Typical Workflows, Definitions, and Examples

ATPESC 2018

Jared O'Neal
Mathematics and Computer Science Division
Argonne National Laboratory

Q Center, St. Charles, IL (USA)
July 29 – August 10, 2018

License, citation, and acknowledgments

License and Citation

- Requested citation: Jared O'Neal, Typical Workflows, Definitions, and Examples, tutorial, in Argonne Training Program on Extreme-Scale Computing (ATPESC) 2018. DOI: 10.6084/m9.figshare.6970502.

Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357
- Anshu Dubey, Klaus Weide, Saurabh Chawdhary, and Carlo Graziani
- Iulian Grindeanu

Goals

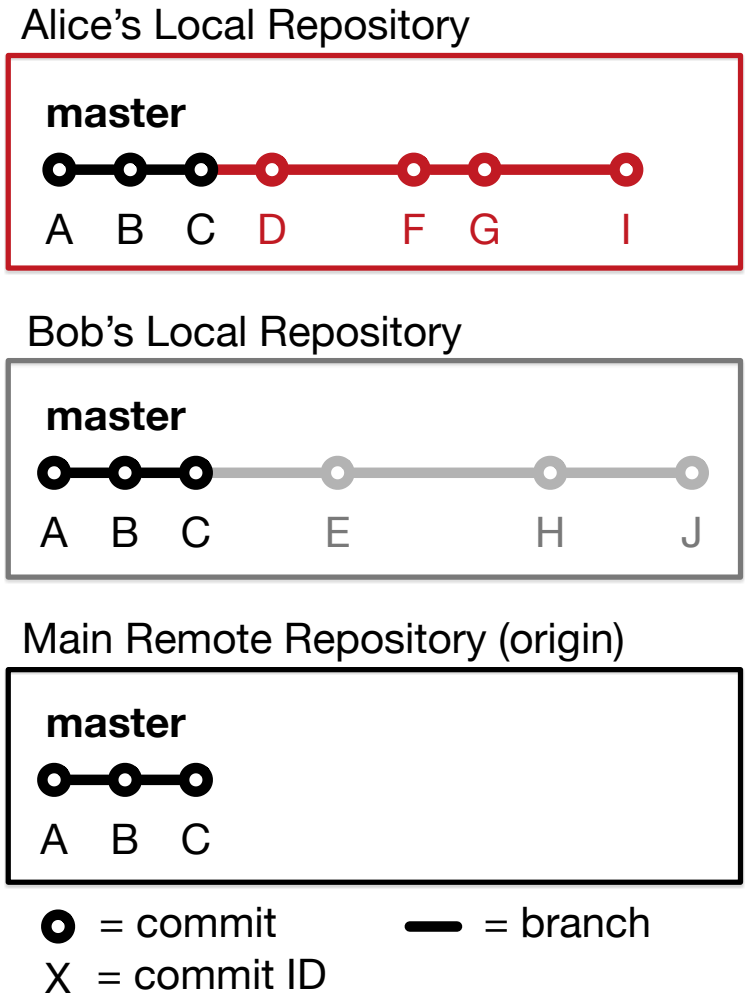
Development teams would like to use version control to collaborate productively and ensure correct code

- Understand challenges related to parallel code development *via* distributed version control
- Understand extra dimensions of distributed version control & how to use them
  - Local vs. remote repositories
  - Branches
  - Issues, Pull Requests, & Code Reviews (next talk)
- Exposure to workflows of different complexity
- What to think about when evaluating different workflows
- Motivate continuous integration

Distributed Version Control System (DVCS)

