Agile Source Code Management for Teams

JP White
Developer Advocate &
Cloud Engineer

Erin Perry
Developer Advocate &
Large Scale Agilist

Introduction

This document provides an example of how to use git and GitHub to manage source code in an agile environment with multiple teams. We'll be referencing the LeSS framework, but all of the principles apply regardless of scaling structure. We didn't invent this scheme - it is largely based on the work of Henrik Kniberg, Martin Fowler, and Vincent Driessen. We highly suggest you give those authors a read in advance.

We wrote this guide because we know that many teams are new to both git, GitHub, and the concepts around Agile source code management. We wanted something that could act as a guide to developers in both agile engineering practices as well as basic git usage within an agile context. This is our attempt at describing a model in an easy-to-read fashion that any team could consume.

Note: This is only one model, not a silver bullet. Each team will have to adapt this model to match their particular context.
Note: Hard rules will be explicitly called out. Everything else can be adapted.
Note: We will be establishing a working agreement throughout. We recommend all teams use this as a baseline for their own source control agreements.

Assumptions

- We assume you are familiar with the basic elements of team-level Kanban or Scrum
- We assume you are on a team, working with multiple teams on the same code base
- We assume you are comfortable with basic git commands and concepts.
  If you are new to git, we highly recommend the Pluralsight course “How Git Works”.

Goals and Principles

If a set of teams intends to work together in an agile way, the way they use their version control system needs to enable that. We have a set of goals and principles that underpin all of our recommendations here. If your strategies and behaviors enable this, you’re well on your way to agile development.

Integrate early, fail fast

- Discover issues with integration and conflicts in your code as soon as you possibly can to reduce the cost of resolving them.
- Keep changes and merges small to reduce the splash damage of problems.
  “*It’s better to fix small problems often than fix large problems seldom*.” -Kniberg
- Branching delays integration. Branch only when the motivation to do so is worth that cost.

Each iteration should create releasable work

- Unless something catastrophic happens, each iteration should produce at least one “done” thing.
- To recoup the benefits of agility and reduce lean wastes, “done” should be releasable. That means you can stage the work to be released.
- Releasable does not mean deployed. There are many reasons a team may choose to delay deployment, though each comes with trade-offs. But that should not stop you from continuing to produce new releases.
- This includes ensuring all relevant regression tests are passing. This is easily done if you’ve been building a net of automated functional tests. If your regression includes manual testing, you’ll need to be more selective with which of the tests are applicable to this change.

Simple rules at the right level

- Any policies and rules established for version control need to be lived every day. The simpler they are, the easier it will be for people to embrace and apply them.
- Very few rules should be centralized. Delegate as much as possible to teams; let them learn and grow largely at their own pace.
- Only set rules that are possible and practical. You may want to include everyone in a large code review for every pull request, but if that is impractical you will only serve to bottleneck delivery.

The Rules

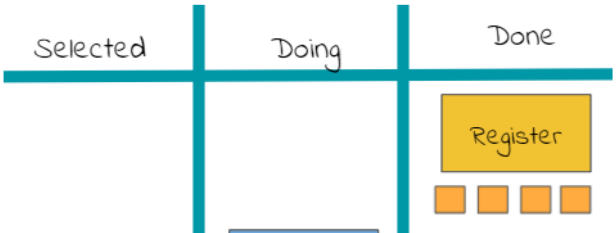
While we don't have many, there are a few hard rules we put into place. Let's take a look at each one.

Rule #1 - Each type of branch has a policy and owner(s), no exceptions.

Every branch will have a set of rules for what it is used for and when team members are allowed to merge their changes to it. For working branches, it's likely the owners of that policy (those in charge of making sure the branch is used correctly) are the development team members that created it. For eternal branches like main and `develop`, it's likely the owners are a community of practice formed from development team members. It's imperative that policies are owned by those that use them. In this case, developers. This is to ensure that decisions are made by people with first-hand knowledge, and skin in the game. The people setting the policies must be the same people feeling the pain of bad choices - motivated and empowered to fix things quickly.

Rule #2 - A Definition of Done is agreed to and enforced by everyone working on the code

All the teams that are working together on the same code base need to agree on what "Done" means. Each team can extend that definition, but all of them must agree on the baseline definition. Because "Done" means "releasable" to us, we're going to assume that the "Definition of Done" includes all of the things needed to make that true. For example - code reviews, passing tests, security scans, and quality checks.



