Git for Teams

Dieu Pham
@dieuph

Goals

- ~~Git Commands~~
- Git Workflow

Agenda

- The Good, Bad & Ugly
- Access Control
- Branching Pattern
- Repository Architecture
- Maintenance Strategy
- Challenge

Part 1: The Good, Bad & Ugly

The Good

Git truths you need to internalize:

- Git is a very good content tracker for text files.
- Git is very fast compared to centralized VCS.

The Bad

Git core will not solve all of your problems:

- Git is not a dependency manager.
- Git takes whole file snapshots.
- Git is not optimized for tracking binary files.
- Git does not include in-repository access control.
- Git becomes slower as your history gets very, very large.

The Ugly

- Git is notorious for its "holy wars".
- This makes it seem very complicated and hard to learn.
- This presentation unpacks the rationale behind the most common arguments.

Part 2: Access Control

When you first create a project, you will need to decide who can commit their code to the repository.

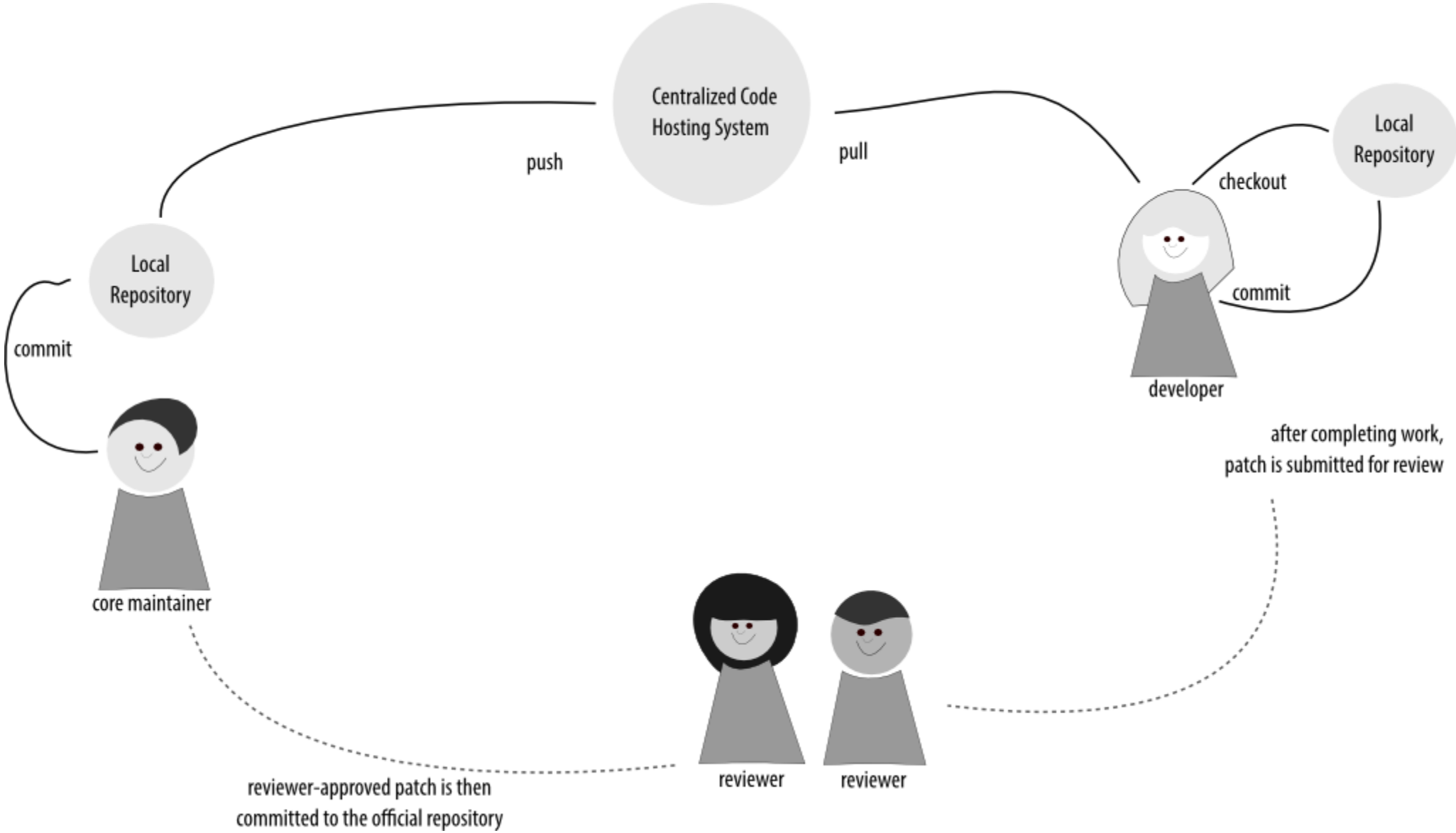
- Who gets write access?
- What can they commit to?

Strategies

- Dispersed Contributor
- Collocated Contributor
- Shared Maintenance

Strategy #1: Dispersed Contributor

- Trust No One; Propose a Solution
- Everyone has read access. Very few have written access. Suggested changes are presented as whole ideas in a single patch file for review.


Centralized Code Hosting System
push
pull
Local Repository
commit
core maintainer
Local Repository
checkout
commit
developer
after completing work, patch is submitted for review
reviewer
reviewer
reviewer-approved patch is then committed to the official repository

Pro & Con

Pro

- Forces a review process.
- Works well with git tools (bisect, gitk).