Two developers collaborating *via* Git

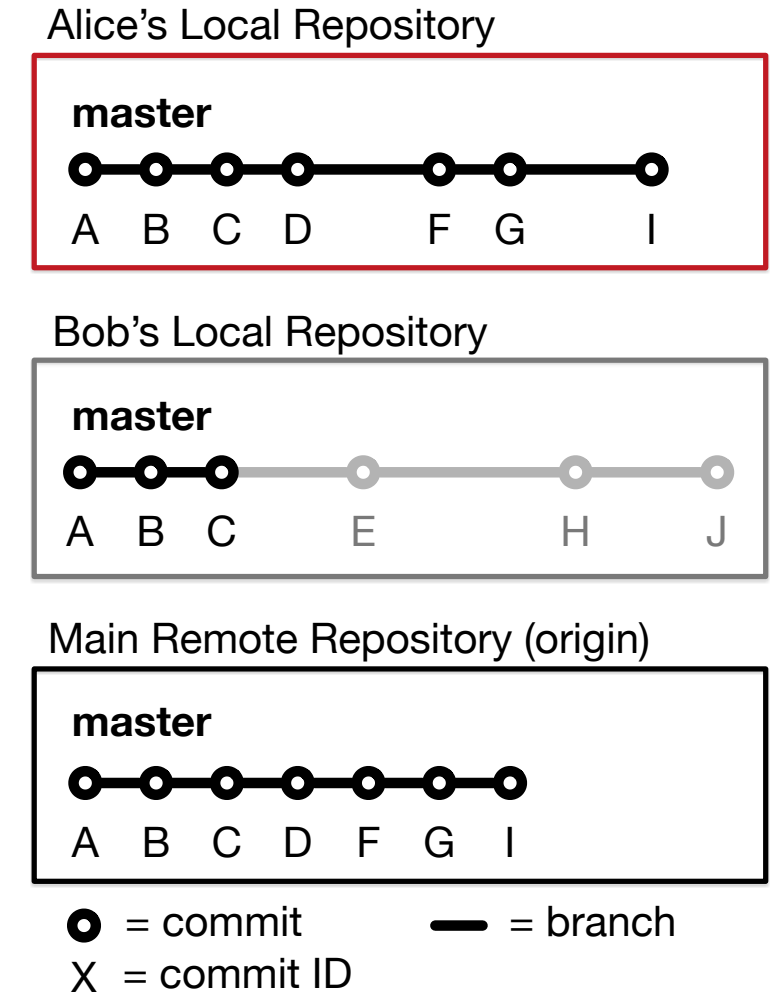
- Local copies of master branch synched to origin
- Each develops on **local** copy of master branch
- All copies of master immediately diverge
- How to **integrate** work on origin?



DVCS Race Condition

Integration of independent work occurs when local repos interact with remote repo

- Alice pushes her local commits to remote repo first
- No integration conflicts
- No risk
- Alice's local repo identical to remote repo

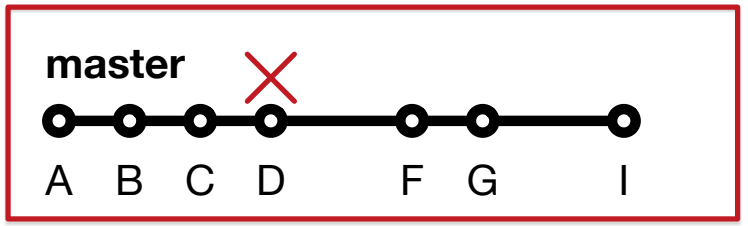


Integration Conflicts Happen

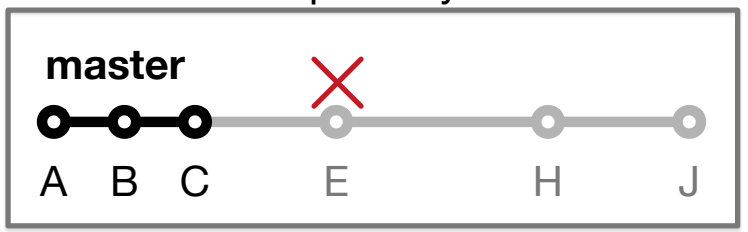
Bob's push to remote repo is rejected

- Alice updated code in commit D
- Bob updated same code in commit E
- Alice and Bob need to study conflict and decide on resolution at pull (time-consuming)
- Possibility of introducing bug on master branch (risky)

Alice's Local Repository



Bob's Local Repository



loops.cpp (commit C)

```
36
37      // TODO: Code very important loop here ASAP
38
39
40              ...
41
42
43      // TODO: Code other very important loop here ASAP
44
```

loops.cpp (commit D)

```
36
37      // Very important loop
38      for (int i=0; i<N; ++i) {
39
40              ...
41
42      // Another very important loop
43      for (int i=1; i<=N; ++i) {
44          foo[i] = bar[i] * i;
45
```

loops.cpp (commit E)

```
36
37      // Very important loop
38      for (int i=0; i<N; i++) {
39
40              ...
41
42      // Another very important loop
43      for (int i=0; i<N; i++) {
44          foo[i] = bar[i] * i;
45
```

Our First Workflow

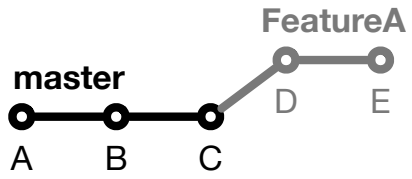
This process of collaborating *via* Git is called the **Centralized Workflow**

- See Atlassian/BitBucket for more information
- “Simple” to learn and “easy” to use
- Leverages local vs. remote repo dimension
  - Integration in local repo when local repos interact with remote repo
- What if you have many team members?
- What if developers only push once a month?
- What if team members works on different parts of the code?
- Working directly on master

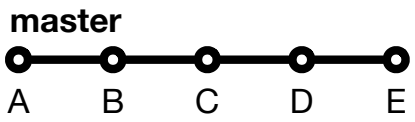
Branches

Branches are independent lines of development

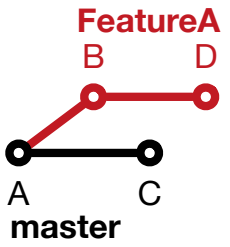
- Use branches to protect master branch
- Feature branches
  - Organize a new feature as a sequence of related commits in a branch
- Branches are usually combined or **merged**
- Develop on a branch, test on the branch, and merge into master
- Integration occurs at merge commits