You'll notice on this example board that we only include 3 backlog items for the team. That's

because we value completion over work in progress. Getting your work to “Done” before starting a new item helps reinforce this, and produces value faster.

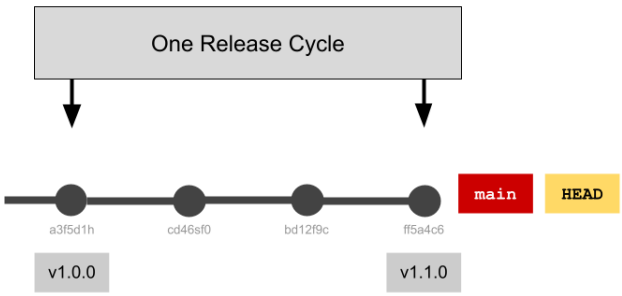
“Working software is the primary measure of progress” - Agile Manifesto

Workflow Overview

Let’s see how this works. We’re going to talk through some situations and standards we recommend. As we go, we’ll build up a working agreement.

The “Done” branch `[main]`

When a backlog item reaches “Done”, we need a spot to put it. That spot should be a branch where you can deploy from. We will be using `main` for this purpose. It becomes the source of truth for your most stable code, and acts as the default release branch. We recommend enabling branch protection - rejecting any direct commits, merges, deletions, or destructive history rewrites. All changes to `main` must be introduced through approved Pull Requests. The `main` branch should be incredibly stable. A failing build on this branch should be nearly impossible, since all integration and acceptance testing is done prior to merging to this branch. Remember, “done” includes regression testing as well.

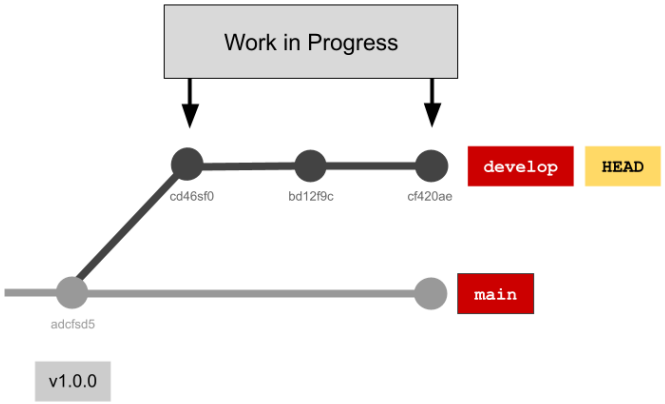


SCM Policy

- `main` will serve as the default release branch in CICD
- `main` represents only DONE work; it must always be releasable
- The build must be passing at all times; stop and fix otherwise
- No direct commits allowed. Everything goes through Pull requests

The “Integration” branch `[develop]`

We need a stable place to integrate our work together, a branch that represents the “LATEST” development. We will use the `develop` branch for this purpose. Changes pushed to the `develop` branch should be at a mostly stable state. The team should define exactly what that entails, but it normally means functionally complete, meets quality standards, and all tests are passing. These changes are very close to “Done”, but may not yet be releasable until some other activities are complete; advanced testing, security scanning, code signing, etc.



So, now we have two branches. A stable `main` branch where releasable work lives and one, slightly less stable, integration branch where teams merge their completed work together. The work on the `develop` branch is functionally complete and has passing tests, but not stable enough for release. We expect that the build on this branch will be fairly stable (mostly passing) if work is well-tested before it’s merged. But, since it hasn’t been integrated fully yet it’s possible for builds to fail. We recommend that teams adopt “stop and fix it” behaviour when it comes to broken builds. Whenever you break a build, you stop everything you are doing and fix it. Healthy teams never leave a build red and quickly resolve any build or test issues.

SCM Policy

- `main` will serve as the default release branch in CICD
- `main` represents only DONE work; it must always be releasable
- The build must be passing at all times; stop and fix otherwise
- No direct commits allowed. Everything goes through Pull requests
- `develop` is our primary integration branch, and serves as the mainline
- `develop` should be ready for review at all times
- All work pushed to `develop` must be complete with passing functional tests
- Must not have persistently failing build (mostly passing with the occasional failure)

When do we create additional branches?

