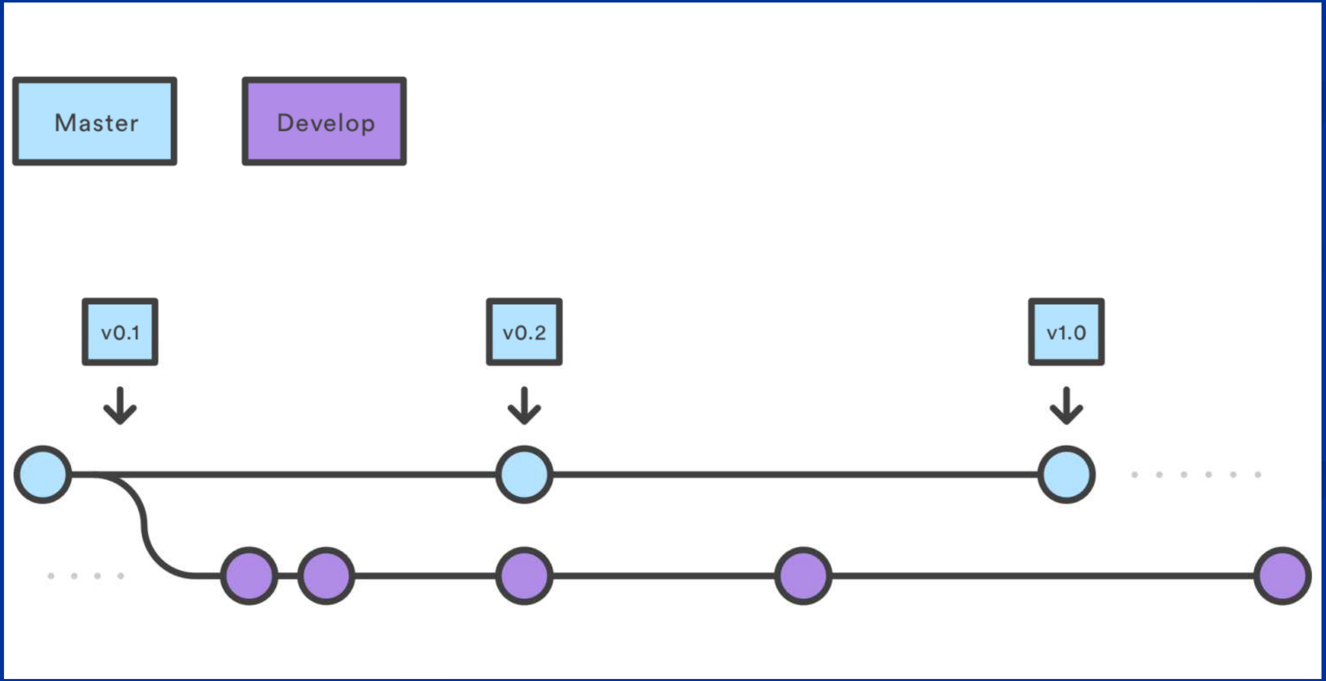
feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Incorporate bugfix in current release branch
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Tag 0.3
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Gitflow Workflow

- Proposed by Vincent Driessen at 2010
- Strict branching model designed around the project release
- Robust framework for managing larger projects
- Assigns very specific **roles** to different branches and defines how and when they should interact
- As before, one central bare repository as the communication hub for all developers