Con

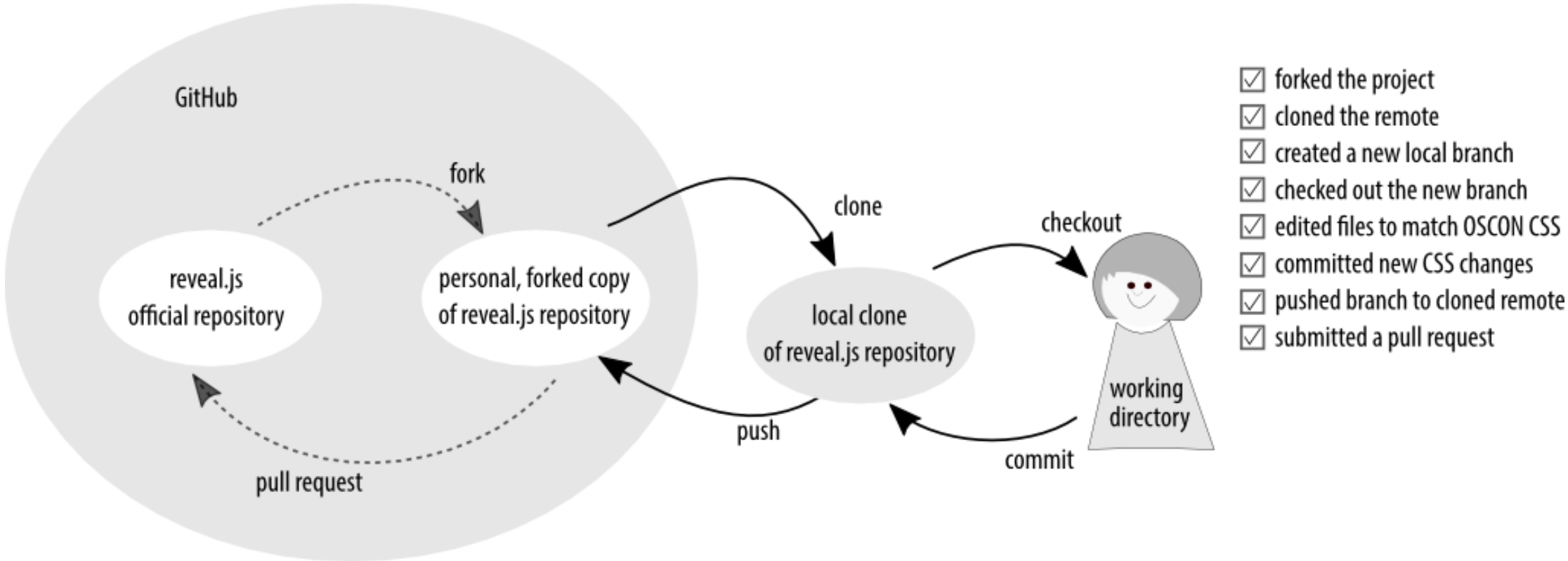
- Sharing work is more complicated than branching.
- Contributors (potentially) need to setup their own code hosting platform.

Examples

- Linux
- FOSS projects still using a centralized code hosting model OR mailing-list code sharing model

Strategy #2: Collocated Contributor

- Trust No One; Show Your Work
- Project forks give full permissions to developers so they can do work in any commit granularity they choose. New work is added to the main project through a request to the upstream project via a proposed branch of commits.



- ☑ forked the project
- ☑ cloned the remote
- ☑ created a new local branch
- ☑ checked out the new branch
- ☑ edited files to match OSCON CSS
- ☑ committed new CSS changes
- ☑ pushed branch to cloned remote
- ☑ submitted a pull request

Pro & Con

Pro

- Forces a review process.

Con

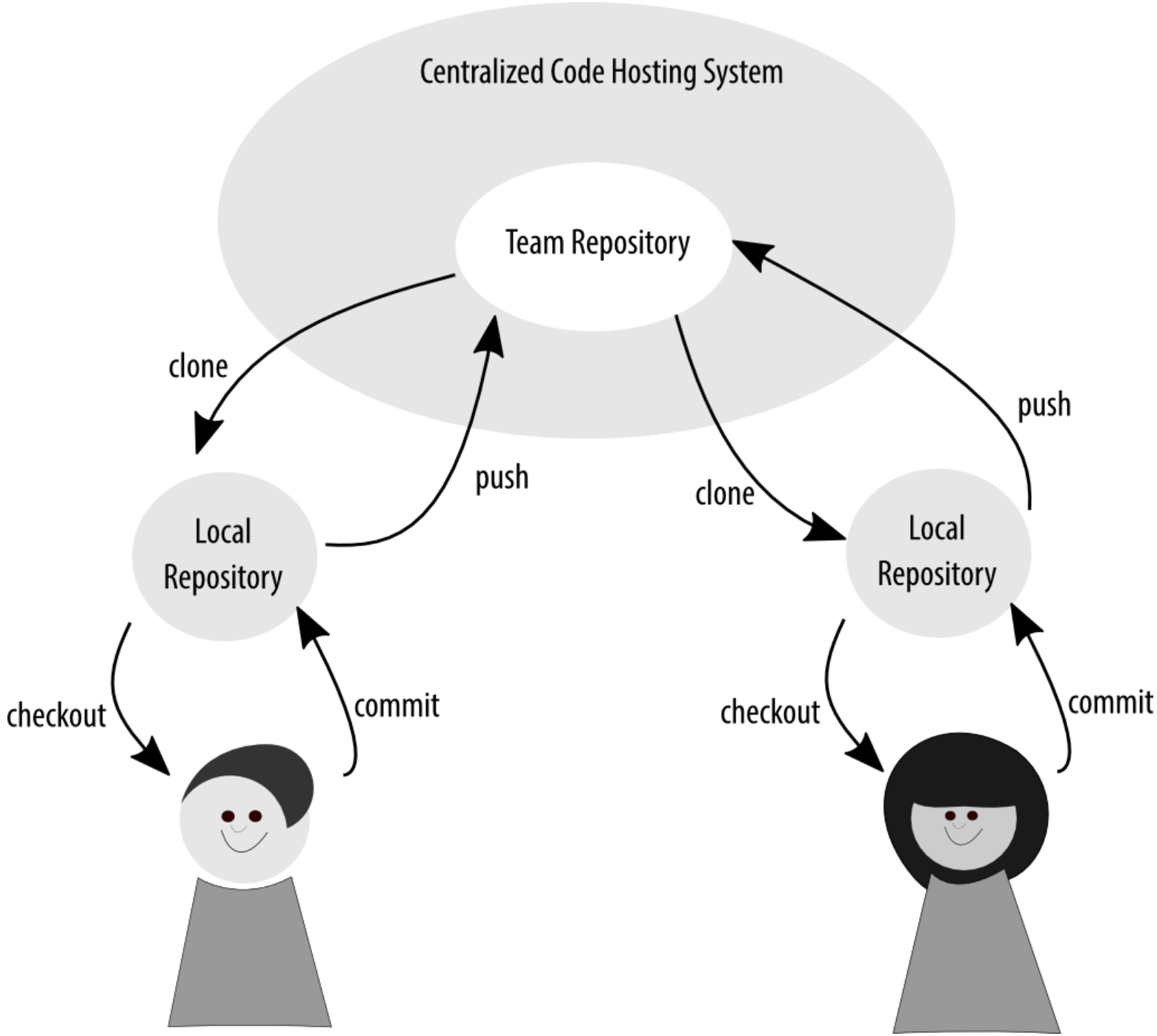
- Commit granularity may prevent effective debugging.
- Private repos must be duplicated per team member.
- More steps to incorporate new work.