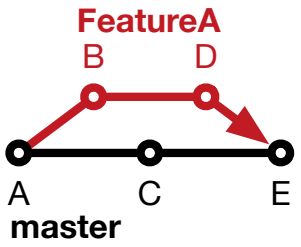
Fast-Forward



No Merge



Divergence

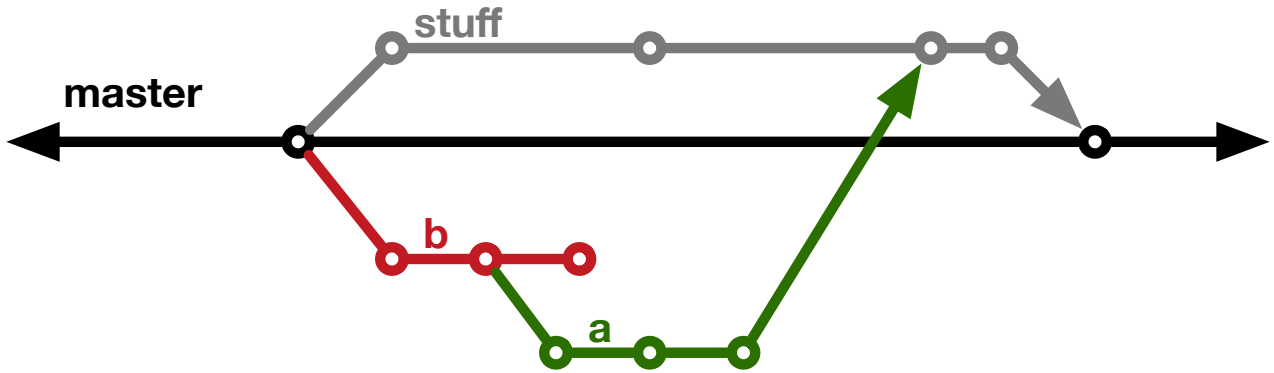


Merge Commit

Control Branch Complexity

Workflow policy is needed

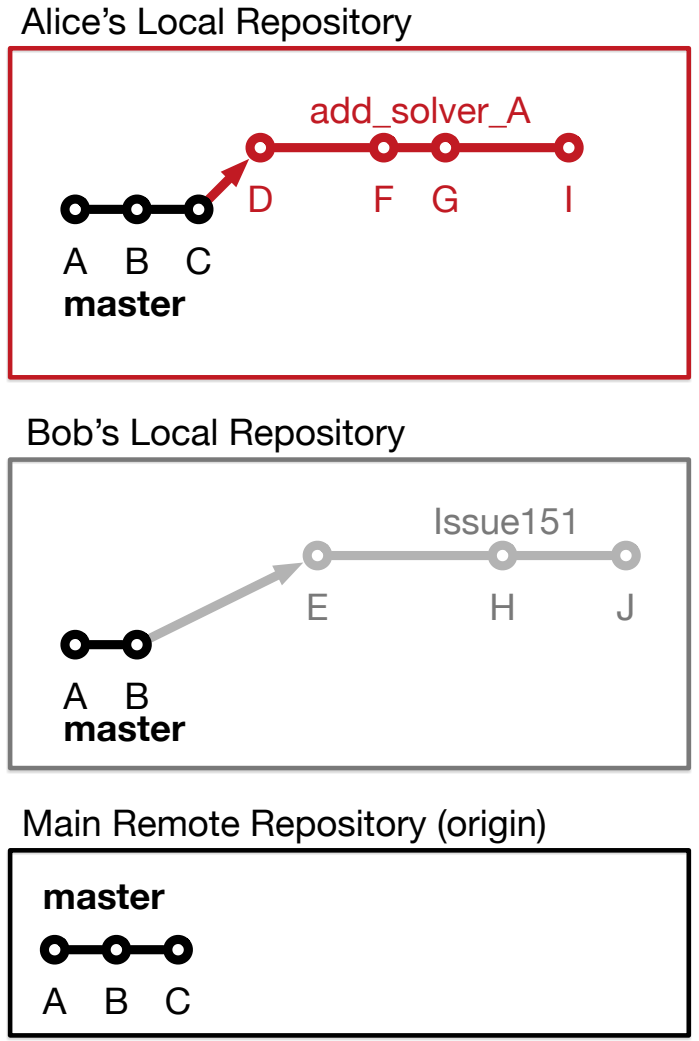
- Descriptive names or linked to issue tracking system
- Where do branches start and end?
- Can multiple people work on one branch?