Gitflow Workflow

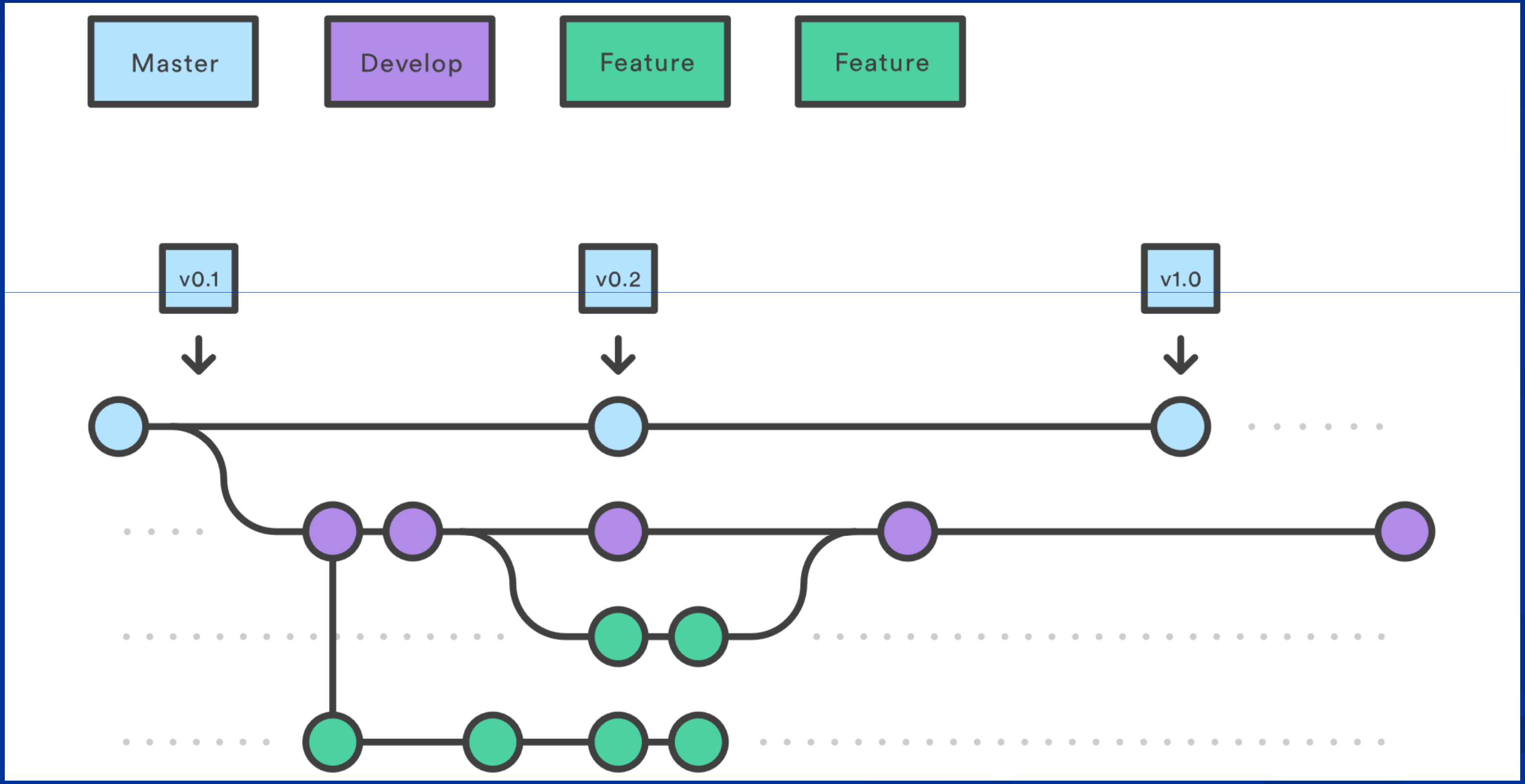
- Two „infinite” branches:
  - **master** - stores the official release history
  - **develop** - integration branch for features



Gitflow Workflow

- Each new feature should reside in its own branch, which can be pushed to the central repository for backup/collaboration.
- Feature branches branch from develop
- When a feature is complete, it gets merged back into develop
- Feature branches combined with the develop branch is, for all intents and purposes, the Feature Branch Workflow