Examples

- Django
- Ruby on Rails
- CakePHP
- FOSS projects hosted on GitHub

Strategy #3: Shared Maintenance

- Trust the Process
- Developers work in a branch of the centralized code repository. Only the politics of the project prevent them from committing their work to the main body of work.


Centralized Code Hosting System
Team Repository
clone
push
clone
push
Local Repository
Local Repository
checkout
commit
checkout
commit

Pro & Con

Pro

- Encourages clean/working master.

Con

- Encourages, but does not require code review.
- Must give explicit write permission to all team members.

Examples

- Internal projects with trusted developers

Summary

- Dispersed Contributors - Trust No One; Propose a Solution
- Collocated Contributors - Trust No One; Show Your Work
- Shared Maintenance - Trust the Process

So What?

- If you choose **shared maintenance**, you need to setup a **PRIVATE** repository for your code, and to push their changes to the grant permission to all team member.
- If you choose **collocated repositories**, you need to setup **PUBLIC** or **PRIVATE** repository for your code, and ensure all team members can create their own PUBLIC or PRIVATE copy of the project, AND submit merge requests to the main project.

Best Practices

- Choose and use a strategy that suits your governance model.
- Basic setup has teammates share a centralized repository.
- Pull requests are commonly used as an access gate (for specific branches).

Part 3: Branching Pattern

Identify and describe how your code is collated within your repository.

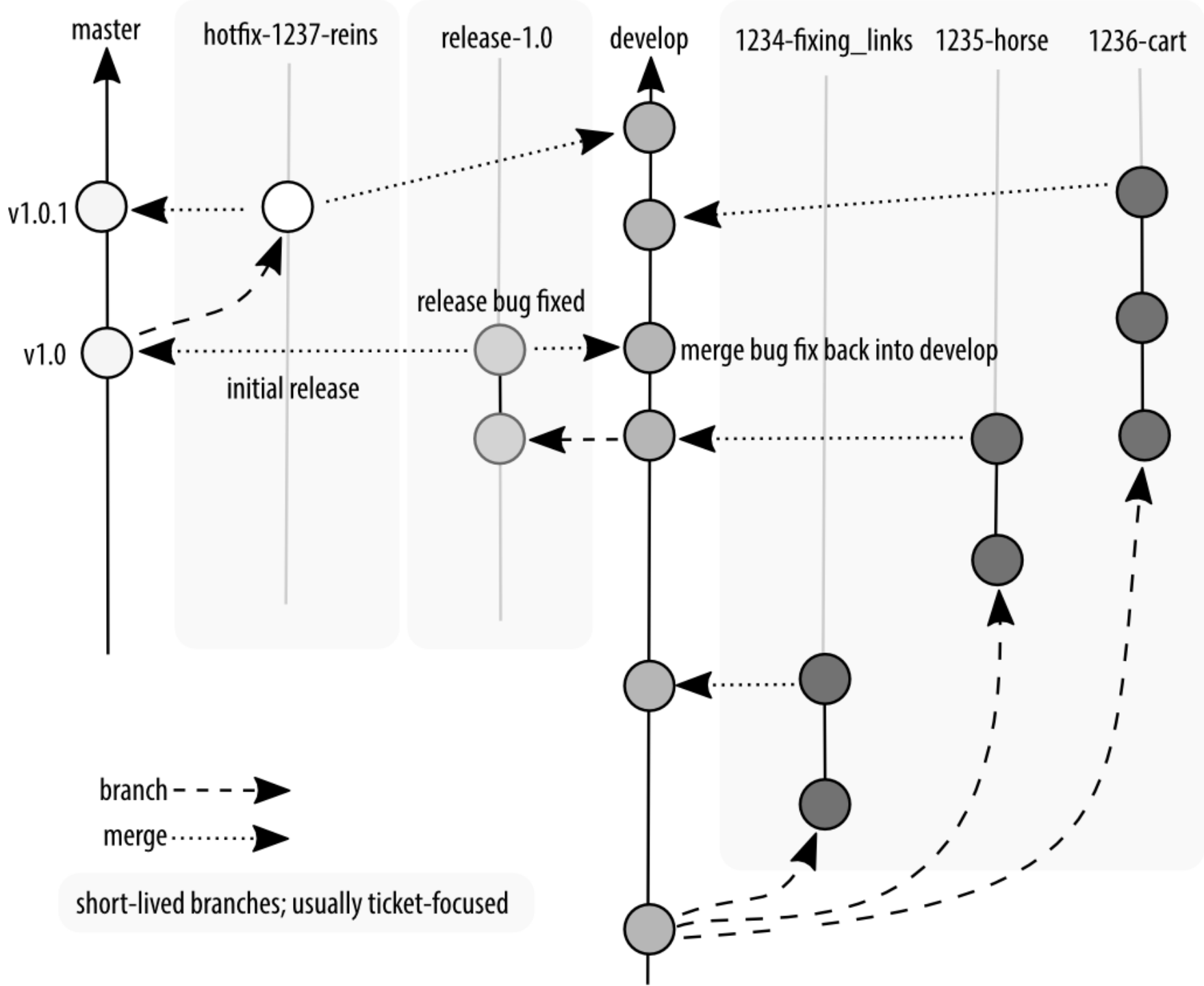
- How do you separate your work in progress from fully tested, approved work?

