

# Design Patterns for Git Workflows

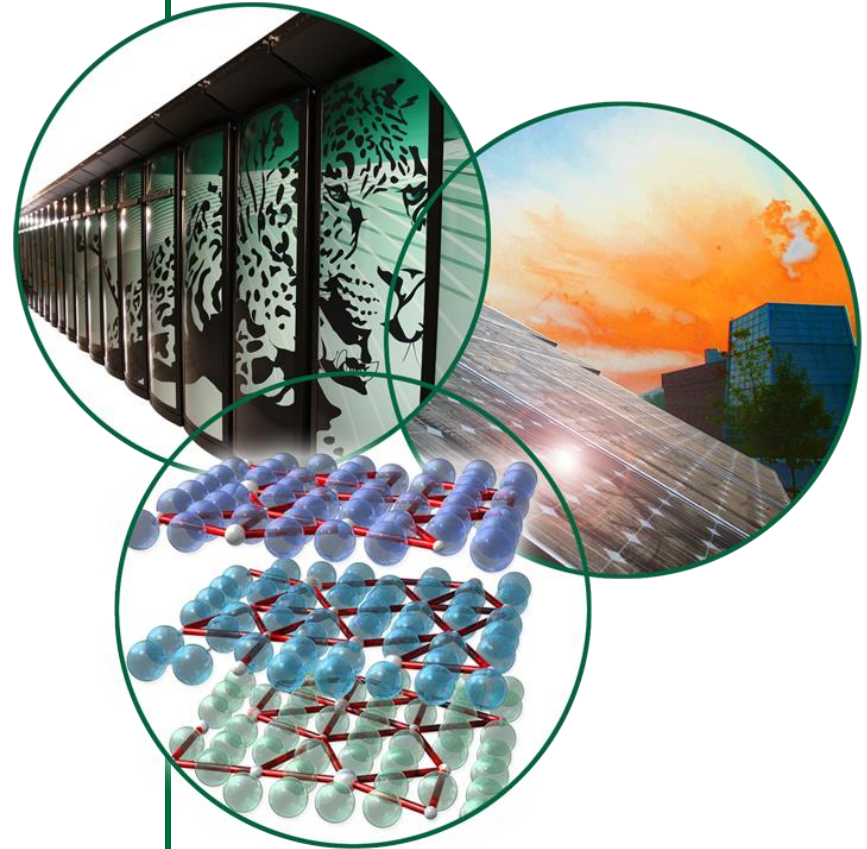
Trilinos Spring Developers Meeting

May 13, 2015

Roscoe A. Bartlett

Oak Ridge National Lab

- Computational Eng. & Energy Sciences
- Computer Science and Mathematics Div



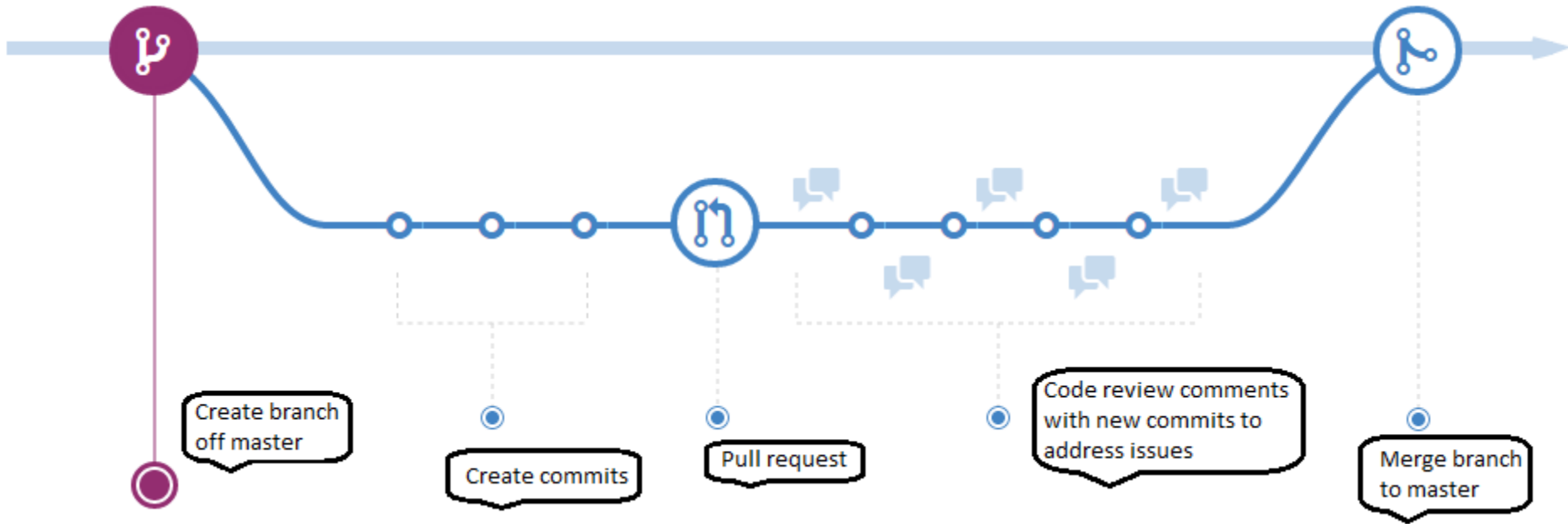
# Background on Git Workflows

- 2005: Git development begins (Linus Torvalds, for Linux Kernel)
- 2010: “**Gitflow**” is presented (Vincent Driessen)
  - Uses ‘develop’ and ‘master’ branches with short-lived “feature”, “release” and “hotfix” branches
  - Most well known and popular git workflow (by far)
  - Every workflow since compares itself to gitflow (and criticizes gitflow in the process)
- After Gitflow:
  - “**Github Flow**”: Advocated by Github
    - Simple feature branches with pull requests
  - “**Simple Git Workflow is Simple**”: Advocated by Atlassian
    - Simple feature branches with rebasing on top of ‘master’ and --no-ff merges to ‘master’
  - “**Gitlab Flow**”: Advocated by Gitlab team
  - “**Git.git Worklow**” (i.e. “gitworkflows(7)”: Official git man page “gitworkflows(7)”
    - All feature branches with ‘next’ and ‘pu’ temp testing branches, graduate to ‘master’.
    - Use by the developers for git itself (i.e. git.git)
    - Used by Linux Kernel developers

See [“Overview and Analysis of Version Control and Development Strategies with Git used by CSE Projects”](#)