Gitflow Workflow



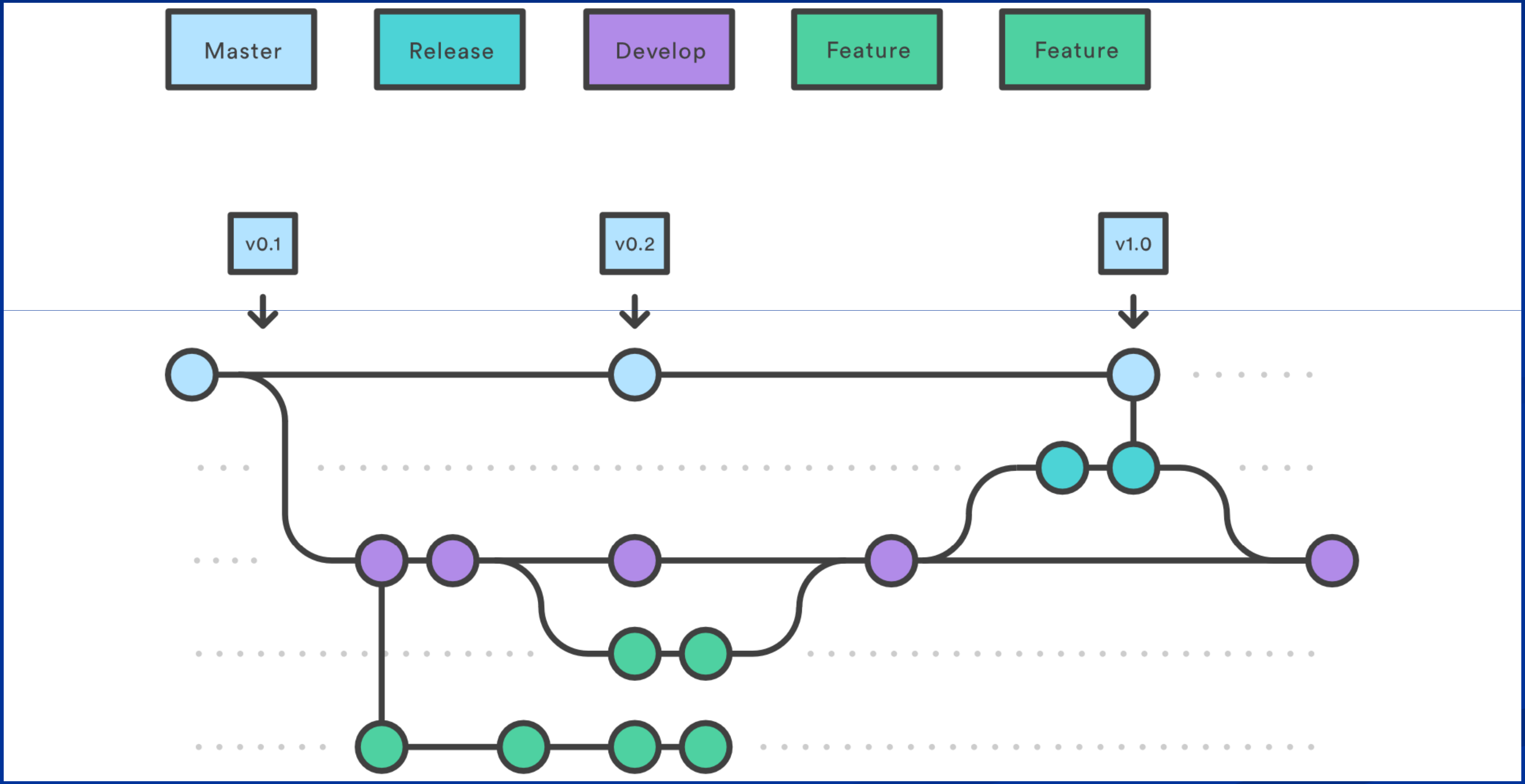
Gitflow Workflow

- Once develop has acquired enough features for a release (or a predetermined release date is approaching), a **release branch** is forked off of develop.
- Naming convention: release-* or release/*
- This 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.

Gitflow Workflow

- Once it's ready to ship, the release gets merged into master and tagged with a version number.
- It should be merged back into develop.
- 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.