Popular Convention

- Scheduled Release
- Branch-per-Feature
- State/Environment Branching

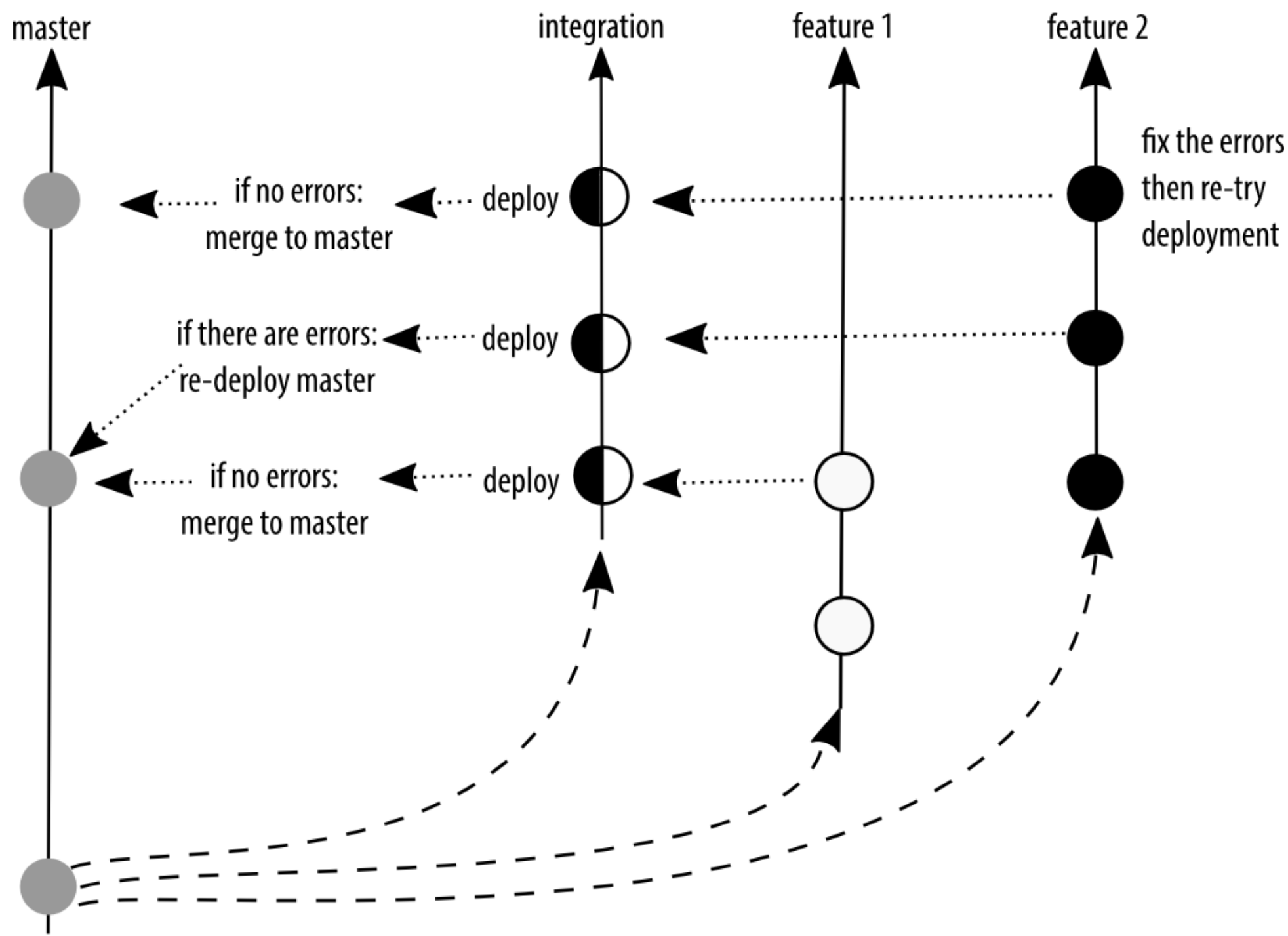
Scheduled Release

- Optimized for the collation of many smaller changes into a single release.
- Typically used for a download-able product; or web site with a scheduled release cycle (e.g. "Wednesdays").
- Incorporates human-reviews, and possibly automated tests.