Feature Branches

Extend Centralized Workflow

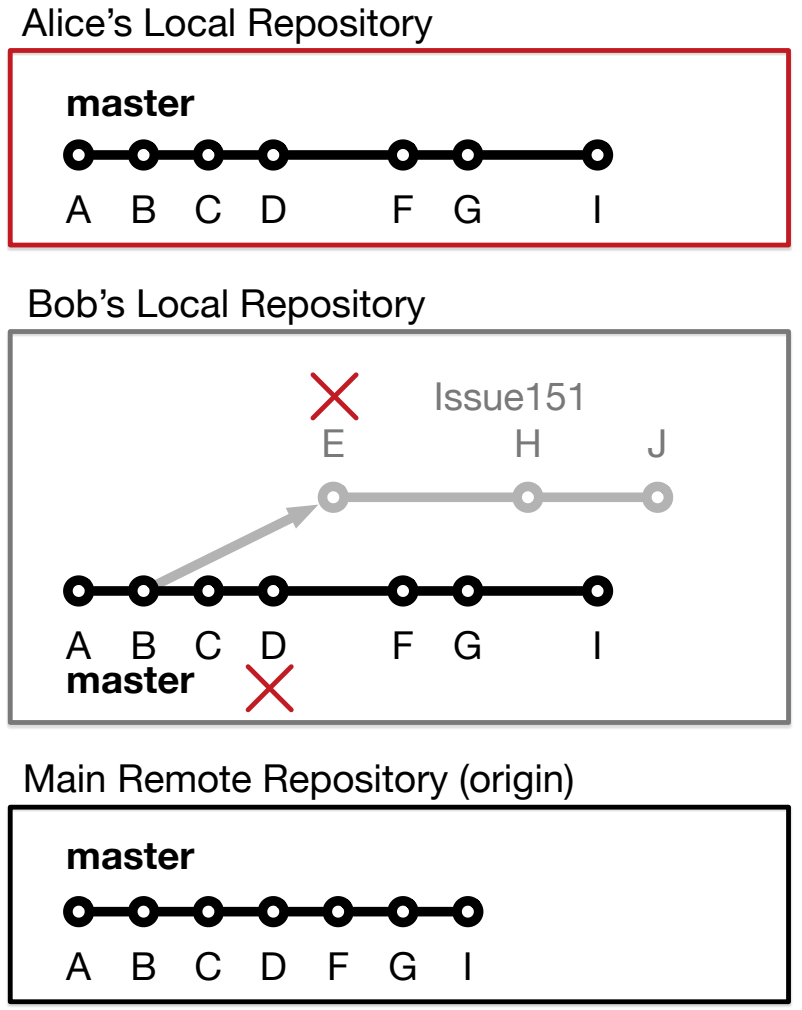
- Remote repo has commits A & B
- Bob pulls remote to synchronize local repo to remote
- Bob creates local feature branch based on commit B
- Commit C pushed to remote repo
- Alice pulls remote to synchronize local repo to remote
- Alice creates local feature branch based on commit C
- Both develop independently on local feature branches



Feature Branch Divergence

Alice integrates first without issue

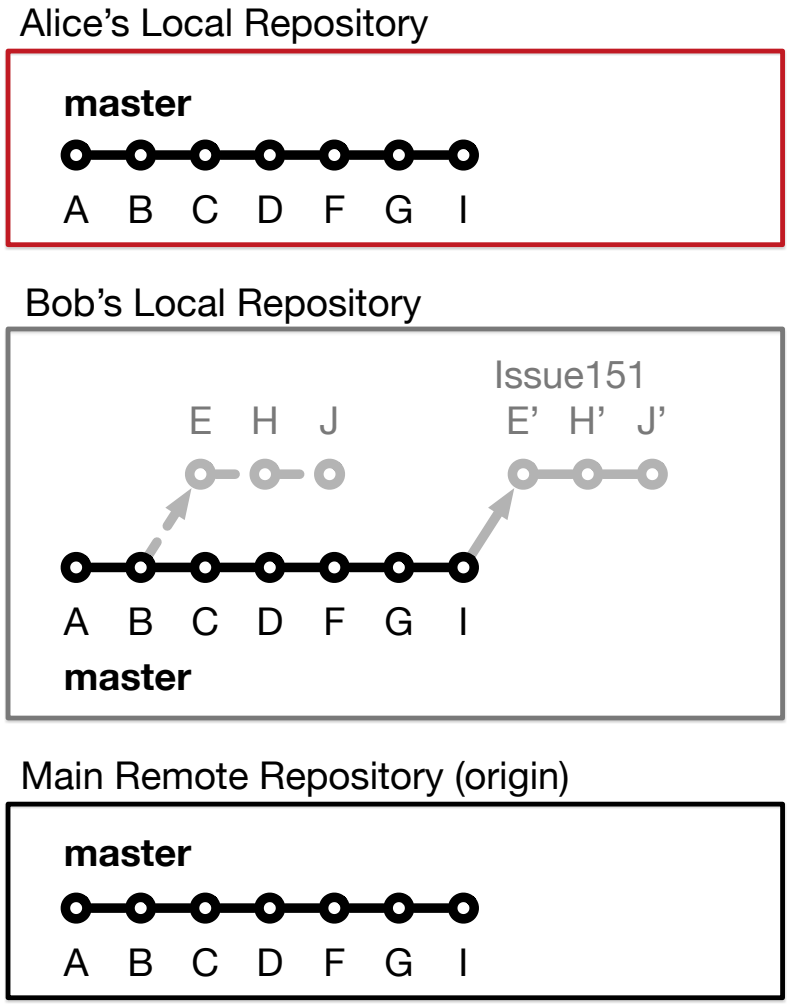
- Alice does fast-forward merge to local master
- Alice deletes local feature branch
- Alice pushes master to remote
- Meanwhile, Bob pulls master from remote and finds Alice's changes
- Merge conflict between commits D and E



