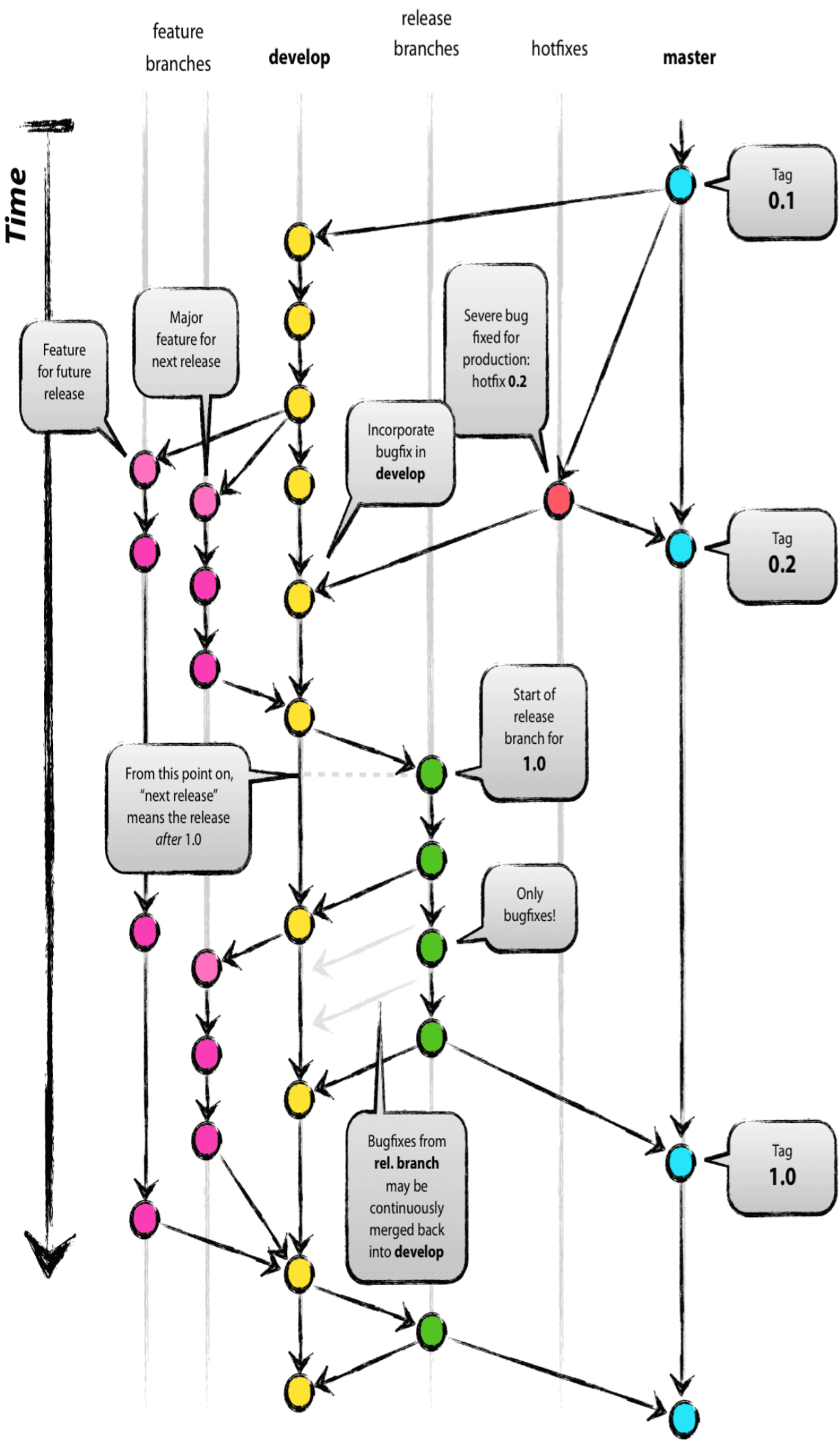
Basic Git Flow

Anthony J Souza

Content and images sourced by Vincent Driessen and Atlassian Tutorials

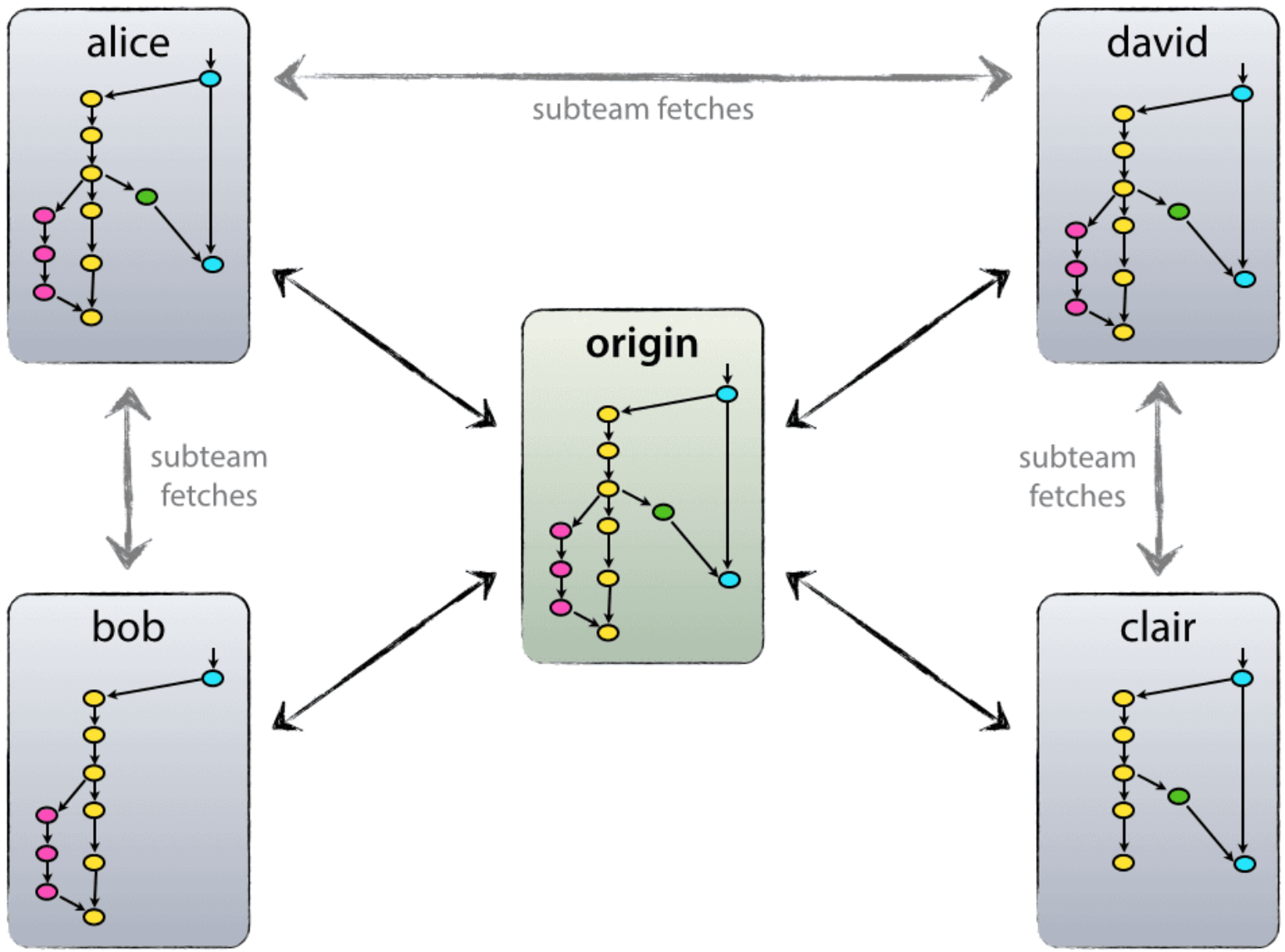
Introduction

- Git Decentralized VCS
  - Technically no main repo, but one can be assigned.
- Changed how developers think about merging and branching
- In other VCS, merging and branching can be costly.
- Git branching and merging is simple and fast


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 merged back into develop
Tag 1.0

Decentralized with a Centralized Repo

- With the git flow model being presented we have one “true” central repository.
- Note that technically there is no actual central repo, but rather we consider it our central repo at the conceptual level.
- For the next few slides we will call this repo origin.