Only when you’re motivated to. You may be motivated to branch because you want a place to work on something that’s not ready to integrate to `develop`. Or because you have several releases in progress at the same time.The simplest version of this is well-stated by Kniberg:

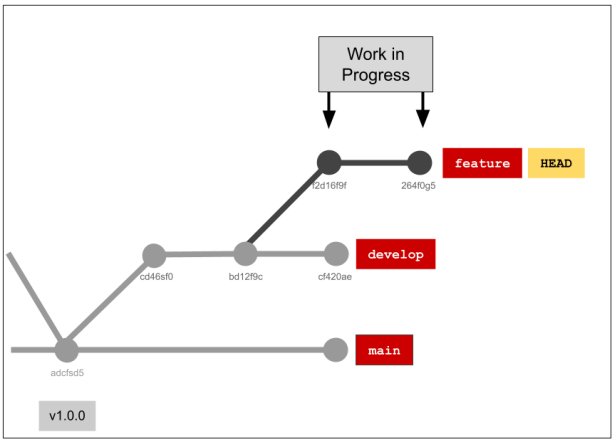
"Only create a new branch when you have something you want to check in, and there is no existing branch that you can use without violating its branch policy."

Feature branches (aka work branches)

You and your team want to work together on a change and commit and merge your changes frequently. But...those changes aren't "stable" yet. They won't be code complete and passing functional tests. It's still work in progress. With the current policy, you can't check that in to `develop` or `main`. So, you're motivated to create a new branch.

```
# The long way
git switch develop                    # Always start from this branch
git branch my-new-feature             # Create new branch
git switch my-new-feature             # Switch to the new branch

# The short way
git switch -c my-new-feature develop  # Does the same as above in one step

# Push the branch to the remote
git push --set-upstream origin my-new-feature
```

This branch exists for a team to integrate their work together until it reaches a state where it can be merged to develop. You can call this a work branch. Many people call this a feature branch, since most of the work that a team does is creation of new features. It can also be called a topic branch or a team branch. Because we're most comfortable with GitFlow, and that strategy calls them Feature Branches, that's what we'll use here.



(Note: This document assumes you are working on a shared code base with many teams. If your team is the only one working on the code, you may not be motivated to branch and could instead modify the policy and use `develop` as your unstable work-in-progress branch.)

These types of branches need policies as well.

SCM Policy

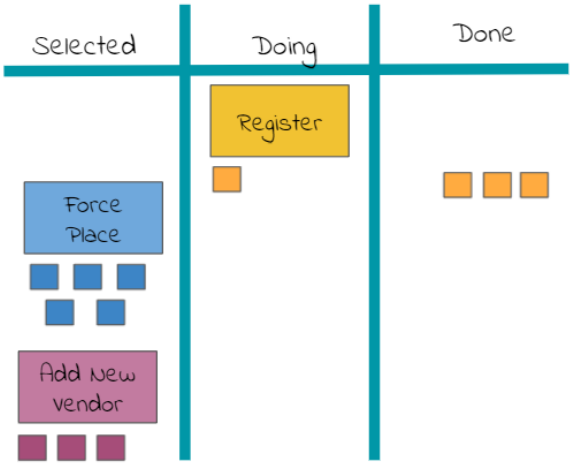
- `main` will serve as the default release branch in CICD

- `main` represents only DONE work; it must always be releasable
- The build must be passing at all times; stop and fix otherwise
- No direct commits allowed. Everything goes through Pull requests
- `develop` is our primary integration branch, and serves as the mainline
- `develop` should be ready for review at all times
- All work pushed to `develop` must be complete with passing functional tests
- Must not have persistently failing build (frequent red to green movement expected)
- Feature branches are created by a team when they start a work item
- Feature branches will be named `feature`* where * is a descriptive name of the work item
- Work pushed to feature branches will compile, build, and pass unit tests
- Feature branches are deleted after they have been merged

Pushing from Feature branches to develop

As a team works to complete a feature, it's going to reach a stable state where it's ready to merge into `develop`. According to our policy, that means that it is code complete, unit tested, functional tested, and ready for integration and other types of testing that can only be done with a merged code base.