Feature Race Condition

Integration occurs on Bob's local repo

- Bob laments not having fast-forward merge
- Bob **rebases** local feature branch to latest commit on master
  - E based off of commit B
  - E' based off of Alice's commit I
  - E' is E integrated with commits C, D, F, G, I
- Merge conflict resolved by Bob & Alice on Bob's local branch when converting commit E into E'
- Can test on feature branch and merge easily and cleanly



Feature Branches Summary

- Multiple, parallel lines of development possible on single local repo
- Easily maintain local master up-to-date and useable
- Integration with rebase on local repo is safe and can be aborted
- Testing before updating local and remote master branches
- Rebase is advanced Git command
  - Rebase can cause complications and should be used carefully.
- Hide actual workflow
  - History in repo is not represent actual development history
  - Less communication
  - Fewer back-ups using remote repo
- Does it scale with team size? What if team integrates frequently?
- Commits on master can be broken
- See Atlassian/BitBucket for a richer Feature Branch Workflow

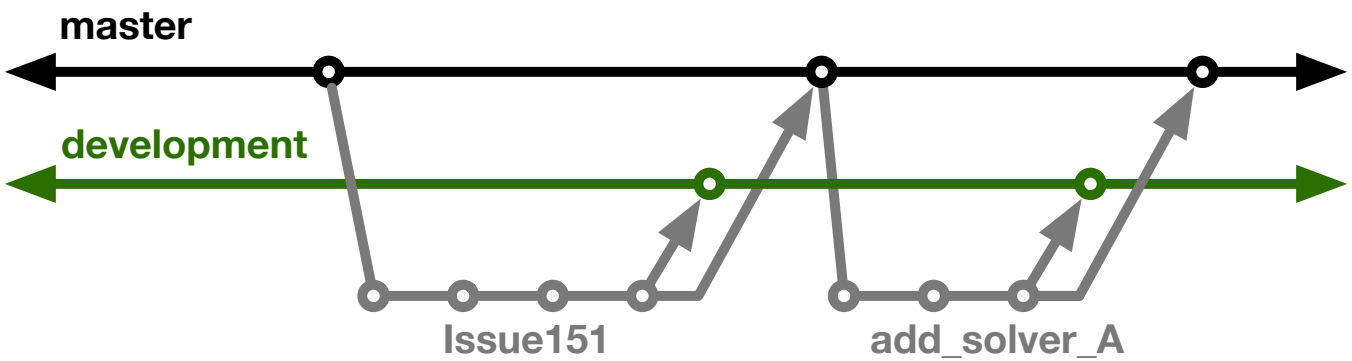
More Branches

Branches with infinite lifetime

- Base off of master branch
- Exist in all copies of a repository
- Each provides a distinct **environment**
  - Development vs. pre-production

For this example,

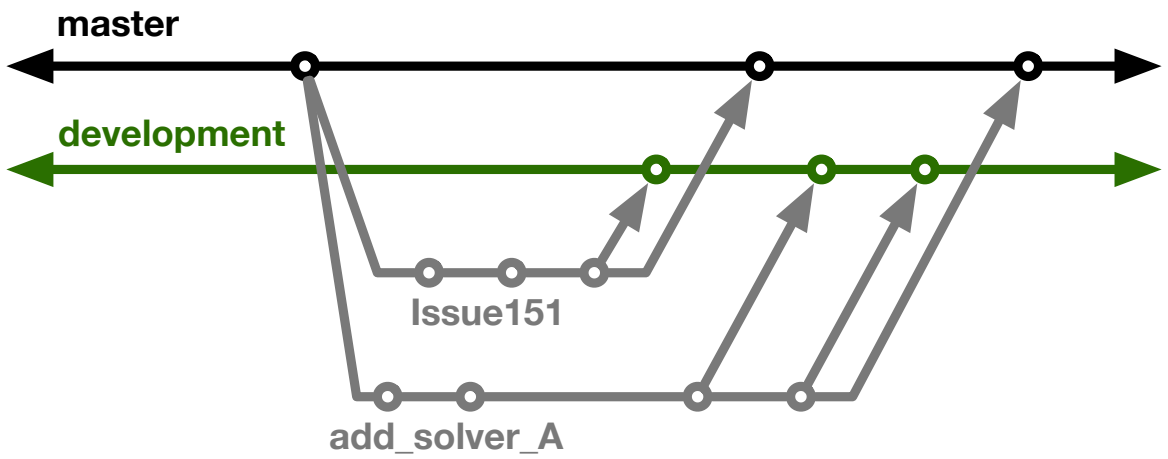
- All feature branches start and end on master
- Merge into development before merging into master
- No integration happening