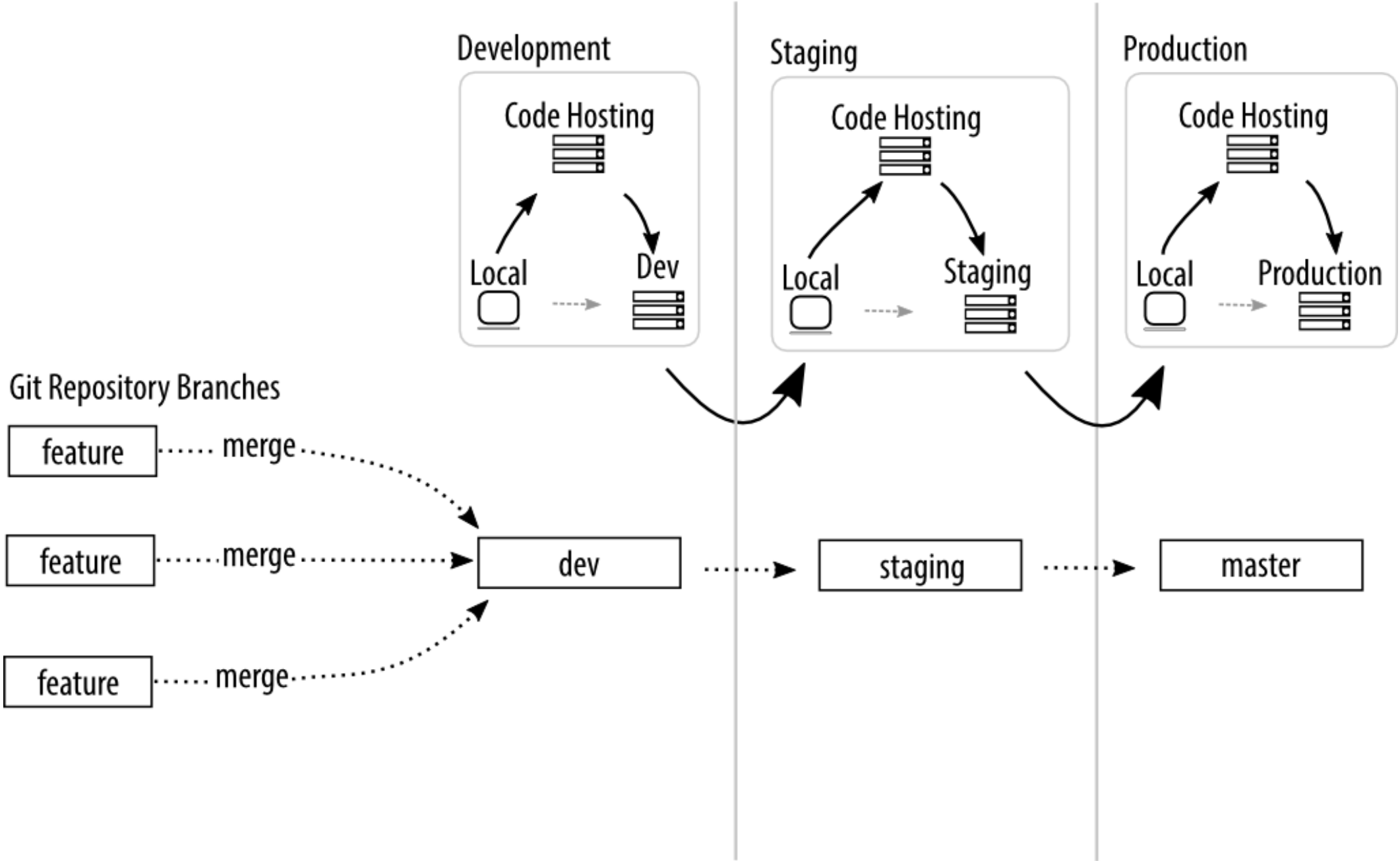

master
hotfix-1237-reins
release-1.0
develop
1234-fixing_links
1235-horse
1236-cart
v1.0.1
v1.0
release bug fixed
merge bug fix back into develop
initial release
branch
merge
short-lived branches; usually ticket-focused

Branch-per-Feature

- Code is deployed faster than scheduled releases; assumes all check-ins are deployable.
- Requires (trusted) test coverage.
- Typically uses a mechanical gatekeeper (CI) to check in code to the master branch.
- Often has flippers/flags for fine grained access to in-progress features.
- Fewer branches to maintain / keep updated.


master
integration
feature 1
feature 2
fix the errors
then re-try
deployment
if no errors:
merge to master
deploy
if there are errors:
re-deploy master
deploy
if no errors:
merge to master
deploy

State/Environment Branching


Development
Code Hosting
Local
Dev
Staging
Code Hosting
Local
Staging
Production
Code Hosting
Local
Production
Git Repository Branches
feature
merge
feature
merge
feature
merge
dev
staging
master

So What?

- If you choose **SCHEDULED DEPLOYMENT**, streamline how your code is collated for release.
- If you choose **BRANCH-PER-FEATURE**, codify how trust is deployed in your code.
- If you choose **STATE BRANCHING**, establish your infrastructure and automate where possible.

Best Practices

- Pick a branching strategy which suits your deployment schedule.
- It doesn't really matter which strategy you use.
- Document the exact steps people should use. Ensure documentation is followed.

Part 4: Repository Architecture

- How do you manage dependencies?
- Where do you store very large files?
- How do optimize your build process for very fast deployments?

Dependency Management

- "Vendor branches" -- named branches for upstream work
- Version your build manifest
- Subtrees -- nest repositories without tracking
- Submodules -- nested repositories with hierarchical tracking

Microservices or Monolith

Monolith:

- Consider your audience: if you don't need to scale, and it's easier for your team, use a single repository to store all knowledge for a project.
- If you don't know exactly what you're building, stick to one repo for your code.
- If the language you're working in doesn't have a package manager, consider using one repo for deployments.

Microservices:

- Think OOP: For separate functionality, use separate repositories.
- Pull together related pieces at build time.

Very Large Files

- Compiled dependencies: i.e., libraries your program needs to run.
- Asset binaries: e.g., source files for images, video files.

Use Offsite Storage for Very Large Files

Do not version binaries in the repository; reference them from another location.

- git-annex
- git-bigfiles
- GLFS

Use Shallow Clones for Faster Deployments

Avoid grabbing all versions of a file for the deployment.

$ git clone --depth [depth] [remote-url]

$ git clone [URL] --branch [branch_name] --single-branch [folder]

Best Practices

- Store very large files outside of the repository.
- Speed up deployments with shallow clones.
- Start with a single repository while you finalize your architecture.
- Refactor into stand-alone repositories when you can identify discrete operations.