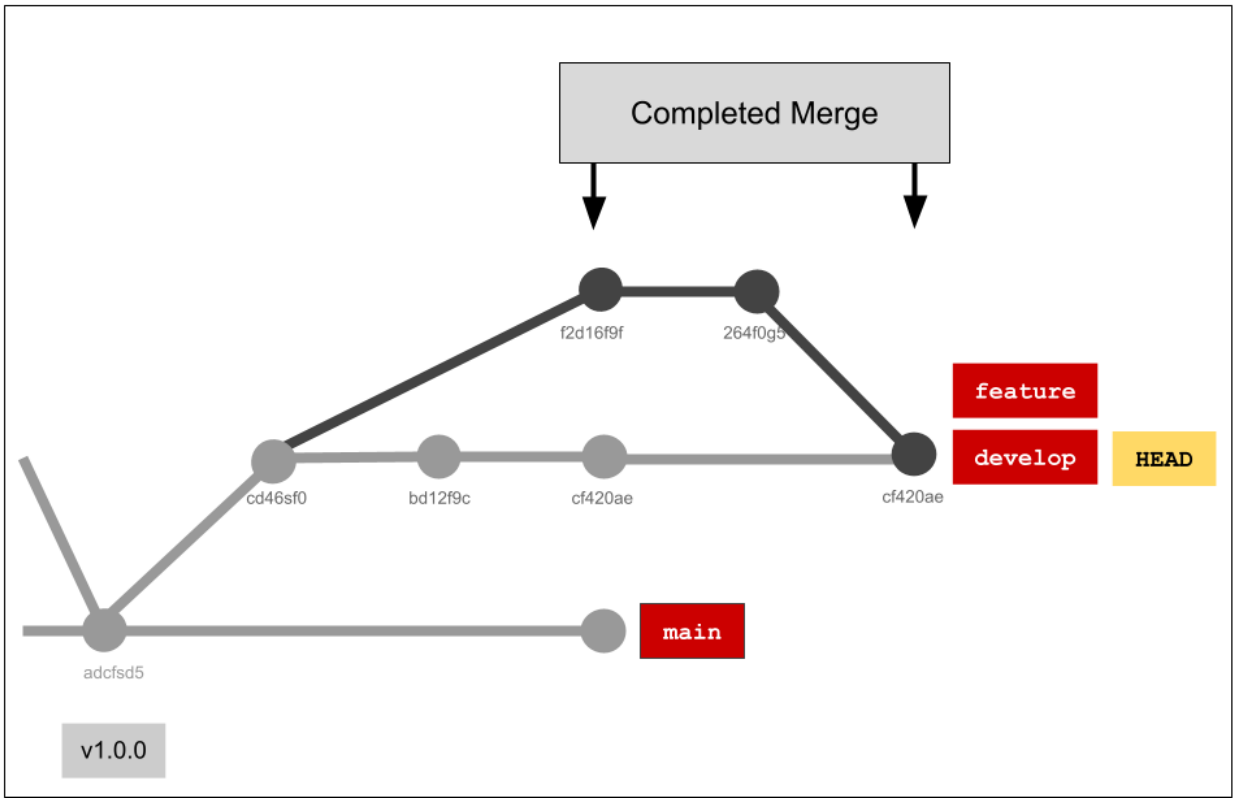
In this example, "Register" is in this state. It's not quite done, but everything is complete that can be done by the team prior to integration. Your task board would look like this:



Each of the small sticky notes represents a task. The large sticky note represents the entire backlog item. The task left in the "Doing" column is the integration and acceptance testing needed prior to reaching "Done."

Our team implements "Register", commits their work to the feature branch and ensures it's stable there. Then, merge it into `develop` when it's ready to integrate.

```
git switch develop                  # Switch to the target branch
git merge my-new-feature --no-ff    # Merge the source, using a merge commit
git push                            # Push any changes to the remote
```



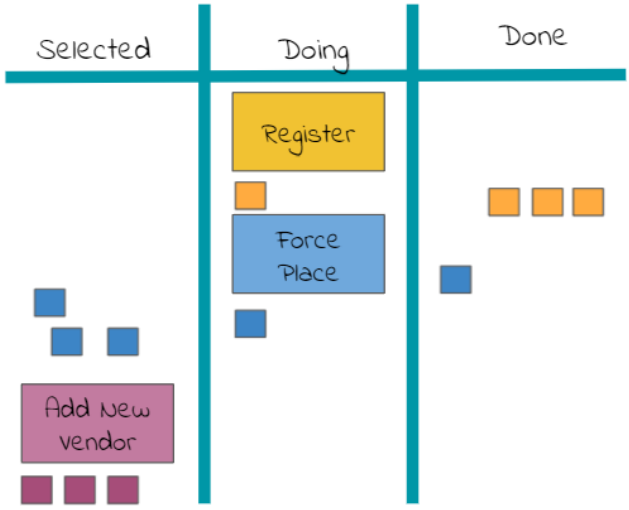

```
git branch -d my-merged-branch       # Delete the local branch you merged
git push -d origin my-merged-branch  # Delete the remote side as well
```

An ideal state team is limiting their Work in Progress (WIP) and finishing one item before moving on to the next one. But...what if your team worked on several items in parallel? Or, even if you're working with a WIP limit of 1, what if other teams pushed to `develop` as well?