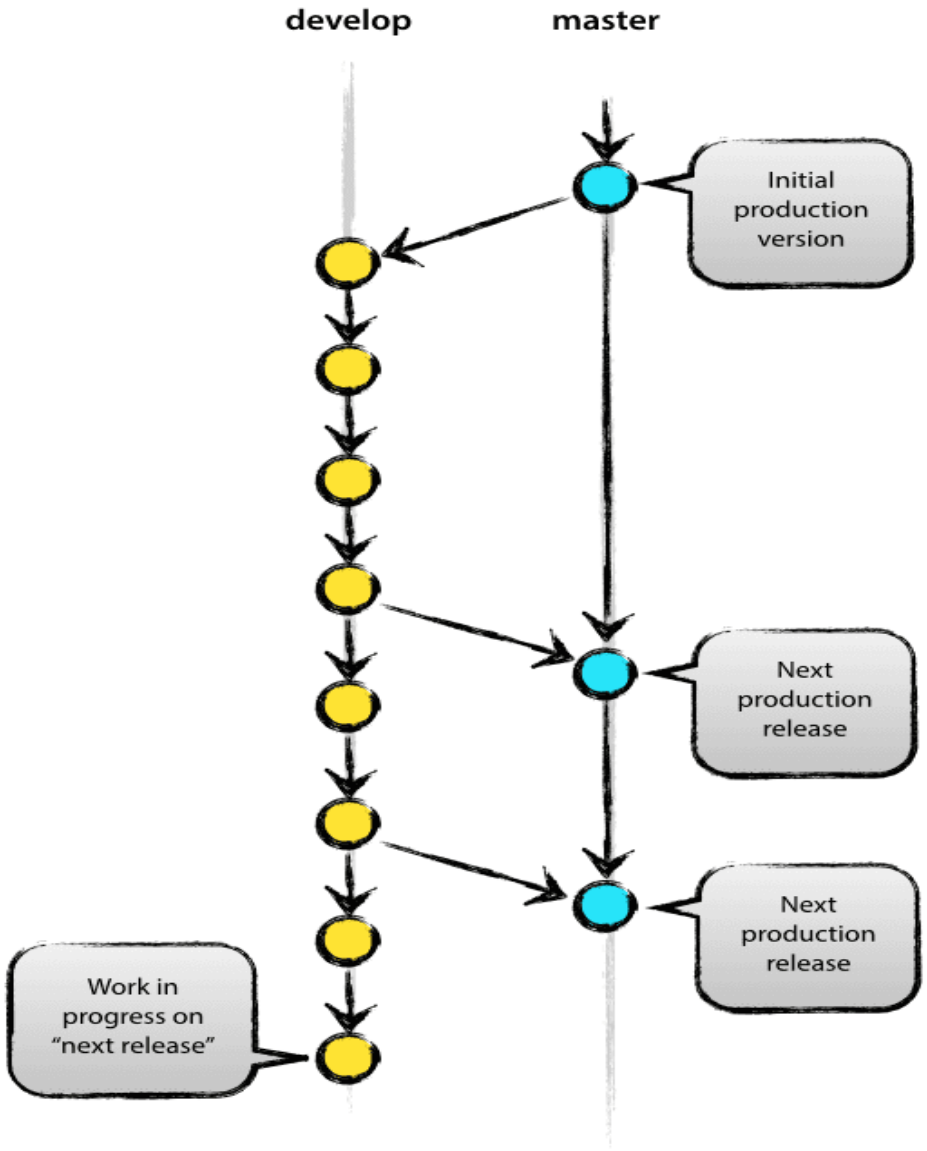

alice
david
subteam fetches
origin
subteam
fetches
subteam
fetches
bob
clair

Decentralized with a Centralized Repo

- Given the image on the previous slide, each developer has a push/pull relationship with origin.
- In addition, each developer may develop push/pull relationships with other developers' repos to form teams.
  - This can be useful if more than 1 developer is working on one big feature.
  - Work on the feature can be completed without ever pushing code to origin till it’s complete.

Main Branches

- This model was influenced by many existing models.
- It has two main branches develop and master.



Main Branch

- The master branch should be familiar to every git user
- We consider origin/master to be the main branch where the head of the branch ALWAYS reflects a production ready state of the code.
- Untested code should not be pushed here.
- Hot-fixes should be fixed here.
- Deployment or auto deployment will stem from this branch.

Develop Branch

- Develop branch will be the branch where the HEAD of the branch reflects a state of the source code where the latest delivered development changes are ready for the next release.
- Nightly automatic builds can be performed on this branch.
- Code on this branch is never directly pushed to a production environment.
- It must be merged into master first.

Master / Develop Branch

- When code on develop branch has reached a stable point it can be merged back into master.
- It should also be tagged with git tag for a version number.
- Anytime a merge is performed back into master, this is technically a new production release.
- Sticking this flow between master and develop makes using automatic build and deploy scripts simple.
- It can also be automatically triggered on a master branch merge.

Supporting Branches

- In addition to the main branches we have a few supporting branches.
- These include :
  - Feature branches
  - Release branches
  - Hotfix Branches
- Each of these branches have a specific purpose.
- They “should” follow strict rules as to:
  - Which branch they originate from
  - Which branches must be their merge targets
  - Even how they should be named.
- They are not special branches but the way we use them makes them special.