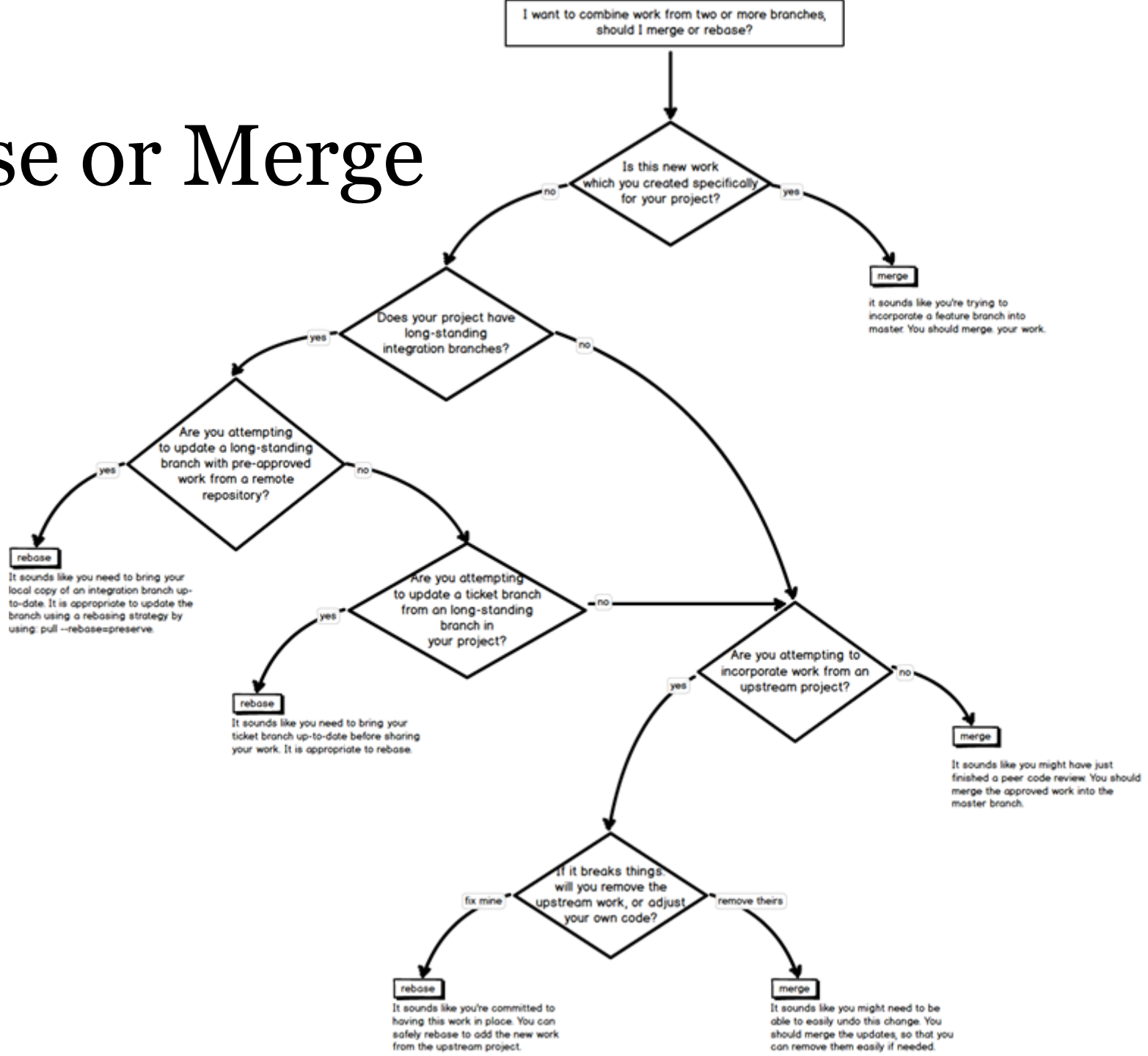
Part 5: Maintenance Strategies

- How do you incorporate upstream work? aka How do you bring branches up to date?
- How do you combine newly approved work into your project's stable branch?