Let's see!

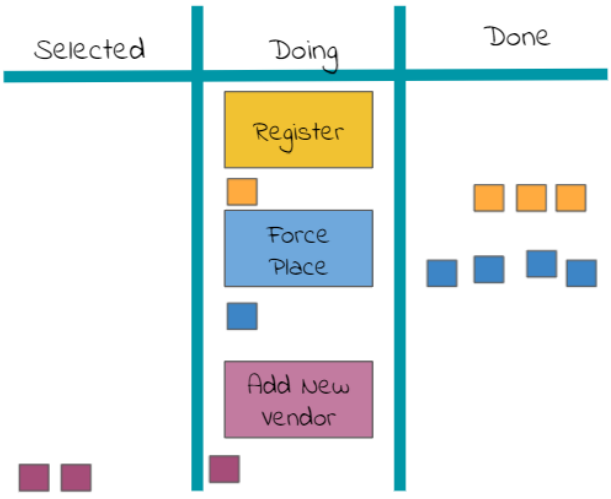
What if our team implements multiple backlog items in parallel?

If your team focuses on getting to "Done" for one item at a time, merging your work is trivial. As soon as it's complete, you merge your `feature` branch to `develop`. But, what if you are just finishing up "Register" while your team starts "Force Place?"



If we merged our branch to `develop` right now, we'd also include the partially completed Force Place work and violate our policy for `develop`! We COULD wait for that work to complete on Force Place…

Ugh! "Force Place" is ready to merge, but someone on the team started work on "Add New Vendor" and we can't merge that yet…



We could wait for all of the backlog items to be complete and THEN merge towards the end of our sprint. But, that means we haven't integration tested our work until the last minute. If integration tests fail, we won't know which of our items caused it. We also won't get any of that valuable feedback while there's time to course correct. If we don't find out things aren't working until the last day of the sprint...we'll have no time to fix it and we'll end up with nothing releasable!

Waiting doesn't help us at all. Instead, we end up saving up for a big bang release. So, what do we do instead?

Strategies for handling parallel work

- Don't do it. WIP limit your team to 1 item at a time. Focus on getting the highest priority item DONE and delivering value before you move on to the next item.
- Create a separate feature branch for every backlog item. To avoid waste and encourage whole-team behaviors, only create the feature branches when you start the backlog item. (Don't pre-create feature branches).

Scrum is called Scrum because the whole team is expected to come together to move things forward. A healthy team should be completing things together before moving onto the next. Humans, however, have an expectation that doing many things in parallel is more efficient. It gives a nice feeling of speed. It may be tempting to integrate and merge all at once and try to "bundle the pain." However, all you're doing is magnifying the pain of integration and making it more likely you'll cut corners at end-of-sprint "crunch time." The reason Scrum teams are small (fewer than 9 people) is so they can collaborate and focus their efforts together. If everyone has their own branch and is working on their own items...there's not a lot of collaboration happening.

It's entirely likely that, as you are finishing one backlog item, a few people may be starting a new one. But, the bulk of the team's effort should be concentrated together on the highest priority.

Since this article is about scaled Scrum, we fully expect that each team is working on a different item, in parallel. We'll look at that in a second. But first, let's see what you do when your work conflicts with someone else on your team.

Diverging code (merging conflicts)

Sam is writing code that calls the Widget class. About an hour ago, her team-mate, Jim, removed the Widge class during refactoring. That's not a simple conflict to merge. Imagine if Sam kept coding along for another few days on a local branch, waiting to merge her code. She'd have no idea there was a conflict, and resolving it would be increasingly difficult and expensive.

> *Working Agreement: Synchronize your code with others as often as possible, without violating branch policies.*

Synchronize means to both pull down code from a remote branch, as well as push your changes to that branch. A pull means "catching up" your local copy to get all the changes others have checked-in to that branch. Pushing means making all of your changes available to everyone else.