Supporting Branch – Feature Branch

- Feature branches are used to develop new features for upcoming or distant releases.
- When developing a feature, its target release maybe unknown.
- The branch for a specific feature exists if the feature is still being developed.
- Then eventually will be merged back into develop.

Supporting Branch – Feature Branch

- Feature branches may branch off from:
  - develop
- ***Must*** merge back into:
  - develop.
- Branch naming convention:
  - Anything except master, develop, release-*, hotfix-*
  - One notable suggestion is feature/featureName

- Creating a new feature branch:

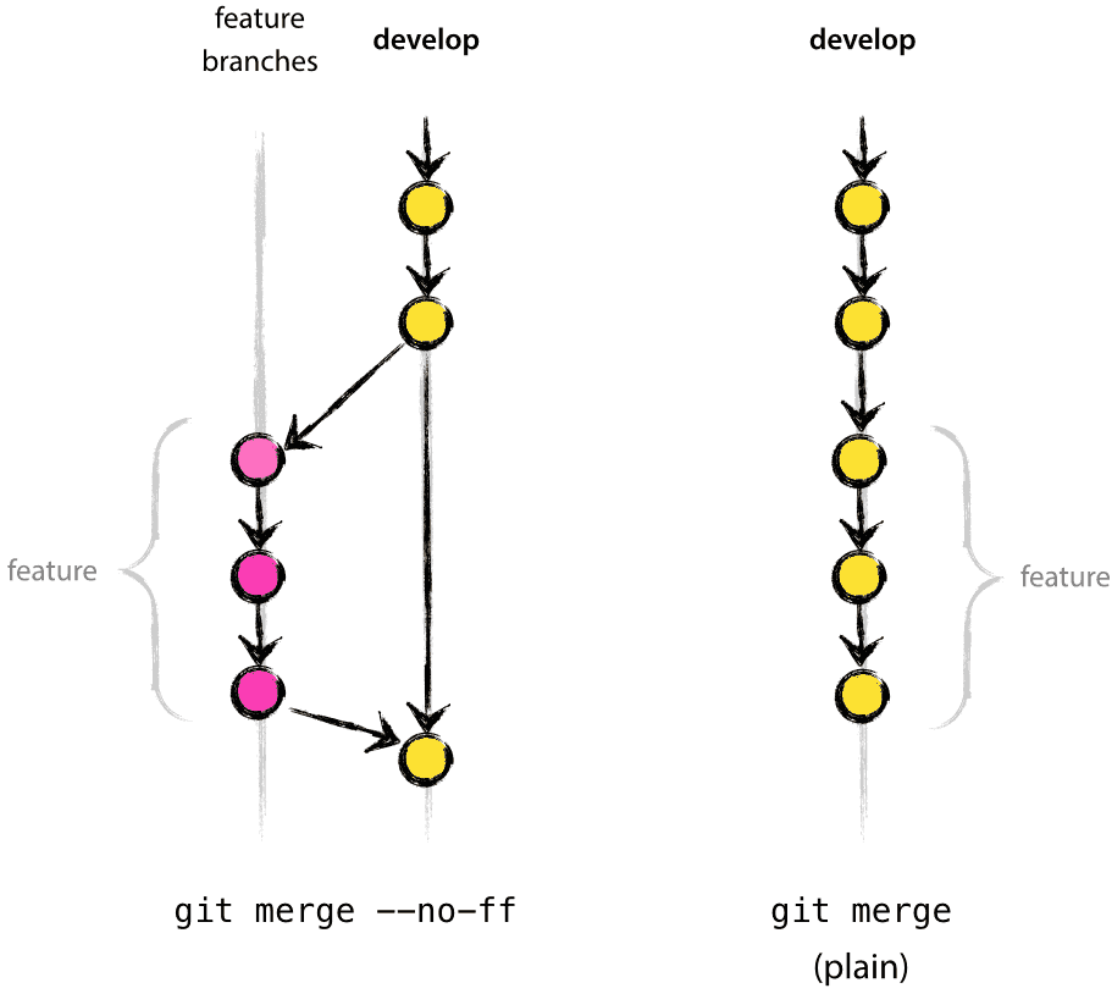
```
$ git checkout -b feature/myfeature develop
Switched to a new branch "feature/myfeature“
```