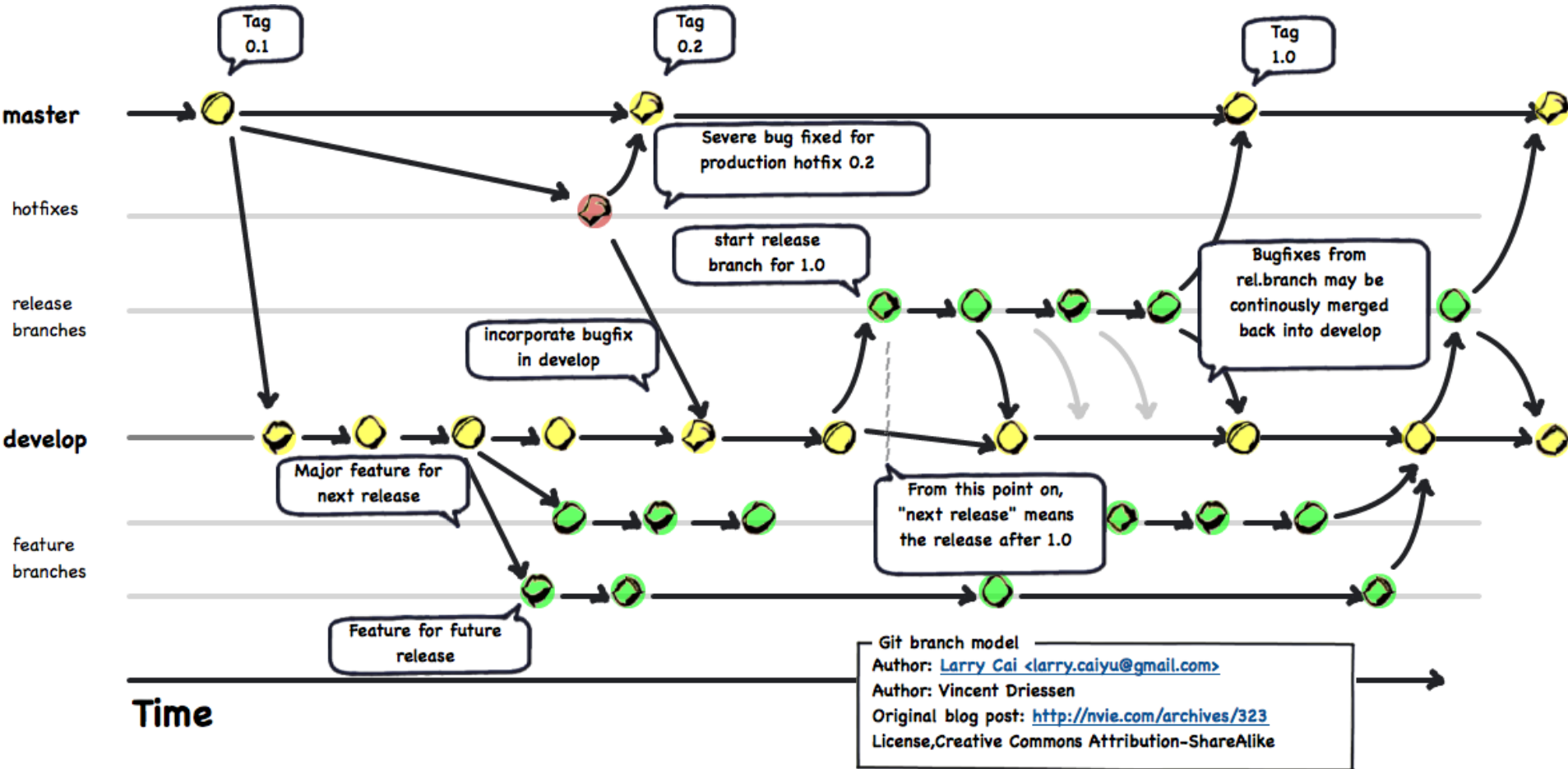
# Overview of Existing Defined Workflows

# Github Flow



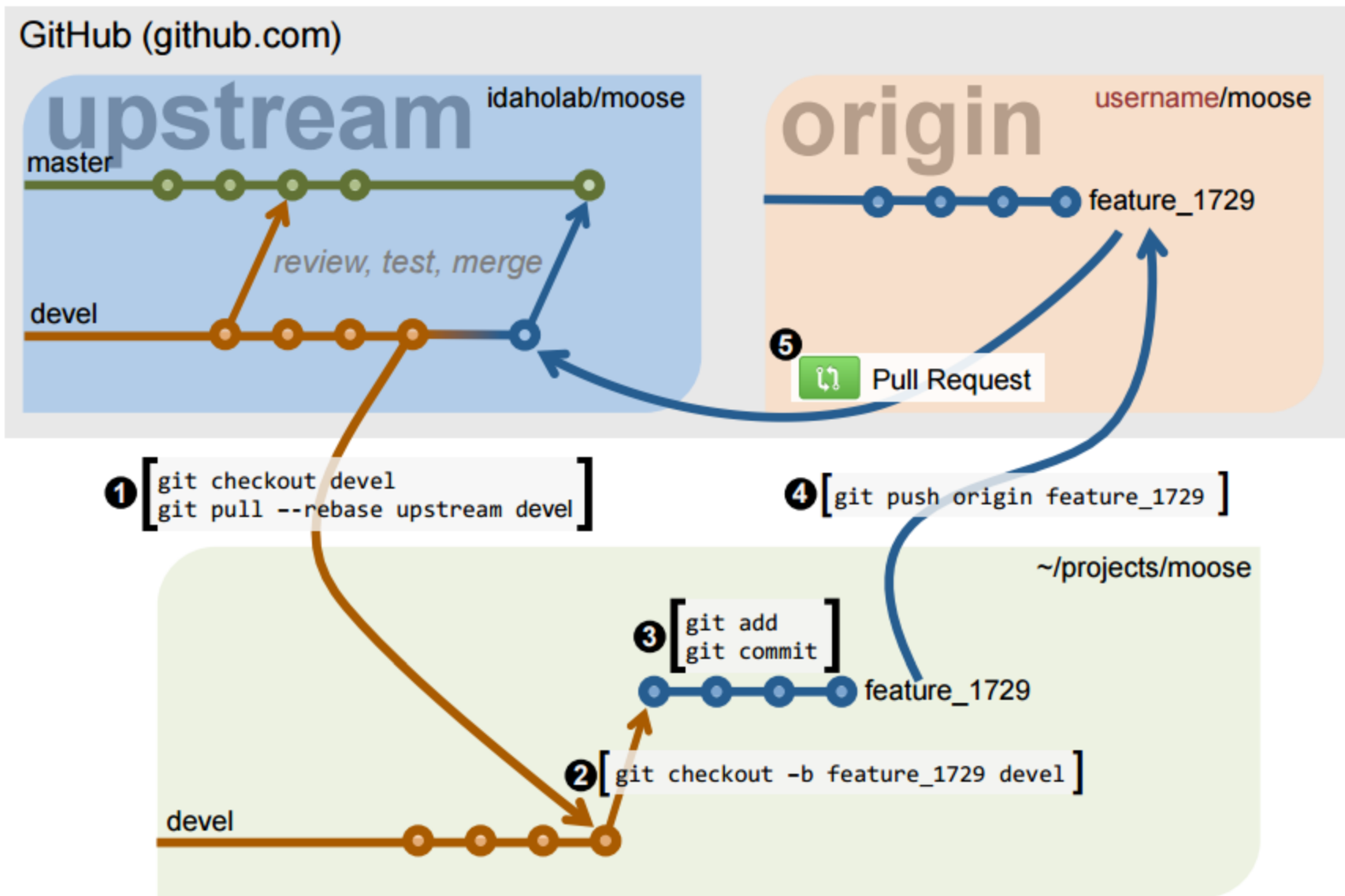
- **Permanent branches:** 'master'
- **Primary focus of testing:** Each individual feature branch
- **Notes:**
  - No release branches! (can't support multiple releases)