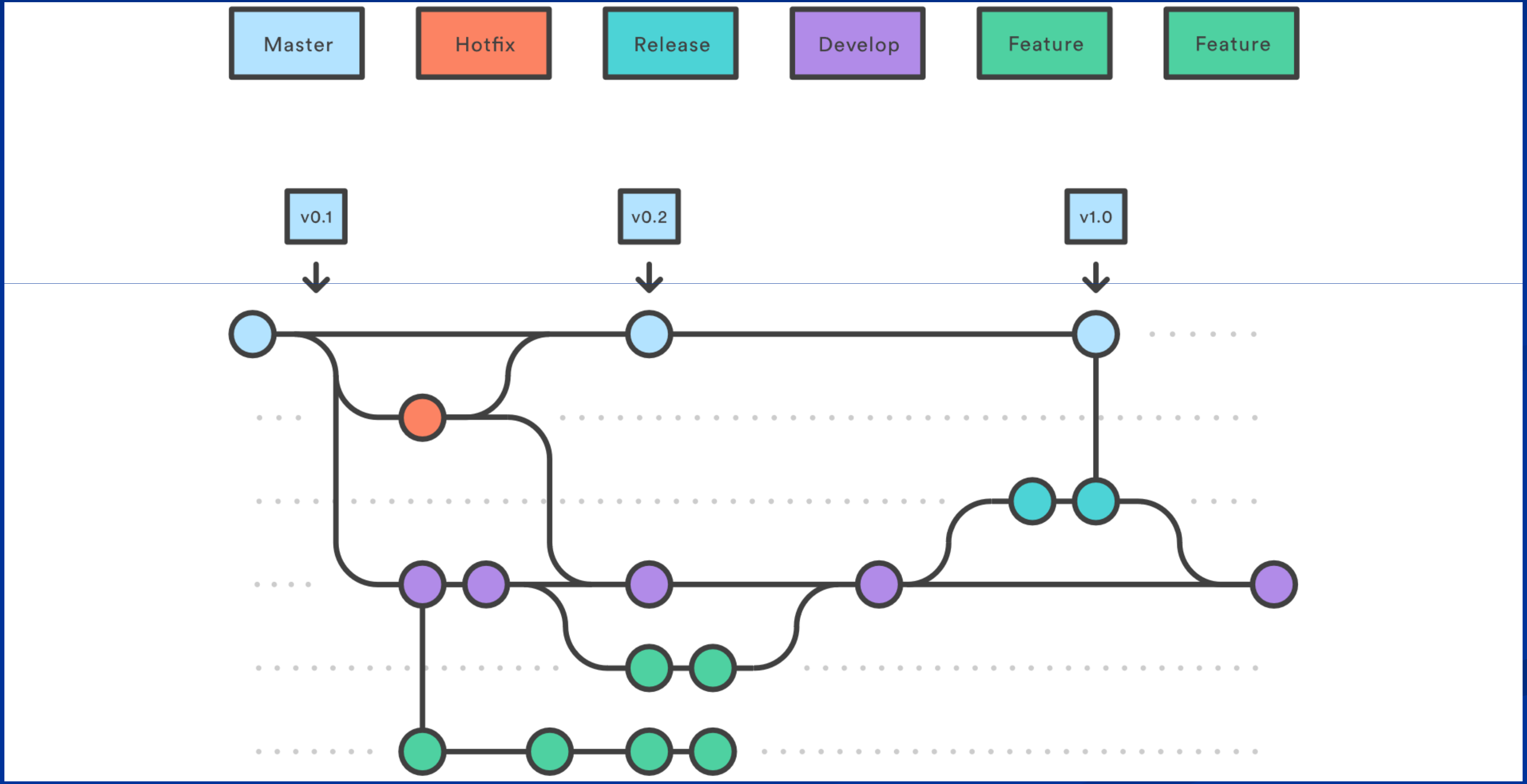
Gitflow Workflow



Gitflow Workflow

- Maintenance or "hotfix" branches are used to quickly patch production releases.
- They fork from 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.
- Maintenance branches - ad hoc release branches that work directly with master.