- Merging completed feature back into develop:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff feature/myfeature
Updating ea1b82a..05e9557
(Summary of changes)
$ git branch -d feature/myfeature
Deleted branch feature/myfeature (was 05e9557).
$ git push origin develop
```

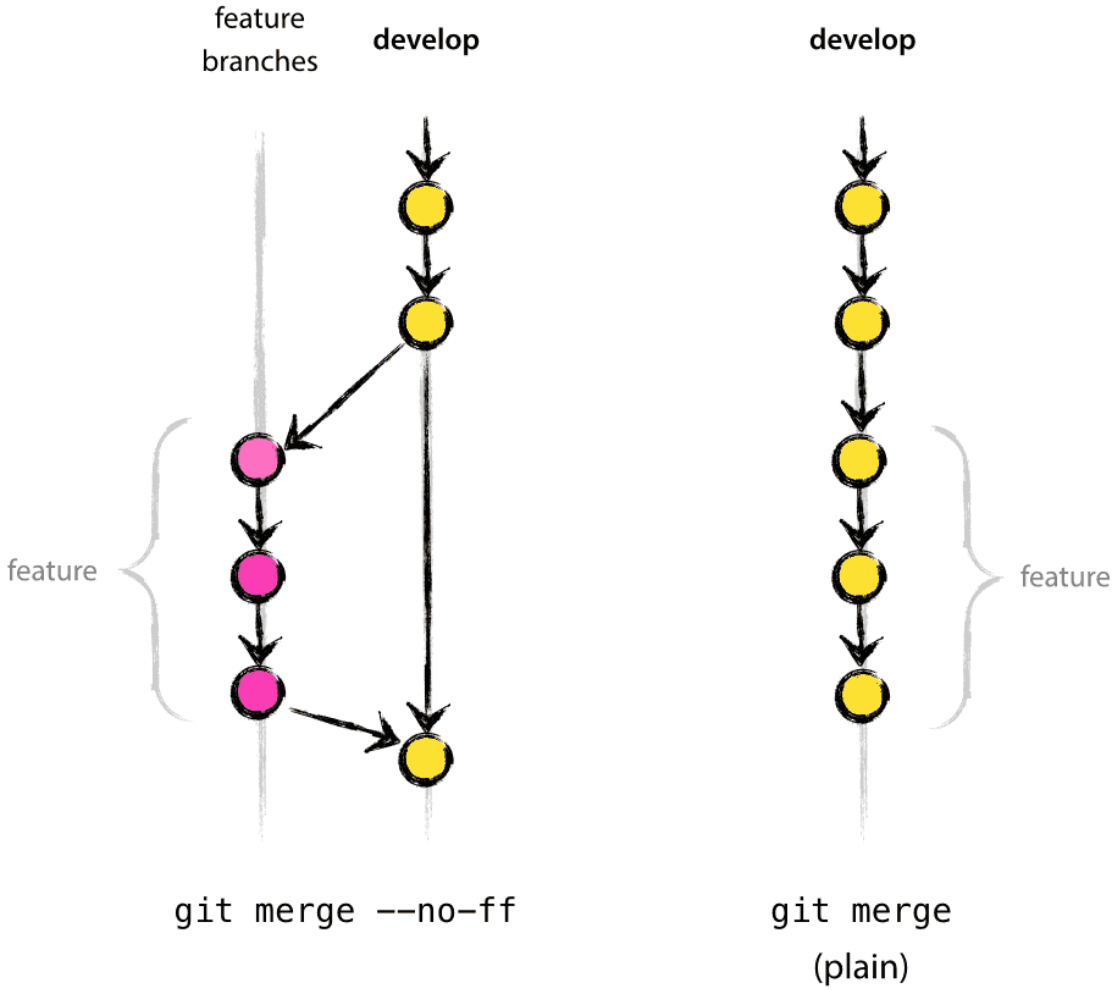
Supporting Branch - Feature

- 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.



Supporting Branch - Feature

- In the case without the `--no-ff`, it is impossible to see from Git History which of the commit objects together have implemented a given feature.
- You would need to manually read all log messages.
- Reverting entire features is tricky as well without the flag, while with the flag it is simple.



Supporting Branch – Release Branch

- Release Branches support preparation of a new production release.
- Can be used for last minute SMALL changes.
- They allow for minor bug fixes and preparing meta-data for release
  - Things like version numbers, builds dates and the like.
- By doing all this work on the release branch, the develop branch is free to work on features for the next release.
- The time to branch off a new release branch is when the develop branch is in a state where it reflects a new release.
  - Sometimes you will see these as Release Candidates as well.

Supporting Branch – Release Branch

- All features that are targeted for the current release should be merged into develop.
- All future release features should be merged into develop and may be merged AFTER the release branch is created.
- Version numbers should be assigned when creating the release branch.

Supporting Branch – Release Branch

- Release branches May branch off from:
  - develop
- ***Must*** merge back into:
  - develop and master
- Branch naming convention:
  - release-*
  - rc-x.x
  - release/x.x

Supporting Branch – Release Branch

- Creating release branch. Assume Previous Release was 1.15 and it is decided the next version release is 1.2 (not 1.6 or 2.0)