# Gitflow

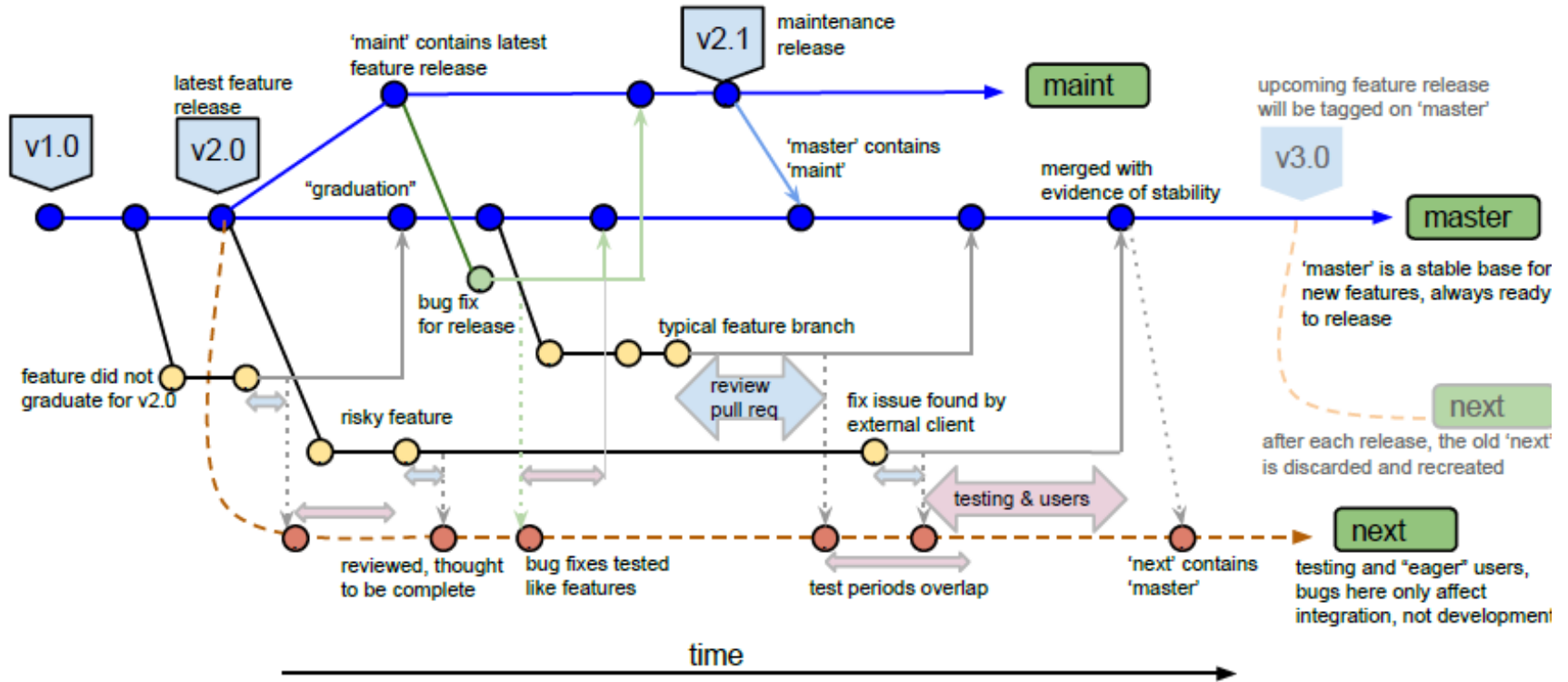


- **Permanent branches:** 'develop' and 'master'
- **Short-lived branches:** "feature", "release", and "hotfix"
- **Primary focus of testing:** 'develop' branch (but still need testing on 'master' if 'hotfixes' exist)
- **Special git command systems** has been created to drive workflow! (Criticized as too complex)

# MOOSE Workflow (Gitflow + Github Flow)



# Git.git Workflow “gitworkflows(7)” (PETSc)



- > first-parent history of branch
- > merge history (not first-parent)
- > merges to be discarded when 'next' is rewound at next release
- merge in first-parent history of 'master' or 'maint' (approximate "changelog")
- merge to branch 'next' (discarded after next major release)
- commit in feature branch (feature branches usually start from 'master')
- commit in bug-fix branch (bug-fix branches usually start from 'maint' or earlier)

See:  
[Gitworkflows\(7\) man page](#)  
[Gitworkflows\(7\) presentation](#)

# Design Patterns for Git Workflows



# Incrementally Expanding Git Workflow (Intro)

- Instead of defining complete workflows to choose from
  - ⇒ **Define git workflow “building blocks” and construct the workflow that is need!**
- Workflow Construction Steps:
  - Consider the properties and challenges for a given project
  - Construct simplest git workflow using building blocks to meet current needs
  - Add new features to workflow as situation changes and more challenges emerge
- Workflow building blocks
  - Begin: The simple centralized CI workflow
  - Addition of a ‘develop’ branch
  - Addition of topic branches
  - Addition of a subteam branch
  - Addition of release branches
  - Addition of feature branches
  - Addition of throw-away integration test branch(es)
  - End: The Git.git Workflow (e.g “gitworkflows(7)”)

These can be added to a git workflow in almost any order!