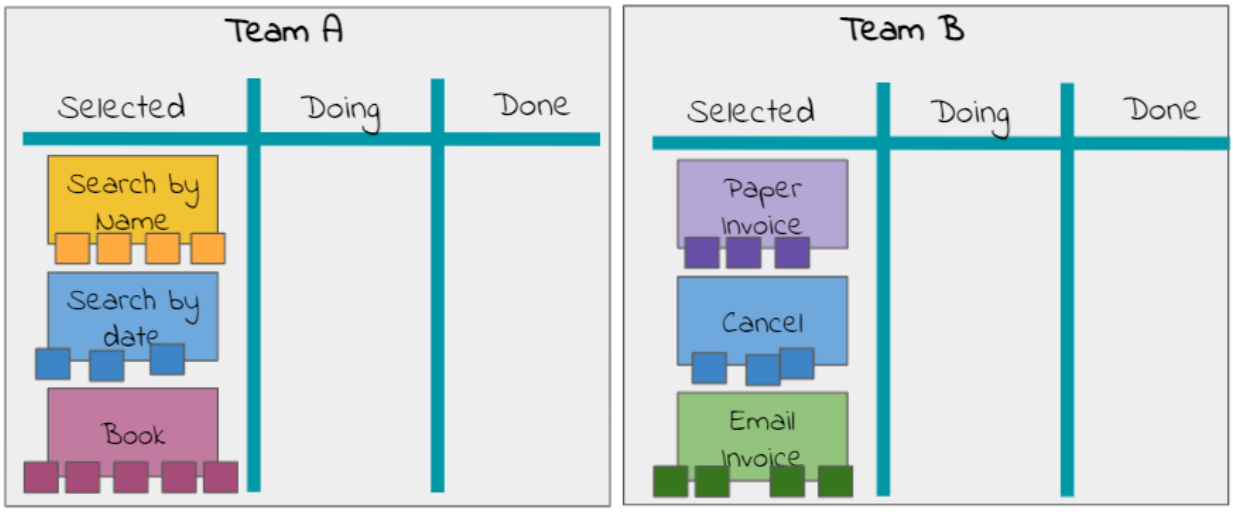
You should pull very frequently, to learn early and often if you have any conflicting changes. Healthy teams communicate constantly when they've pushed changes so that others on the

team can pull those changes down. You should push changes to a feature branch whenever you have passing unit tests. If you're following test-driven-development, that should be several times an hour. Even if not, you should aim to complete small units of work, ensure unit tests and written and passing, and aim to push about once an hour.

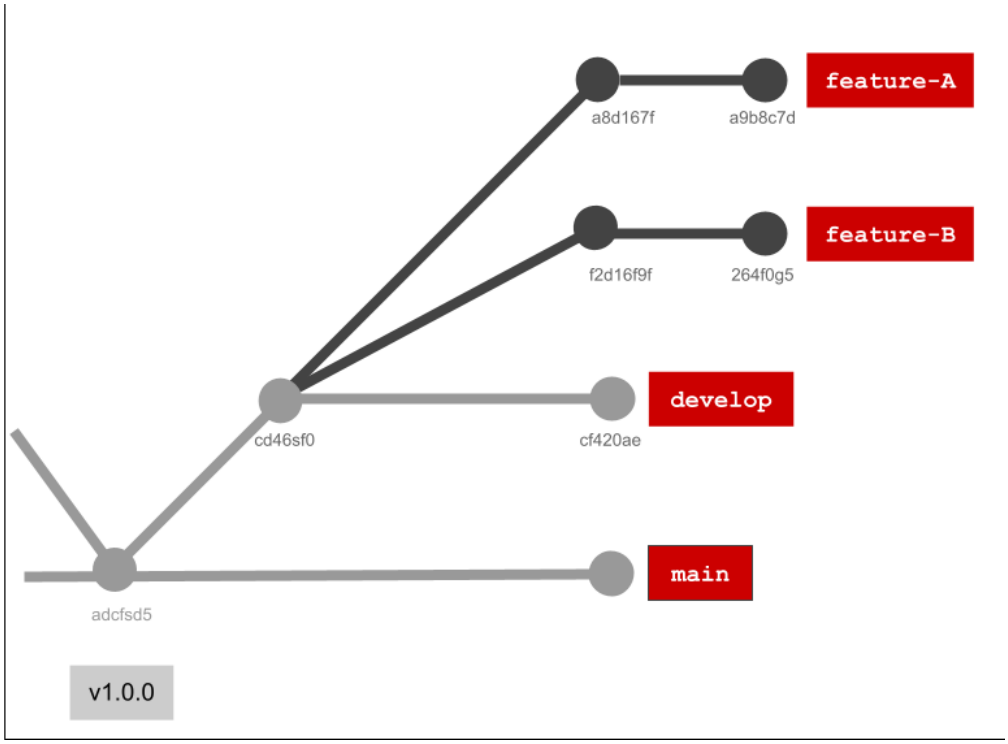
For many developers, team-level synchronization sounds obvious. But, we'll be expanding this same behavior to large-scale development with many teams as well.