```
$ 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(-)
```

- This new branch may exist for some time. Or until the release is rolled out.
- During this time, bug fixes may be applied (NOTE : this does not happen on develop branch
- Also, adding new features here is NOT ALLOWED.

Supporting Branch – Release Branch

- Finishing a release branch can be done when the state of the branch is at a production level.
- First, we merge release into master
- Then tag the commit with the releaser version number.
  - Also need to merged any changes made on release back into develop as well.

- Merging Release into Master:

```
$ 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
```

- Merging changes into Develop:

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-1.2
Merge made by recursive.
(Summary of changes)
```

- Merging release-* back into develop may cause merge conflicts, these should be small and easy to handle.
- Note : You may use the -s or -u <key> flags to sign your tag cryptographically.

Supporting Branch – Release Branch

- When we are finished with the current release branch, we can delete it.
- This can be done with feature branches as well.

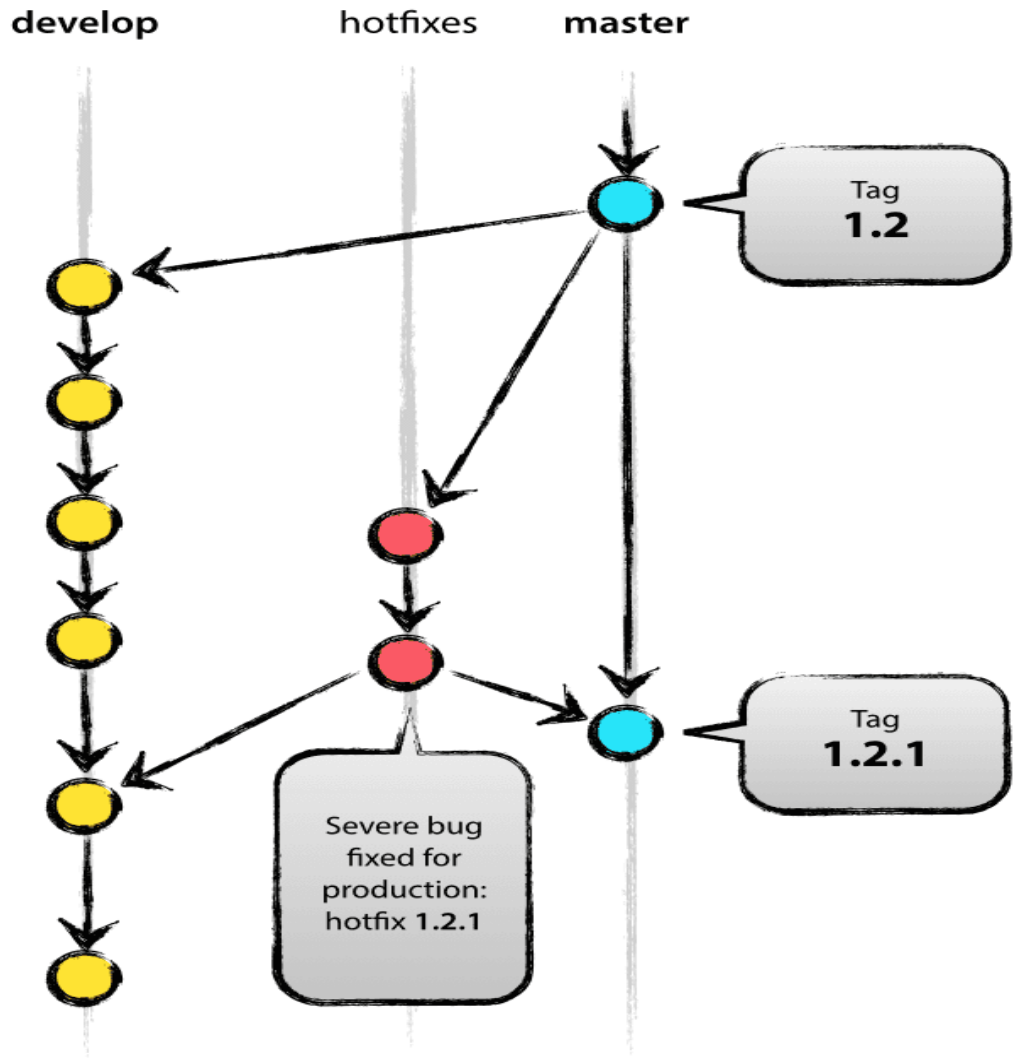
```
$ git branch -d release-1.2
Deleted branch release-1.2 (was ff452fe).
```