- **Consistent with “gitworkflows(7)”**

<https://www.kernel.org/pub/software/scm/git/docs/gitworkflows.html>

# Issues to Consider to Select Git Workflow

- Number of developers
- Distribution of general software knowledge and skills of the developers
- Distribution of git-specific knowledge and skills of the developers
- Amount of (or lack of) communication and coordination between the developers
- Nature of the customers and need for releases of the software
- Sensitivity of the software (e.g. security vulnerabilities?)
- Rate of development and change in the software
- Importance (and urgency) of performing code reviews
- Portability requirements and portability challenges of the software
- Heterogeneity of the development and testing environments

# Testing Issues/Support for Git Workflows

## Test Suites:

- **CI Build:** This is a **Continuous Integration** (CI) [2, 3] build of the code on a single platform and single configuration and the running of a (relatively) fast test suite. The CI Build should be constructed so that it protects the major features of the code that are needed by the other developers to continue their development work. The CI Build would be performed before any branch is updated that impacts other developers. This is the **pre-push regression test suite** described in [1].
- **Nightly Builds:** This is a collection of builds on different platforms with different compilers and running a more comprehensive (i.e. expensive) test suite. This is the **nightly regression test suite** described in [1].

## Testing assumptions:

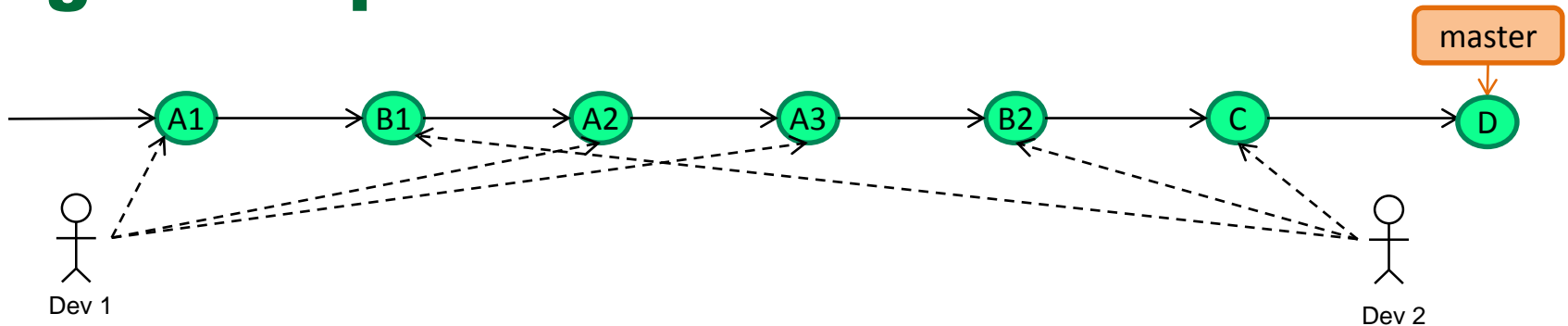
- **Additive test assumption of branches:** If 'm + a' PASSES and 'm + b' PASSES, then 'm + a + b' also PASSES
- **Subtractive test assumption of branches:** If 'm + a + b' PASSES then 'm + a' or 'm + b' also PASSES.

[1] [“How to Add and Improve Testing in Your CSE Software Project”](#), IDEAS Project, 2015.

[2] [“Continuous Integration”](#), Wikipedia, [http://en.wikipedia.org/wiki/Continuous\\_integration](http://en.wikipedia.org/wiki/Continuous_integration)

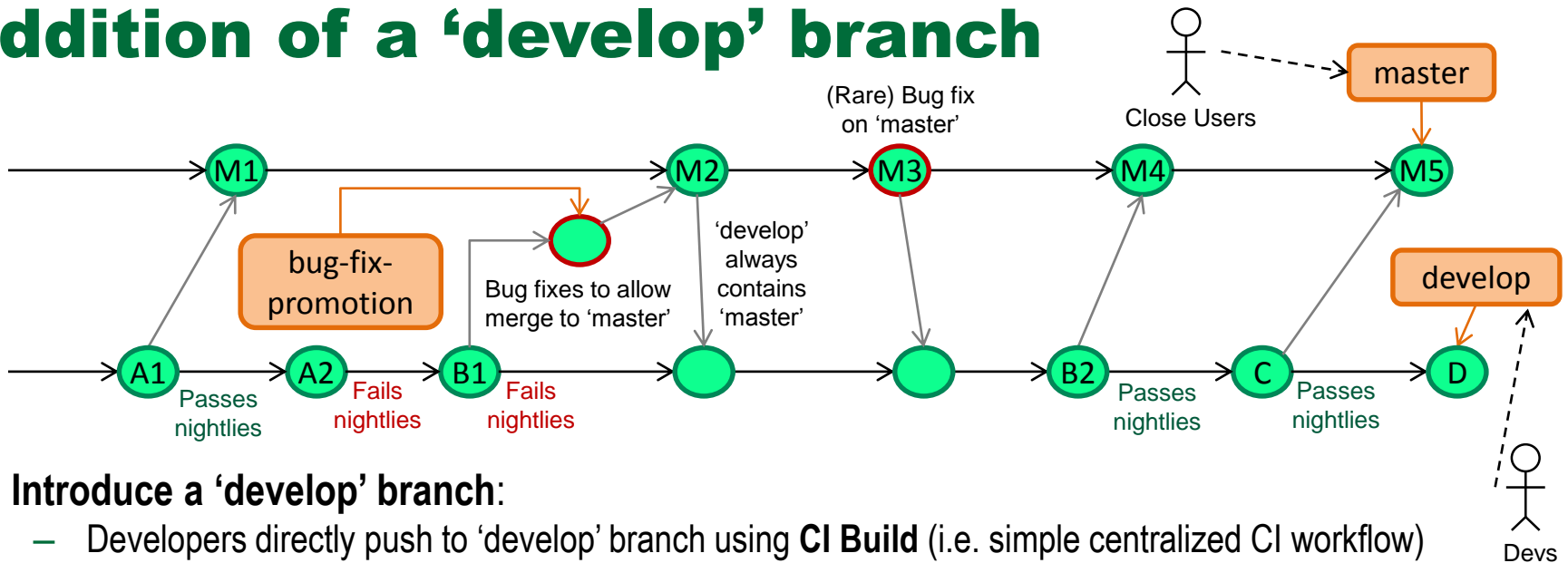
[3] Fowler, Martin. [“Continuous Integration”](http://martinfowler.com/articles/continuousIntegration.html), <http://martinfowler.com/articles/continuousIntegration.html>