Multiple teams - many teams working on the same code base in parallel

The principles here apply to situations where dozens of teams may be working on the same code base. Let's start with a simpler example, though: TeamA and TeamB. Both teams are cross-functional feature teams working on a hotel system. TeamA is currently working on booking reservations. TeamB is focusing on accounting. They've just finished sprint planning and are about to start their first backlog items.



TeamA starts "Search by Name" and creates a feature branch for it. TeamB starts "Paper Invoice" and starts their own feature branch for that.



Now things get interesting. Sam is in TeamA, working on her team's feature branch. While her team is working, changes could be made to `develop` and even `main` without being merged into her feature branch first. How could that happen? Well, because TeamB is out there working at their own pace, and they may have reached "done" with one of the features. At any given moment, there may be changes on `develop` and `main` that Sam doesn't know about. And that code might...ugh...conflict with her changes. TeamB might've renamed a class or refactored a method Sam is using. Just like someone on Sam's team could make changes that conflict with hers, other teams can too. And that gets even harder to account for, since cross-team communication is less frequent than intra-team communication. We can use a similar solution across teams to the one we used within a team, frequent integration.

Working Agreement: Merge from develop to your work branch at least once a day

For Sam's team, they've decided to do this every morning. Someone in TeamA is responsible for pulling down from `develop` into their feature branch and resolving any conflicts immediately. If a conflict needs help from TeamB to resolve, they contact them as soon as they can and work together to resolve the problem. Pull frequently from `develop` into a feature branch allows the team to resolve conflicts, and the instability they bring, on their working branch, rather than on `develop`. Since `develop` is shared between all teams, instability on that branch can affect everyone. Resolving issues on a feature branch first helps contain that pain.

Working Agreement: Resolve conflicts on the branch that is least stable

Frequently pulling from `develop` is only useful if people are frequently pushing done work to `develop`. The longer individuals and teams wait to push their work, the longer potential conflicts remain invisible to others.

> *Working Agreement: Merge feature branches to `develop` the moment an item is done. Don't wait until the end of the sprint!*

This works best if the team works together to complete an item early in the sprint before starting the next item. Teams that work as individuals, each taking a separate item and merging all their work at the end of the sprint, will encounter the most conflicts and reap few of the benefits of team-based work...the very thing Scrum was designed for.

Notice that we had TeamA in charge of resolving the conflict. That's because, in their working agreement, they agreed that whomever checks in first to `develop` is the source of truth. All other teams are responsible for figuring out how to integrate their changes.

> *Working Agreement*: Whomever checks in first to `develop` wins. All other teams are responsible for pulling changes from `develop` into their working branches and resolving conflicts, though the originating team will help as needed.

Notice that this working agreement encourages people to get to "done" quickly. And, since working software is the primary measure of progress, that's great news for agility!

Let's see what a sprint might look like for many teams working on the same code base.

Day	**TeamA**	**TeamB**	**develop**	**main**
1	After Sprint Planning, Start *Search by Name*. Create feature branch from `develop`. We're the leading team for this release, so we'll make sure any release-prep is complete during the sprint as well.	After Sprint Planning, Start *Paper Invoice*. Create feature branch from `develop`.	Nothing	Nothing
2	Finished "*Search by Name".* Merge to `develop` and let other teams know. Delete that feature branch and create a new one for *Search by Date*.	Pull from `develop` and ensure TeamA's feature works with our code. Keep working on *Paper Invoice.*	*Search by Name* is here now	Nothing
3	Keep working on Search by Date	Finished *Paper Invoice*. Merge to `develop`. Delete feature branch.	*Search by Name and Paper Invoice* is here	Nothing

		Start *Cancel* with a new feature branch.	now	
...				
9	Finish *Book*, merge it to `develop`. Make sure all tests are passing.	Pull from `develop` and ensure merged features work with our code. Keep working on *Email Invoice*.	*Search by Name, Paper Invoice, Search by Date, Cancel, and Book are here*	Nothing
10	If PO and rest of the team confirms readiness to release, create a versioned tag for release, open pull request from `develop` to `main`. Ensure all tests and status checks pass. Work with the other teams if anything needs resolved. Make sure all feature branches have been deleted after they were merged.	Same as yesterday. Didn't finish "*Email Invoice"*.	Same as yesterday	*Search by Name, Paper Invoice, Search by Date, Cancel, and Book are here*

The sprint is all done. All the backlog items except *Email Invoice* are done. Good news, though, we can still release. Everything on `develop` was merged and integrated throughout the sprint. We quickly found any integration issues and were able to resolve them, making the end of the sprint far less stressful. And, Email Invoice is on its own isolated feature branch where we can pick up work on it in a future sprint if it's prioritized.