Supporting Branch – Hotfix Branch

- Hotfix branches are very similar to releases branches
- They are meant to prepare for a new production release, but these can be unplanned.
- They are created from the necessity to act quickly upon an undesired state of a live production version.
- When a critical bug in a production version must resolved quickly, a hotfix branch may be branched off from the corresponding tag on the master branch.
- The idea, is that the team can continue work on the develop branch, while another team member can work on preparing a quick production fix.

Supporting Branch – Hotfix Branch

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



Supporting Branch – Hotfix Branch

- Creating hotfix branch:

```
$ 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.
  - For example if the current version is 1.2, the next version can be 1.2.1.

Supporting Branch – Hotfix Branch

- Committing bug fix to hotfix branch:

```
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 abbe5d6] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)
```

Supporting Branch – Hotfix Branch

- When finished, the bugfix needs to be merged back into *master* , but also needs to merged back into *develop*
  - This is to ensure that the bugfix is included in the next release as well.
- Process is like how release branches are handled when we are done with them.

- Merge hotfix into master

```
$ 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
```

- Merge hotfix into develop

```
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
```

Supporting Branch – Hotfix Branch

- The one exception to the rule, 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.

Supporting Branch – Hotfix Branch

- Finally we can remove or hotfix-* branch:

```
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was abbe5d6).
```

Master / Develop Branches and GitHub

- Now since these two branches are deemed special by the developers, extra measures may be needed to protect their integrity.
- One feature GitHub offers is the ability to lock down branches so only certain developers may push/merge into these branches.
- This can be important because it cans ensure a unified upstream of merges and doesn’t allow N developers to push to master/develop.
- In the next few slides, it will be illustrated how this can be beneficial in keeping your git repository and git history clean and easy to read.

Master / Develop Branches and GitHub

- For the next few slides lets pretend we are a team of developers with the following titles:
  - Team lead
  - GitHub Master
  - Back-End Lead
  - Front-End Lead
  - Front-End Dev
  - Back-End Dev
- The Back-end Lead and GitHub Master has push privileges to Master
- Back and Front-end Leads have push privileges to Develop
- All other developers only have push permissions to the features branches they create.
  - Must make a pull request to merge code into develop.
  - They also should NEVER make a pull request to master from their feature branch.

Master / Develop Branches and GitHub

- Leads develop a list of tasks
- Team lead delegates task assignment to Front and Back-end Leads.