# Begin: Simple Centralized CI Workflow



- Features implemented in commits intermingled on 'master' branch
  - Feature "A": Commits "A1", "A2", "A3"
  - Feature "B": Commits "B1", "B2"
  - Feature "C": Commit "C"
- **Pros and Cons** (w.r.t. other more sophisticated workflows):
  - **Pro:** Simplest workflow with fewest git commands, no distributed VC concepts (i.e. SVN-like)
  - **Pro:** Requires least knowledge of git
  - **Pro:** Minimizes merge conflicts (frequent pushes to and pulls from 'master')
  - **Con:** Difficult to perform pre-merge code reviews
  - **Con:** Difficult to collaborate with other developers with partial changes (can't push broken code to 'master' to share with others)
  - **Con:** Difficult to back out bad feature sets
  - **Con:** Difficult to maintain 100% passing tests for all Nightly Builds
- **Example project:** New research project
  - Small number of closely collaborating developers
  - No real users (e.g. no need to support releases)

# Addition of a 'develop' branch



- **Introduce a 'develop' branch:**

- Developers directly push to 'develop' branch using **CI Build** (i.e. simple centralized CI workflow)
- Only merge from 'develop' into 'master' when all **Nightly Builds** pass (perhaps with minor bug fixes)
- Temp 'bug-fix-promotion' branch can be used to stabilize and fix bugs before update of 'master'
- Close users pull from more stable 'master' branch ('master' is default branch when cloning a git repo!)
- Most testing focused on 'develop' branch. (Little to no testing needed on 'master')

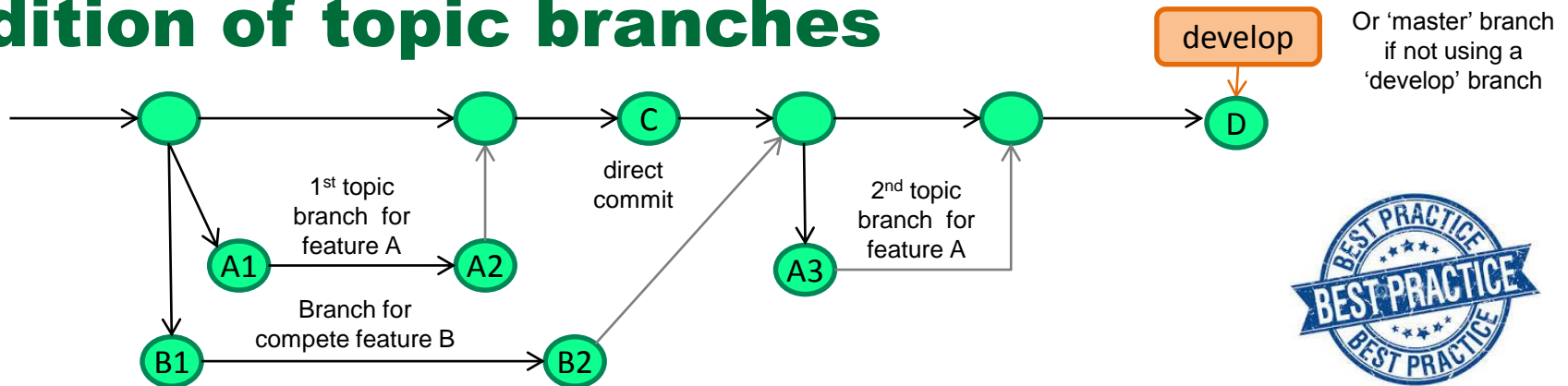
- **Pros and Cons** (w.r.t. single branch workflow):

- **Pro:** Developers still only perform simple centralized CI workflow (only on 'develop' not 'master')
- **Pro:** More stable 'master' branch seen by users
- **Pro:** Allows some time for review of commits on 'develop' before merge to 'master'
- **Con:** Requires knowing how to use multiple branches and merges
- **Con:** Extra effort to perform merges from 'develop' to 'master' (or **could use cron job to do merges**)

- **Example project:** Established research project with close users

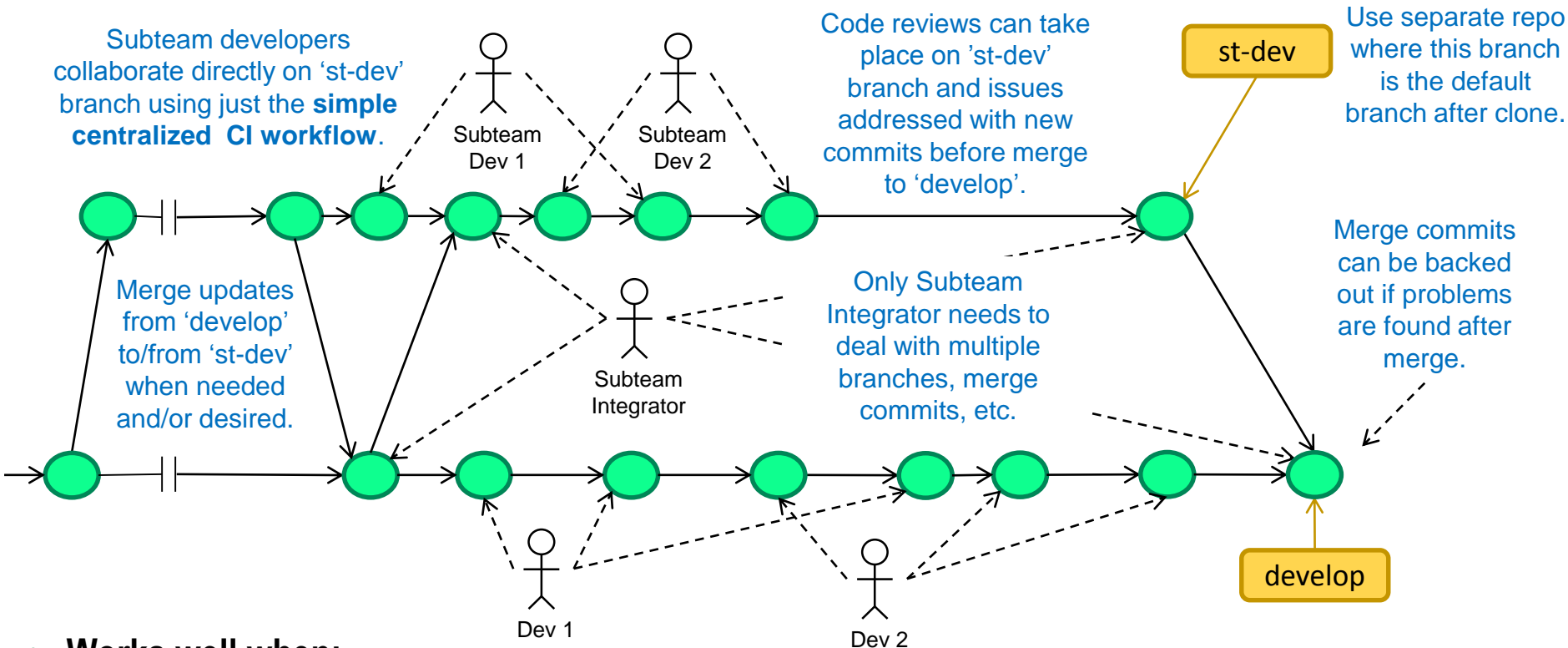
- Small number of closely collaborating developers
- Few close customers that can't handle the instability of the main dev branch