Challenges

Multiple feature branches developed in parallel

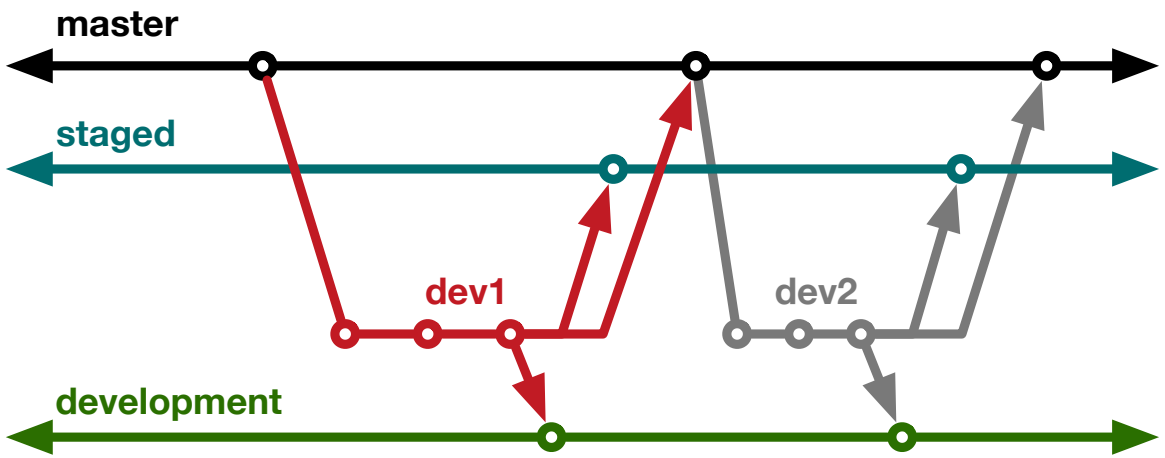
- All commits in master are in development
- Merge conflicts first exposed on development
- Set workflow so that infinite branches don’t diverge



Current FLASH5 Workflow

Test-driven workflow

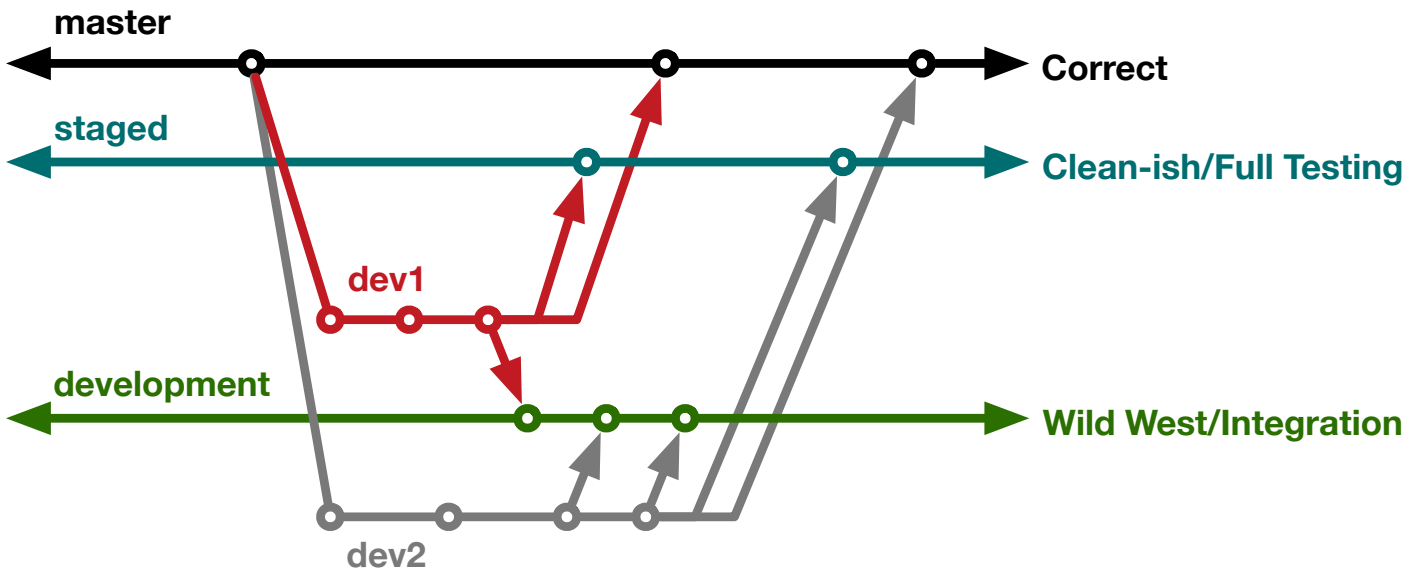
- Feature branches start and end with master
- All feature branches are merged into development for integration & manual testing
- All feature branches are then merged into staged for full, automated testing



More Branch Rules

Is staged really necessary?