- Front-end Lead assigns task of developing Log in page to **fed1**
- Back-end Lead assigns task of developing user log in server-side code to **bed1**.
- Currently, both **fed1** and **bed1** have their tasks and create their branches off develop.
  - **fed1** names their branch feature/login-view
  - **bed1** names their branch feature/login-controller

Master / Develop Branches and GitHub

- **fed1** finishes their code and runs the testing suite made for their project.
  - Since this is front-end work something like MochaJS can be used.
  - These can be unit tests
- If all test pass, **fed1** makes a pull request to develop.
- After the pull request is made **Front-End Lead** performs a code review.
  - If any changes are needed to be made, comments can be made on the pull request.
  - **fed1** can add more commits if needed based on feedback to the feature branch which will automatically be added to the pull request
- If all tests pass and code review is OK, so **Front-End Lead** merges code into develop.

Master / Develop Branches and GitHub

- **bed1** finishes their code and runs the testing suite made for their project.
  - Since this is back-end work, again something like MochaJS can be used.
  - These can be unit tests
- If all test pass, bed1 makes a pull request to develop.
- After the pull request is made **Back-End Lead** performs a code review.
  - If any changes are needed to be made, comments can be made on the pull request.
  - **bed1** can add more commits if needed based on feedback to the feature branch which will automatically be added to the pull request
- If all tests pass and code review is OK, so **Back-End Lead** merges code into develop.

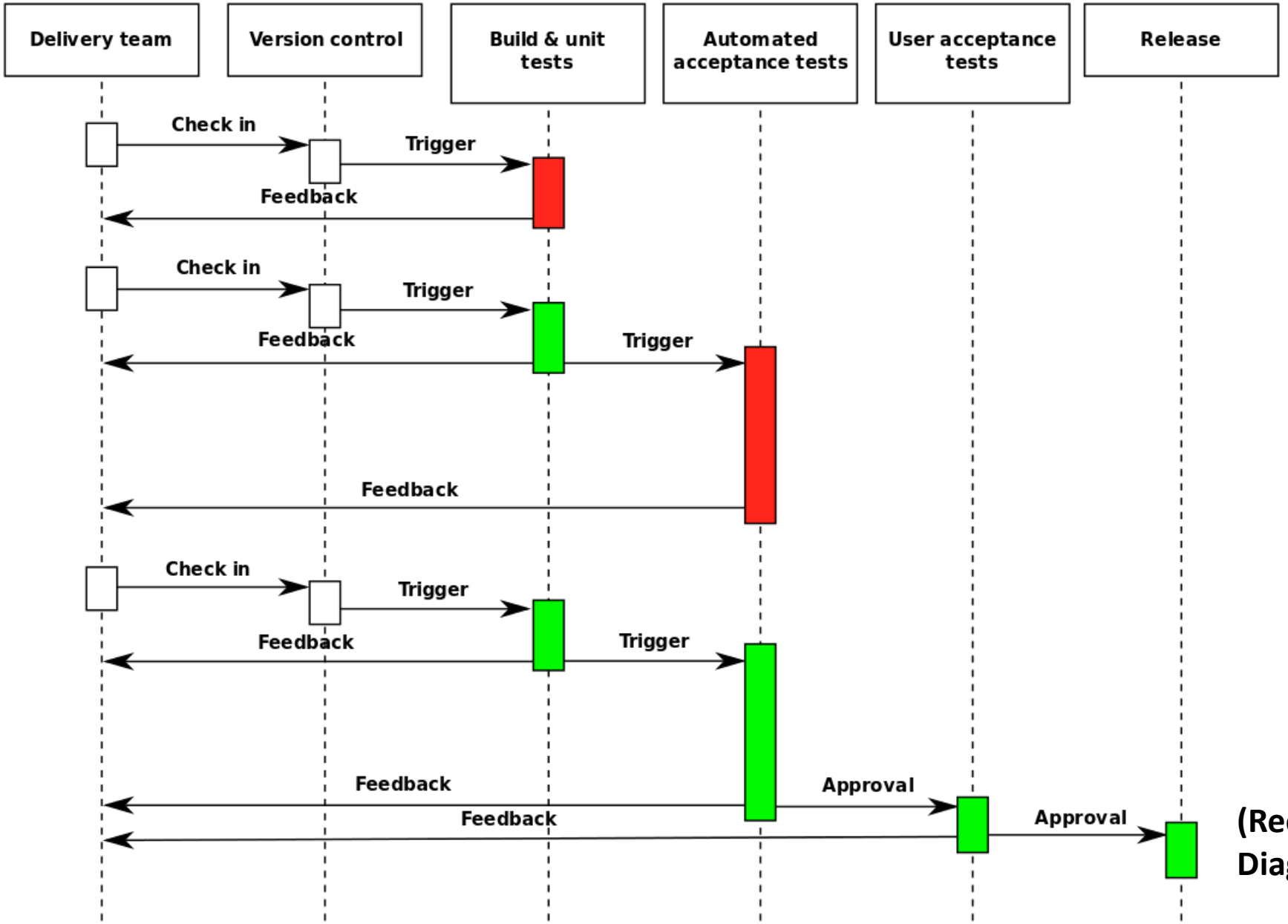
Master / Develop Branches and GitHub

- Now, with **fed1** and **bed1** work on the develop branch.
- **Back-End Lead** can perform more testing.
  - These can include regression testing , integration testing, Systems testing and Acceptance Testing for example.
- If all test pass **Back-End Lead** can create a release branch with the current state of code.
  - We will name this branch **ourapp/rc-1.3**
- On this branch more testing and code review can be performed.
- You will also do other release related tasks like minor bug fixes and documentation generation.

Master / Develop Branches and GitHub

- After the release branch has been reviewed and tested, the **GitHub Master** can begin the process of merging the release branch into the master (production) branch.
- This signals a new release of code that is ready customers to use.
- The release branch can be deleted after the merge if this is the choice of the team managing the repo.
  - Not all release branches are deleted after a merge. Think of software that has versions that are marked as LTS (Long Term Support)
  - Sometimes these LTS version can overlap as well.

https://en.wikipedia.org/wiki/Continuous_delivery



(Recall UML sequence Diagrams?)

References

- A successful Git branching model
- Git Book
- Feature branch Workflow
- Branching Strategies
- GitHub Best Practices