# Addition of topic branches



- **Introduce usage of temporary short-lived topic branches:**
  - Developers (optionally) implement features in one or more topic branches and merge to 'develop'. E.g.:
    - Feature "A": 1<sup>st</sup> topic branch (commits "A1", "A2"), 2<sup>nd</sup> topic branch (commit "A3")
    - Feature "B": Single topic branch (commits "B1", "B2")
  - Topic branches pass **CI Build** merged into 'develop' about once/day or 4-6 hours of work (rule of thumb)
  - Direct pushes to 'develop' are okay for single commit changes that are not shared/reviewed.
  - **NOTE: Usage of topic branches does not degrade CI at all! Does not lead to more merge conflicts!**
  - **NOTE: Not typically long-lived "feature branches" that are hard to merge back!**
- **Pros and Cons** (w.r.t. single branch workflow):
  - **Pro:** Allow changes to be easily backed out if something goes wrong
  - **Pro:** Allow switching between different topic branches quickly
  - **Pro:** Allow easy sharing for quick collaboration with other devs before merging to 'develop'
  - **Pro:** Allow quick code reviews (pull-requests) on the topic branch before merging to 'develop'.
  - **Con:** Requires knowing how to use multiple branches and merges with git
- **Example project:** Established research project with multiple developers
  - Medium number of number of developers who closely collaborate and review code

# Addition of a subteam branch



- **Works well when:**

- Changes by subteam commits don't typically conflict with commits on main 'develop' branch.
- Criteria for pushing to 'st-dev' branch may be different than for pushing to 'develop' branch.

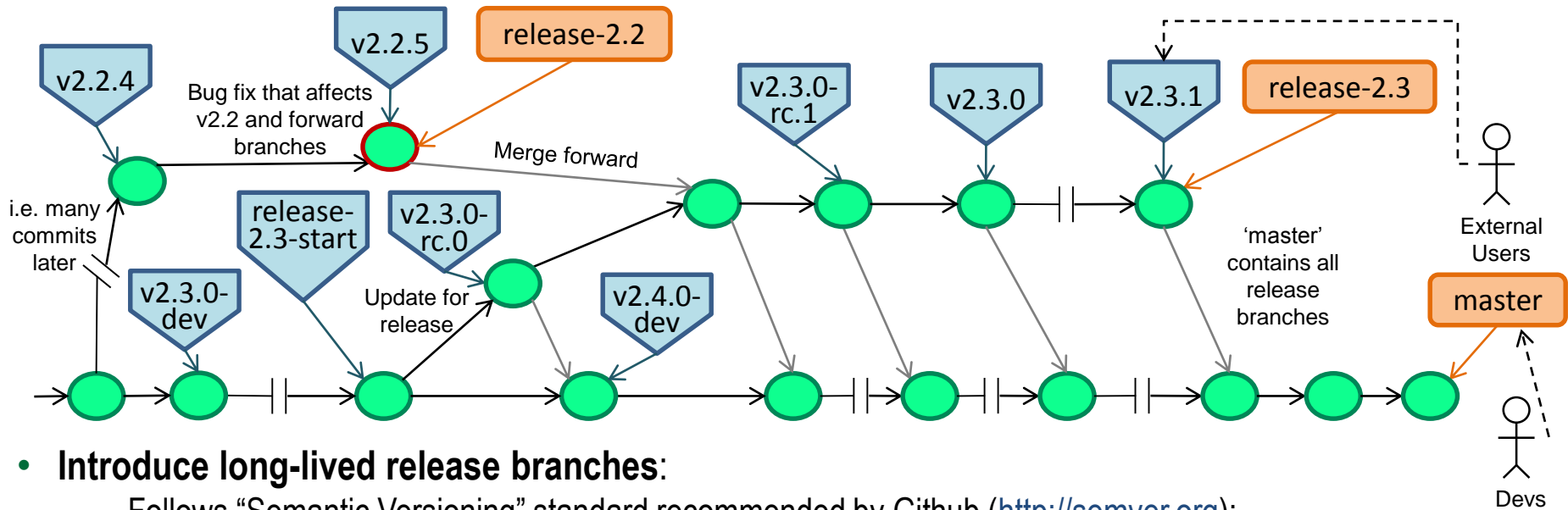
- **Pros and Cons** (w.r.t. topic branch workflow):

- **Pro:** Most subteam developers only need to know and use the **Simple Centralized CI Workflow**
- **Con:** Can create messier git history (e.g. can't 'git rebase -i' to clean up 'st-dev' before merge to 'develop')

- NOTE: Some topic branches can still be used by subteam developers in certain cases

- Used for CASL Trilinos co-development (~ 2013-2015), CASL TriBTS (~2013-now), ROL (currently)

# Addition of release branches



- **Introduce long-lived release branches:**

- Follows “Semantic Versioning” standard recommended by Github (<http://semver.org>):
  - **Release tag:** vX.Y.Z (X = major, Y = minor, Z = patch), **Release candidate:** vX.Y.0-rcZ
- Apply bug fix to oldest release branch that needs the fix then merge forward (“gitworkflows(7)”)
- Cherry-picks from upstream to downstream also allowed (but not preferred)
- NOTE: Since ‘master’ contains all release branches, then just testing ‘master’ provides some release testing.