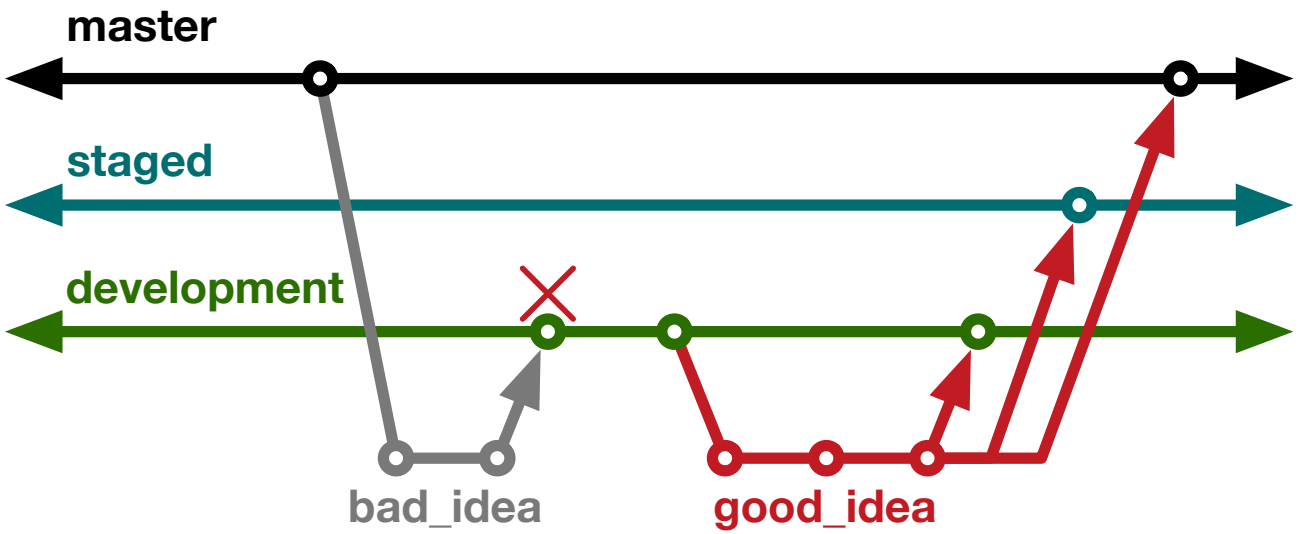
- Contains only changes intended for master
- No integration means cleaner branch
- Allows for extra stage of testing with more tests
- Extra buffer for protecting master branch



Branch Rules

Why base feature branches off master?

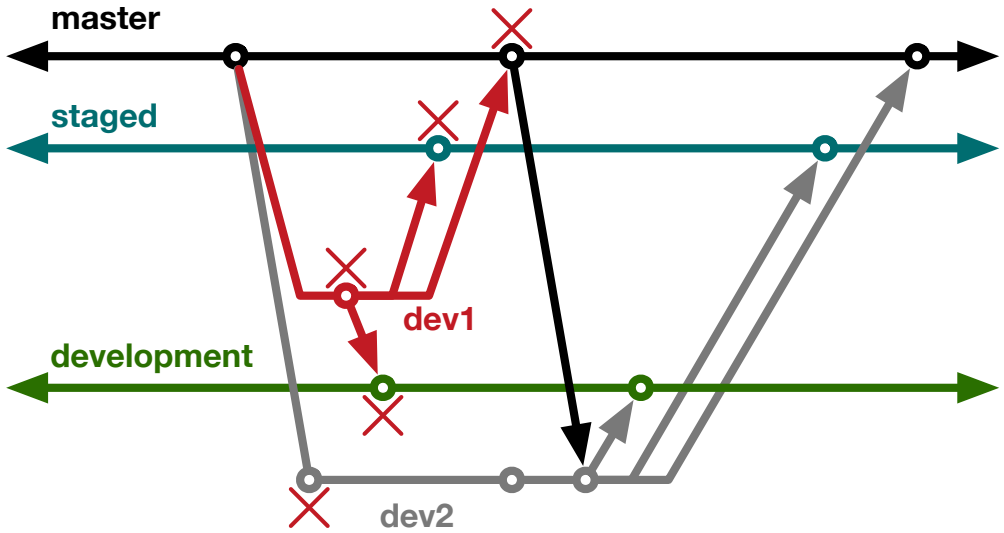
- Start from correct, verified commit
- Clean and simple to learn/enforce
- Isolate master from integration environment