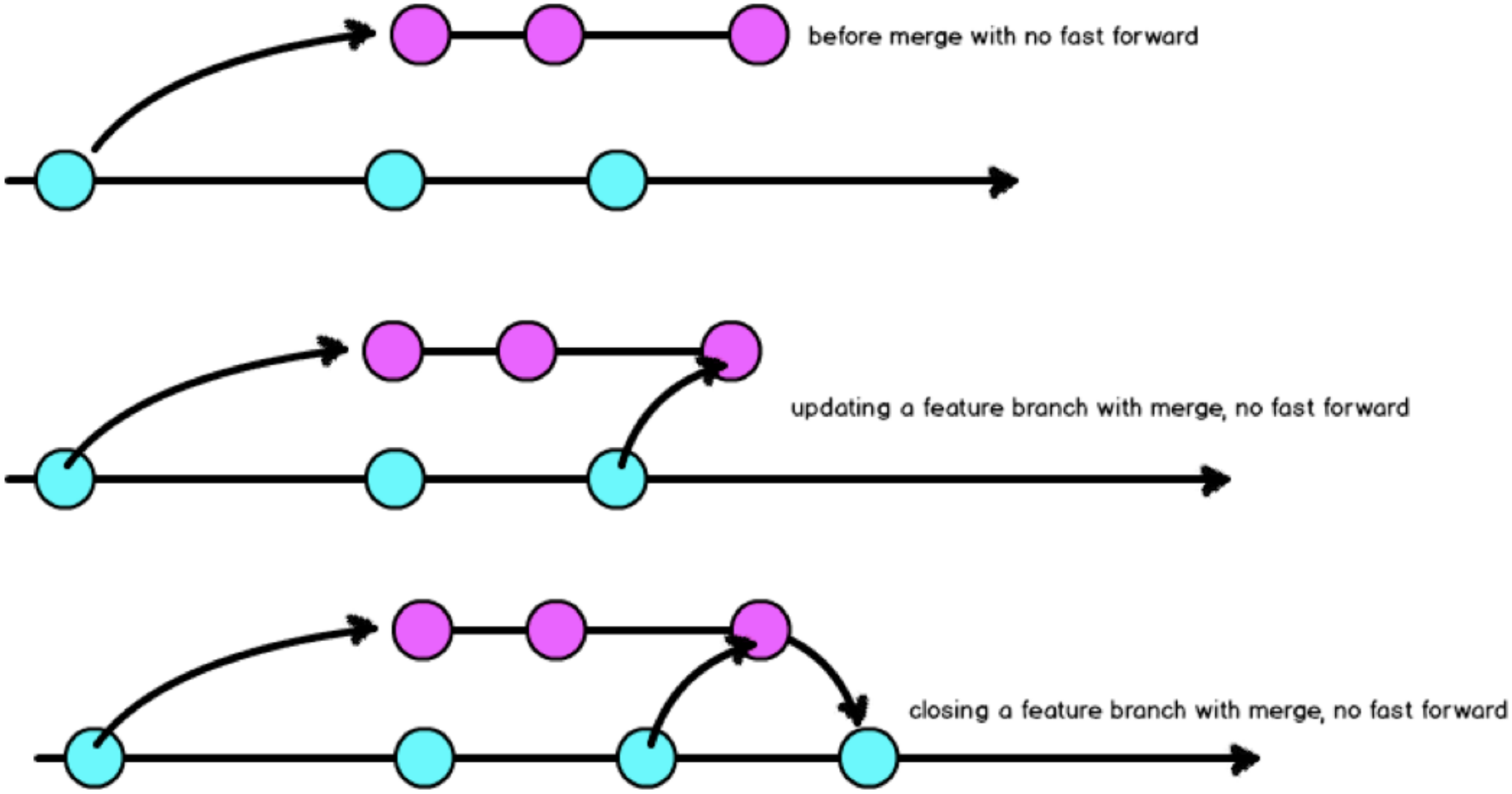
Why the Fuss?
Because TIMTOWTDI

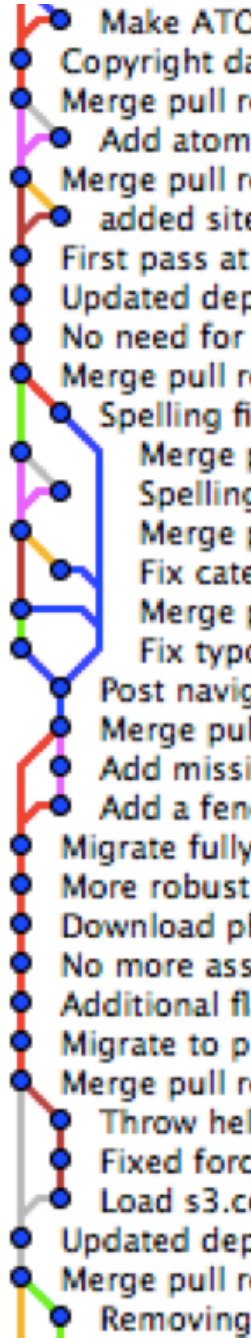
- pull => fetch + merge
- pull --rebase=preserve => fetch + rebase
- merge --no-ff => forces a merge commit object (“true merge”)
- merge --ff-only => fast forward (graph looks like rebase)
- merge --squash => compress commits to one; then merge
- rebase => forward-port local commits
- cherry-pick => merge individual commits

Rebase or Merge

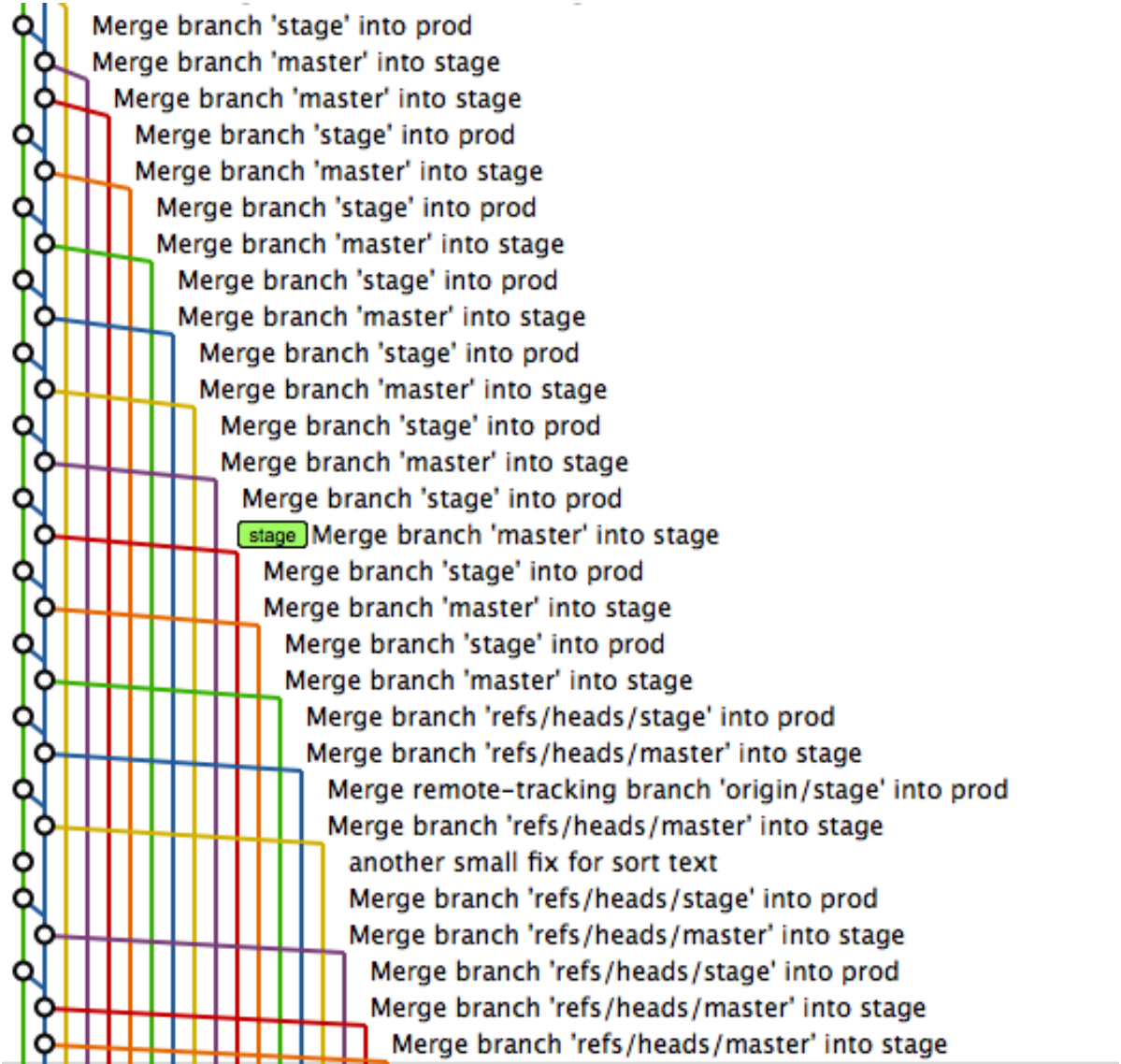


Merging to Update is "Messy"





Merge Commits to Combine are "Messy"



Commit to Whole Thoughts

Reshape commit objects so that they are:

- just the right size
- contain only related code
- conforming to coding standards

$ git rebase --interactive HEAD~n

Convert Conversations to Conclusions

If the review process has resulted in additional commits, squash these commits into logical conclusions.

- Prior to merge:

$ git rebase --interactive HEAD~n

- At merge:

$ git merge --squash NNNN-pull_request_branch

Best Practices

- When submitting work for review: Commit to whole thoughts.
- When merging work: Convert conversations to conclusions.
- By default pull will merge. This is hard to read in a graph.
- (if it matters): use pull --rebase=preserve to update your branches.