- **Pros and Cons** (w.r.t. single ‘master’ branch which provides a single stream of releases):

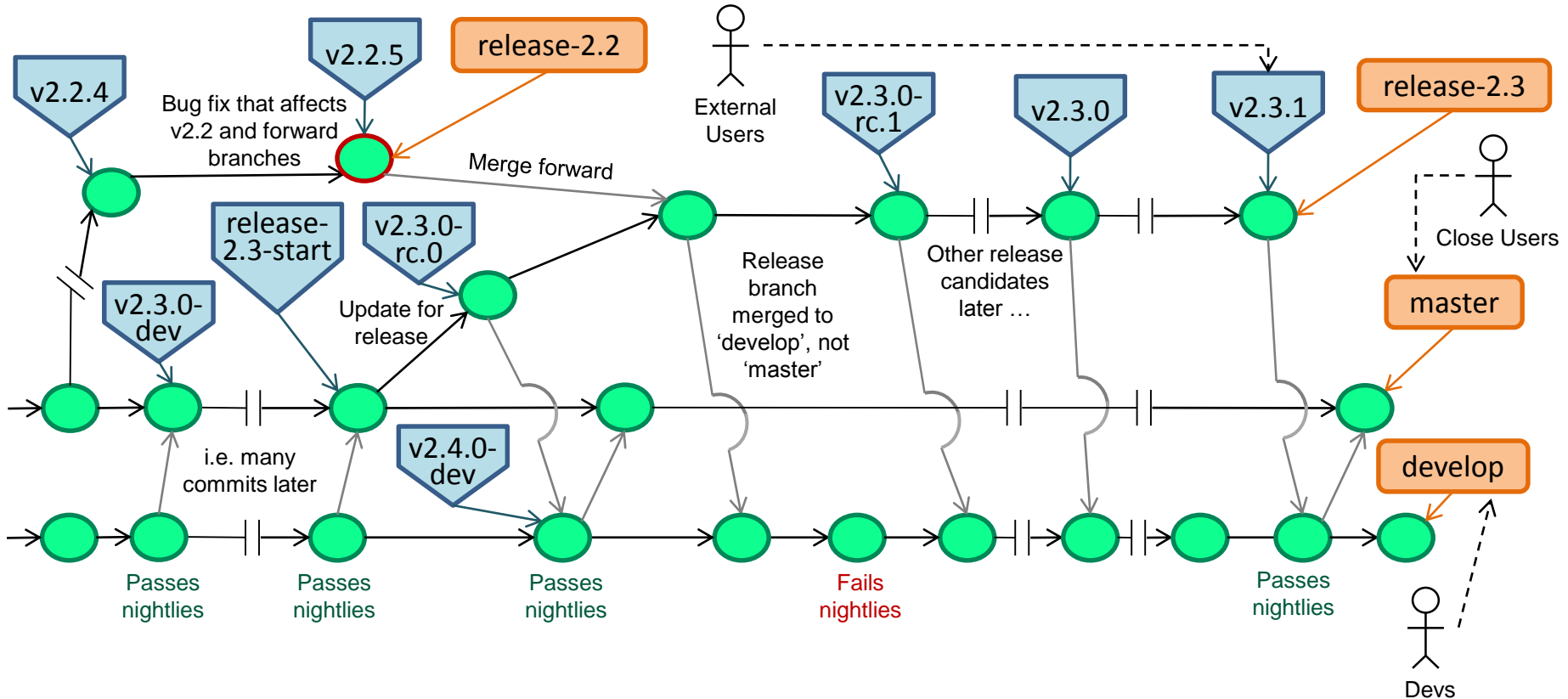
- **Pro:** Allow support for multiple releases
- **Pro:** Allows customers to depend on well-defined named versions of the software
- **Con:** More labor and more testing needed to maintain old releases

- **Example project:** Established project with many customers requiring stable named releases

- **See:** “gitworkflows(7)” <https://www.kernel.org/pub/software/scm/git/docs/gitworkflows.html>



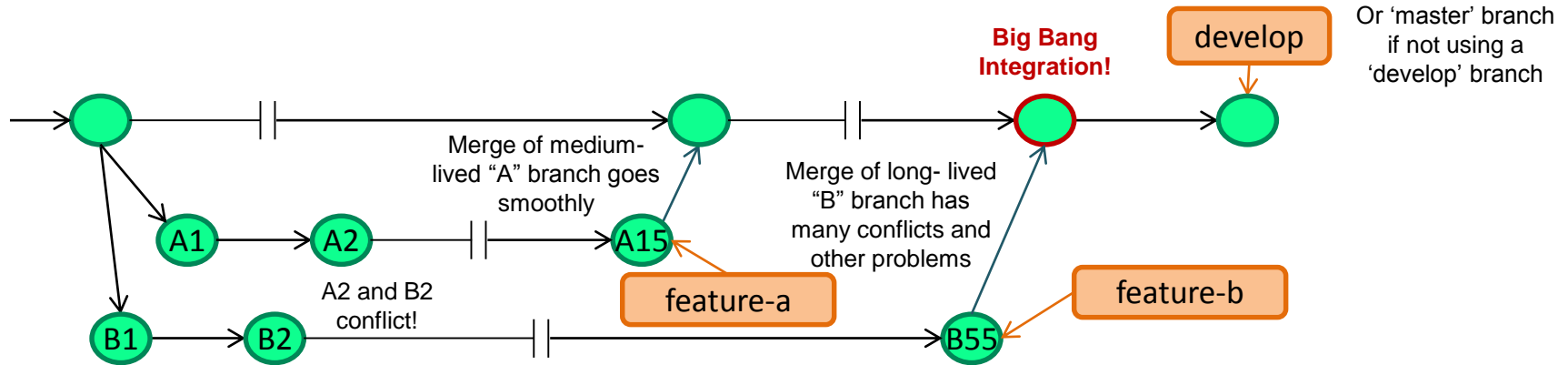
# Addition of release branches (\w 'develop')