Gitflow Workflow



Gitflow Workflow - example

- Examples concerns a single release cycle, and assumes presence of a central repo
- 1) one developer create develop branch:

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

- 2) others clone and track develop:

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

Gitflow Workflow - example

- 3) John and Mary create separate branches for their respective features (branching from develop):

```
git checkout -b some-feature develop
```

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

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

Gitflow Workflow - example

- 5) Mary finishes her feature. She can fire a pull request or merge it into her local develop and push it to the central repository:

```
git checkout develop
git pull
git merge some-feature
git push
git branch -d some-feature
(git push -d origin some-feature)
```

Gitflow Workflow - example

- 6) Mary starts to prepare the first official release of the project:

  ```
  git checkout -b release-0.1 develop
  ```

- 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.
- As soon as Mary creates this branch and pushes it to the central repo, the release is feature-frozen.

Gitflow Workflow - example

- 7) 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.
- If Mary's organization stresses code review, this would be an ideal place for a pull request.

Gitflow Workflow - example

```
git checkout master
git pull
git merge release-0.1
git push
git checkout develop
git pull
git merge release-0.1
git push
git branch -d release-0.1
git tag -a 0.1 -m "Initial public release" master
git push --tags
```

Gitflow Workflow - example

- 8) If a bug is found in the current release, 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 issue-#001 master
# Fix the bug
git checkout master
git pull
git merge issue-#001
git push
```

Gitflow Workflow - example

- Maintenance branches contain important updates that need to be included in develop, so John needs to perform that merge as well. Then, he’s free to delete the branch:

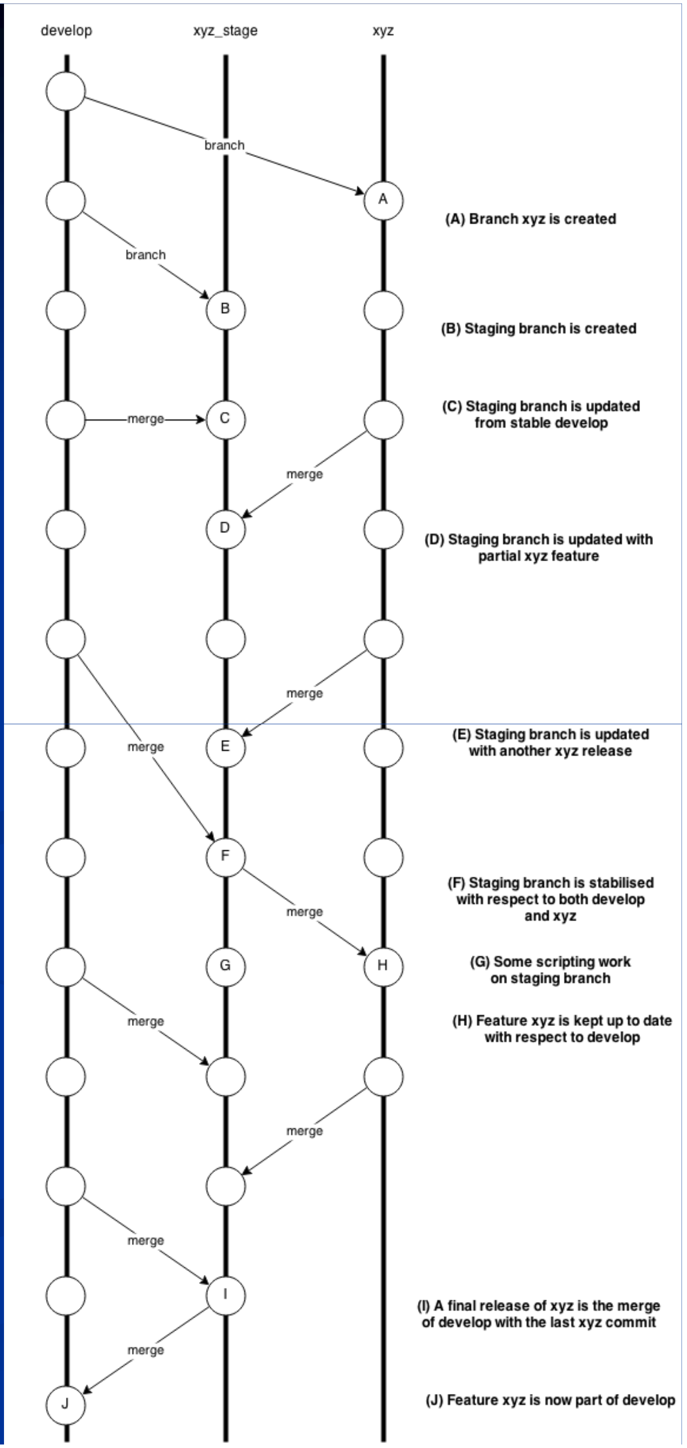
```
git checkout develop
git pull
git merge issue-#001
git push
git branch -d issue-#001
```

Gitflow Workflow - remarks

- CI on develop
- **Simple feature merge**: first develop into feature branch, then (after tests) result back into develop
- **Simple release merge**: first master into release branch, then (after tests) result back into master
- Default branch checked out from GIT repo: develop (HEAD points to the head of develop)
- Merging with --no-ff (create a merge commit even when the merge resolves as a fast-forward)
- Tag (1st &) last commit on feature/release/hotfix branch?

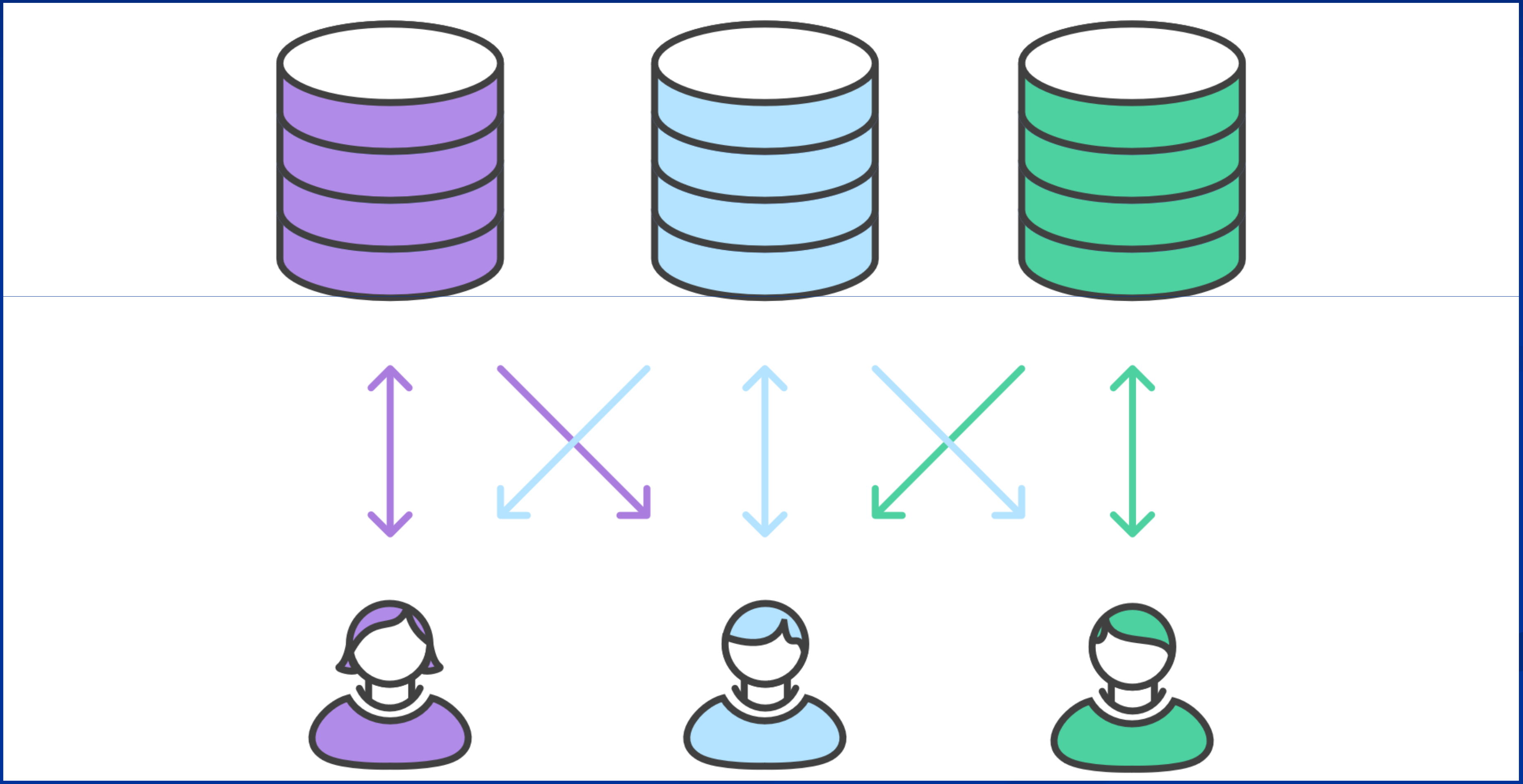
Gitflow Workflow – merging

- **Advanced feature/release merge** (with staging branch, useful, e.g., for testing)
- Tests are essential when integ-rating a branch, as even non-conflict merge may brake the code (e.g., interface change)



develop
xyz_stage
xyz
branch
A
(A) Branch xyz is created
branch
B
(B) Staging branch is created
merge
C
(C) Staging branch is updated from stable develop
merge
D
(D) Staging branch is updated with partial xyz feature
merge
E
(E) Staging branch is updated with another xyz release
merge
F
(F) Staging branch is stabilised with respect to both develop and xyz
merge
G
H
(G) Some scripting work on staging branch
(H) Feature xyz is kept up to date with respect to develop
merge
merge
merge
I
(I) A final release of xyz is the merge of develop with the last xyz commit
merge
J
(J) Feature xyz is now part of develop

Forking Workflow (Integration-Manager Workflow)



Forking Workflow

- Instead of using a single server-side repository to act as the “central” codebase, it gives *every* developer a server-side repository.
- Each contributor has not one, but two Git repositories: a private local one and a public server-side one.
- Contributions can be integrated without the need for everybody to push to a single central repository.

Forking Workflow

- Developers push to *their own* server-side repositories