Part 6: Challenge

Set up a Git project for team

- Development team: 3+ developers, 1 scrum master
- Operation team: 2 operators
- Management team: 1 project manager

Delivery Plan

- 1 Feature Package/2 weeks
- 1 Patch Package/4 weeks
- 1 Production Package/3 months

How to

- Describe the work your team does.
- Map your network of repositories.
- Diagram your branching strategy.
- Defend what gets stored in the repo.
- Simulate delivery plan.
- Submit your team result: link of repositories, diagrams, team members.

Required Using

- cherry-pick
- rebase
- merge
- tag
- branch
- pull request
- code review

Thanks!

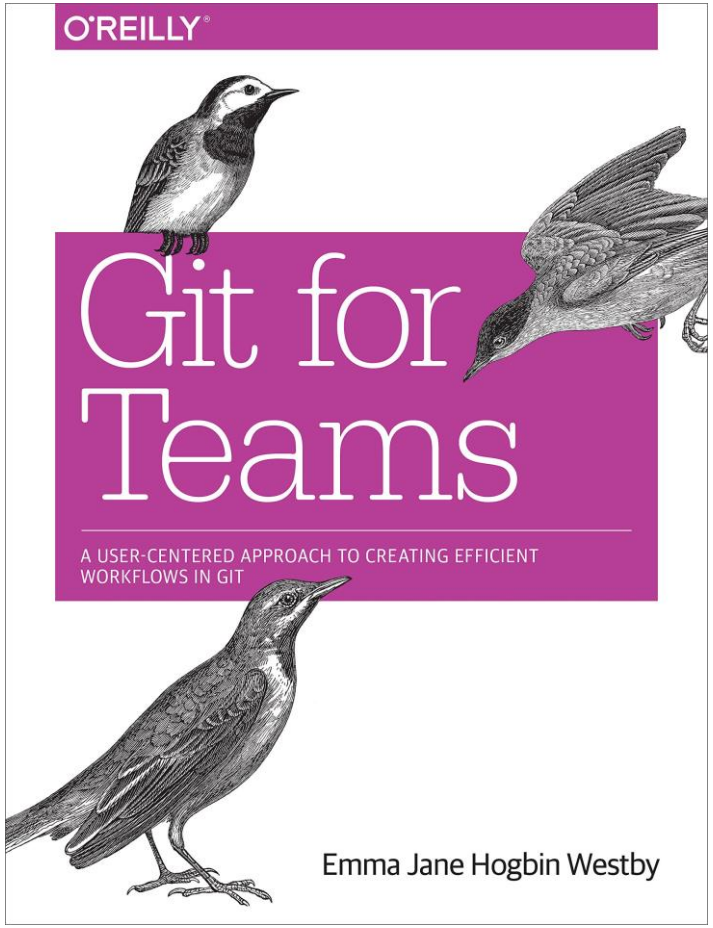
Stay connected!

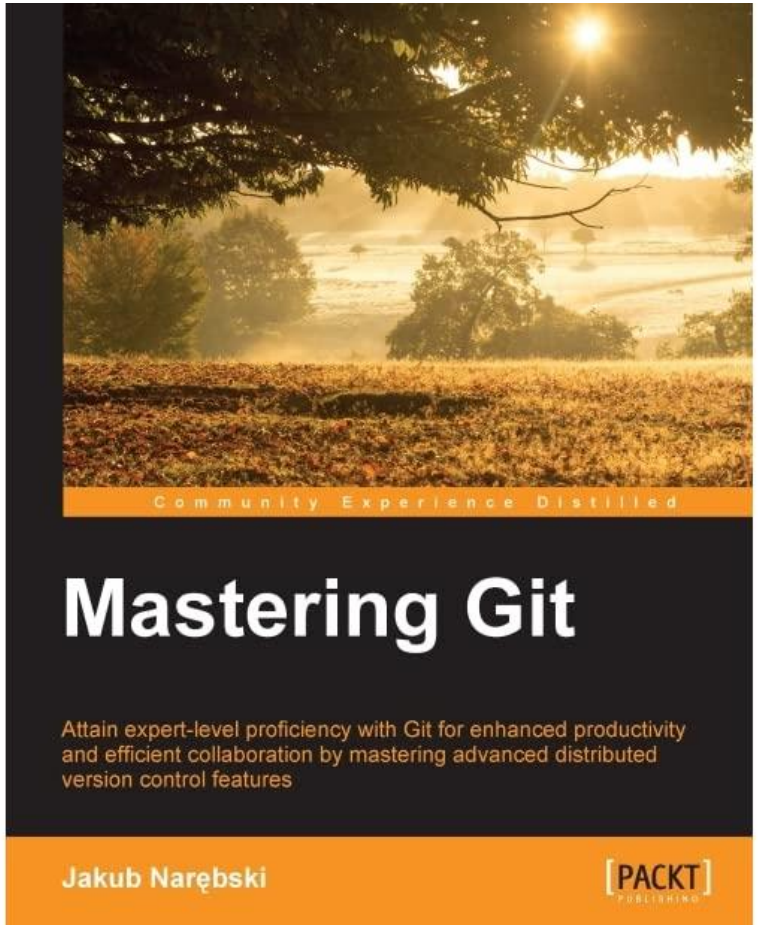
Skype: dieuph2

Resources

Books, References and so on.

Books


O'REILLY
Git for Teams
A USER-CENTERED APPROACH TO CREATING EFFICIENT WORKFLOWS IN GIT
Emma Jane Hogbin Westby


Community Experience Distilled
Mastering Git
Attain expert-level proficiency with Git for enhanced productivity and efficient collaboration by mastering advanced distributed version control features
Jakub Narębski
[PACKT]
PUBLISHING

References

- Introduction to Git
- Git Documentation
- Pro Git
- ungit

- How to Handle Big Repositories with Git
- HackerNews: How do you handle your microservices
- StackExchange: How do you handle external dependencies?
- Organizing Microservices in a Single Repository
- Mastering Submodules

Retrospective

- What did you learn?
- What is (still) confusing?
- What do you think would have made this workshop better?
- What is your interesting topic for the next workshop?