- **Modifications to release workflow when using a 'develop' branch:**

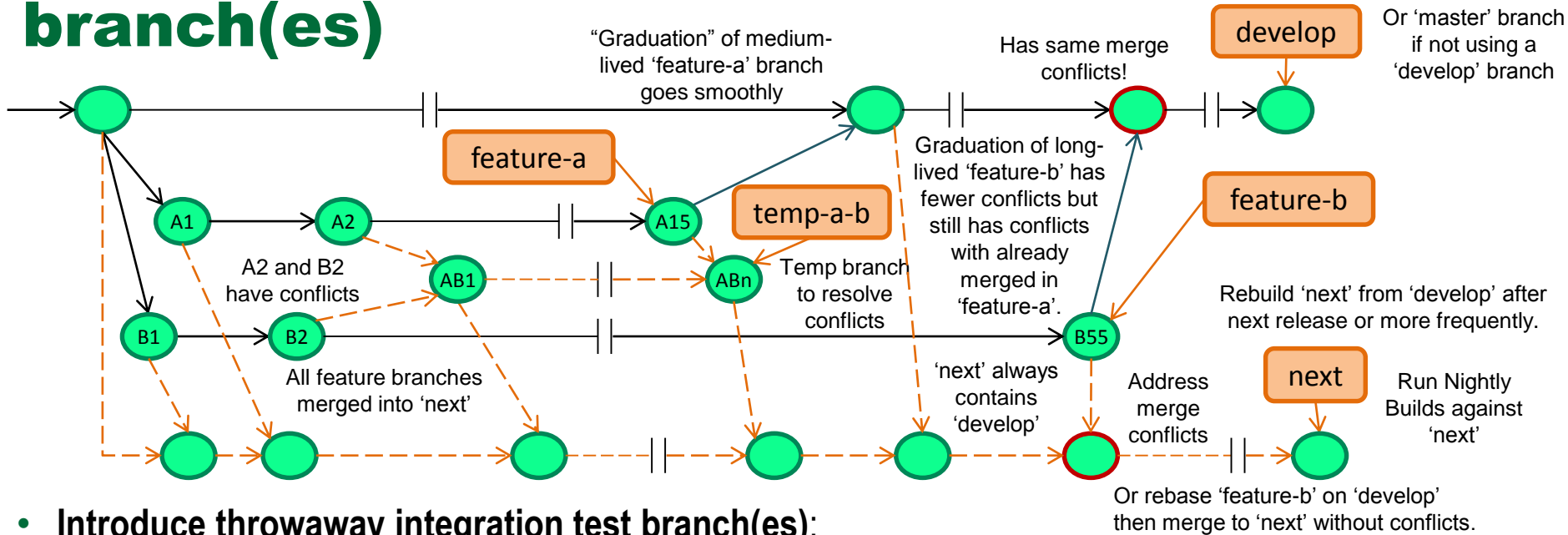
- Most recent release branch is merged to 'develop' (instead of 'master')
- The 'develop' (instead of 'master') branch is tagged for the next release with 'vX.Y.0-dev'

# Addition of feature branches



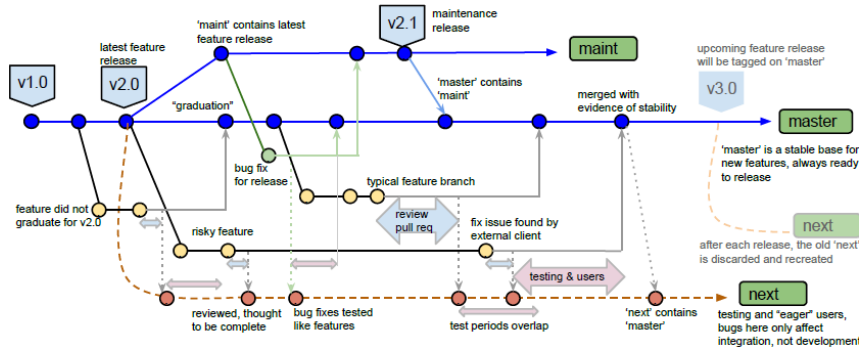
- **Introduce long-lived feature branches:**
  - Features completed in separate (long-lived) branches before single merge into 'develop'.
- **Pros and Cons** (w.r.t. short-lived topic branches):
  - **Pro:** Allow time for detailed code reviews before the changes are merged into 'develop'.
  - **Pro:** Accommodate less experienced developers who can't be trusted to directly push to 'develop'.
  - **Pro:** Accommodate changes from external developers who can't directly push to the 'develop'.
  - **Pro:** Handle risky changes that may never make it into 'master'
  - **Pro:** Keep very clean git history for each feature, merge commits become "changelog".
  - **Con:** Risk of major merge (or semantic) conflicts (i.e. **BIG BANG INTEGRATION, e.g. merge of "B"**)
  - **Con:** Not consistent with Agile best practice of CI (see "[Feature Branch](#)" by Martin Fowler).
  - **Con:** Discourages Agile best practice of **continuous refactoring** (refactoring makes merges difficult)
  - **Con:** Requires more testing resources to test each feature branch individually
- **Example project:** Established project with many external contributors and/or junior developers

# Addition of throwaway integration test branch(es)



- **Introduce throwaway integration test branch(es):**
  - Feature branches (FBs) passing **CI Build** merged into throwaway 'next' branch.
  - **Nightly Builds** run on 'next' branch every night.
  - When ready, FB **“graduates”** and merges into 'develop'. (i.e. **Subtractive test assumption of branches!**)
  - Use **throwaway conflict resolution branches** (e.g. 'temp-a-b') to resolve conflicts between feature branches
- **Pros and Cons** (w.r.t. stand-alone feature branches):
  - **Pro:** Incompatibles between feature branches are tested early and often
  - **Pro:** Multiple feature branches can be tested together, instead of individually, saving test computing resources
  - **Con:** Bad code in a single FB breaks all Nightly Builds run on 'next' branch (and no other FB gets tested).
  - **Con:** Hard to determine which FB is breaking a 'next' Nightly Build
  - **Con:** Have to resolve same conflicts twice! (i.e. merge “feature-b” to 'next' and 'develop') => **use git rerere?**
  - **Con:** More complex and labor intensive workflow!

# End: The Git.git Workflow “gitworkflows(7)”



## Going from “addition of throw-away integration test branches” to Git.git Flow


- Discard the ‘develop’ branch
- Feature branches created from and merged to ‘master’
- Full Git.git Flow also uses throw-away ‘pu’ branch!
- Current release branch is called ‘maint’, not ‘release’
- Old maintained release branches called ‘maint-X.Y.Z’.

In summary, Git.git Flow would be a good starting choice for any project where all of the members of the development team were very good with git and the portability is challenging. However, it comes at the cost of a more complex and labor-intensive development workflow.

The Git.git Workflow may be a good choice for projects when any of the following are true:

- Developers are git experts
- Code is of high consequence and responsible for basic security (e.g. git itself)
- Many changes being suggested in feature branches may never go into the final version
- Desire for a very clean git history
- Close users expect to pull working versions of the code from ‘master’ at any point in time
- Testing on any single platform (or small number of platforms) does not give sufficient confidence that there will not be major problems on other platforms.
- Developers use a heterogeneous set of development environments (e.g. Linux, PC, Mac, and various vendors and versions of compilers) and the code has portability issues.

# Summary

- Instead of defining complete workflows to choose from
    - ⇒ **Define workflow “building blocks” and construct the workflow that you need!**
  - Workflow Construction Steps:
    - Consider the properties and challenges for a given project
    - Construct simplest git workflow using building blocks to meet current needs
    - Add new features to workflow as situation changes and more challenges emerge
  - Workflow building blocks
    - Begin: The simple centralized CI workflow
    - Addition of a ‘develop’ branch
    - Addition of subteam branches
    - Addition of topic branches
    - Addition of release branches
    - Addition of feature branches
    - Addition of throw-away integration test branch(es)
    - End: Git.git Flow (e.g “gitworkflows(7)”) 
  - Next steps:
    - Git training (see [Git Tutorial and Reference Info](#))
- These can be added to a git workflow in almost any order!