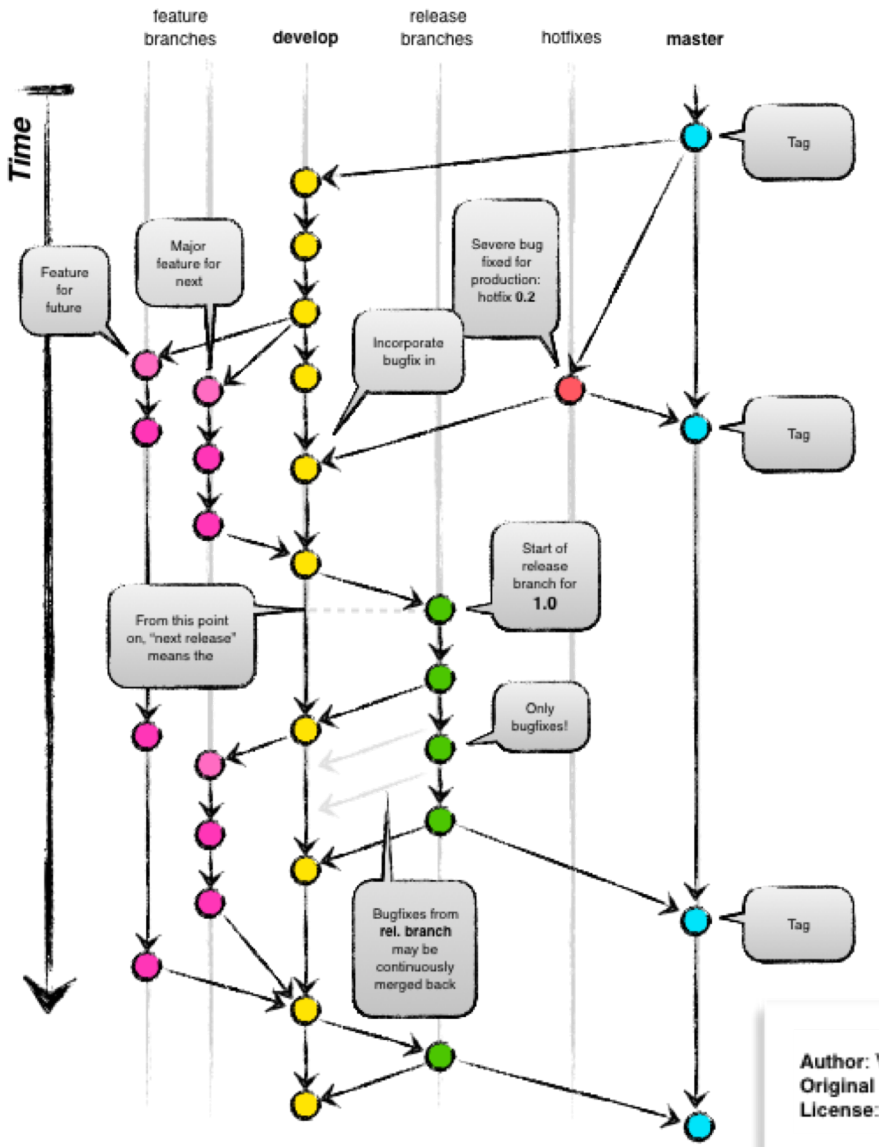
Merge Conflicts

How are merge conflicts resolved in FLASH5 Workflow?

- Merge conflict with master means merge conflict with staged and development
- We want to avoid conflict resolution when merging into master
- Directly on feature branch if resolution is there
- One idea is to merge master into feature branch



Git Flow



- Full-featured workflow
- Increased complexity
- Designed for SW with official releases
- Feature branches based off of develop
- Git extensions to enforce policy
- How are develop and master synchronized?
- Where do merge conflicts occur and how are they resolved?

More Workflows

- GitHub Flow (Scott Chacon)
  - No structured release schedule
  - Continuous deployment & continuous integration allows for simpler workflow
- GitLab Flow

Conclusions

Version control is an amazing tool

- Parallel and distributed working requires coordination and rules to be productive and produce correct code
- Appropriately chosen workflows can ensure quality results and help debugging/verification while helping productivity

Adopt what is good for your team

- Consider team culture and project challenges
- Assess what is and isn't feasible/acceptable
- Start with simplest and add complexity where and when necessary

What do we want from a workflow?

Develop a clear set of polices that

- results in correct code on a particular branch (usually master),
- ensures that a team can develop in parallel and communicate well,
- minimizes difficulties associated with parallel and distributed work, and
- minimizes overhead associated with learning, following, and enforcing policies.