- **Only the project maintainer can push to the official repository**
- Maintainer can accept commits from any developer without giving her/him write access to the official codebase
- Flexible way for large, organic teams (including untrusted third-parties) to collaborate securely

Forking Workflow

- Forking Workflow begins with an official public repository stored on a server.
- New developers **fork** the official repository to create a copy of it **on the server**.
- This new copy serves as their **personal public repository** - no other developers are allowed to push to it, but they can **pull changes** from it.
- **Cloning** of the personal public repository to get a local repo

Forking Workflow

- Local commits are pushed to own public repository – not the official one.
- Developer can fire a **pull request** with the main repository, which lets the project maintainer know that an update is ready to be integrated.
- The pull request also serves as a convenient **discussion thread** if there are issues with the contributed code.

Forking Workflow

- To integrate the feature into the official codebase, the maintainer **pulls** the contributor's changes into their local repository, checks to make sure it doesn't break the project, **merges it into his local master branch**, then **pushes** the master branch to the official repository on the server.
- The contribution is now part of the project, and other developers should **pull from the official repository** to synchronize their local repositories.

Forking Workflow

- Official repository = public repository of the project maintainer
- Personal public repositories are a convenient way to share branches with other developers
- Everybody should still be using branches to isolate individual features
- The only difference is how those branches get shared - pulled into another developer's local repo

Forking Workflow

- Convention:
  - **origin** – public private repository
  - **upstream** – public repository of the project maintainer