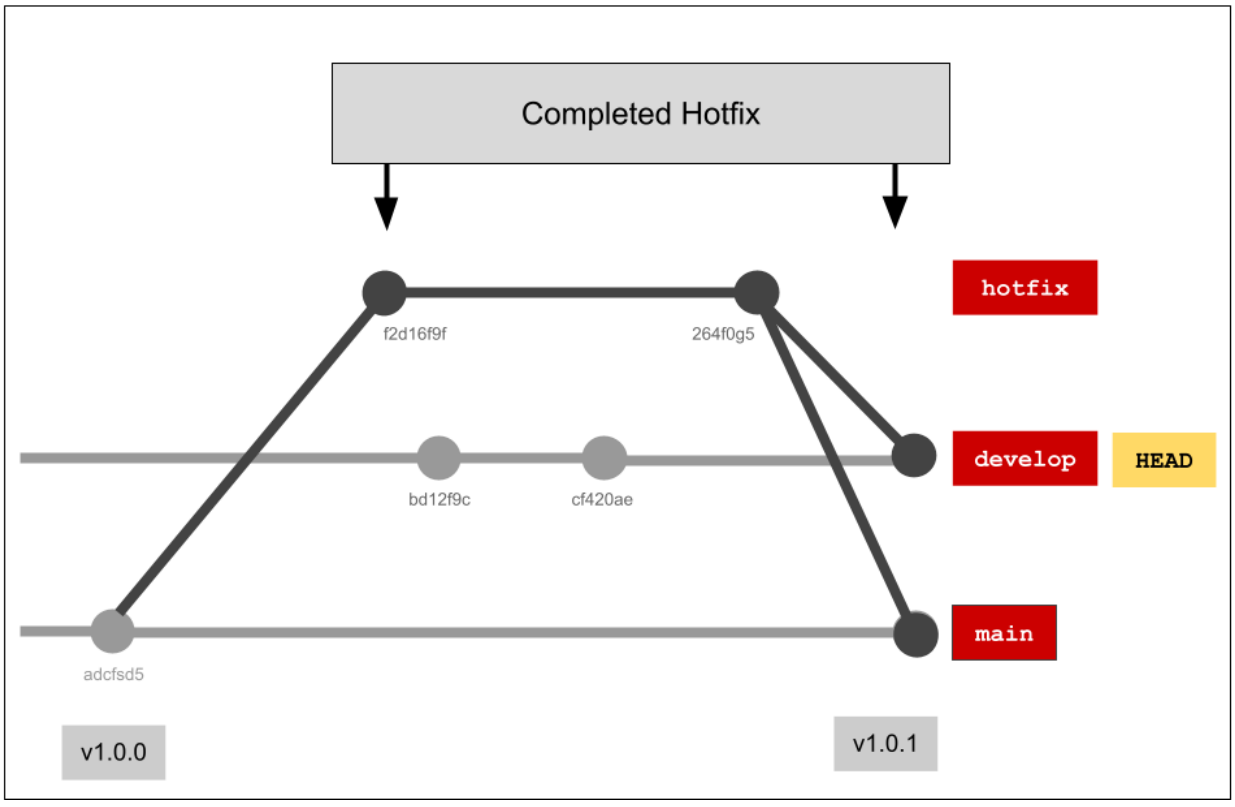
Hotfix branches

So, we've just finished a sprint and have version 1.0.0 of the system tagged, released, and deployed. Now, we're in the middle of the next sprint, but a serious defect is reported on the last released version. We've gotta fix it, but `develop` is "dirty" now with merged changes from our current sprint that may not be ready for deployment yet. So, what do we do?

1. Create a hotfix branch following our naming convention (starting the name with hotfix, followed by some descriptive identifier. This team is going to use hotfix-JXL1384, which includes the ticket associated with the defect.
2. Resolve the defect on the hotfix branch
3. Tag for release and create a pull request from the hotfix branch to `main` to release it.
4. Merge from `main` to `develop` so the patch is integrated into current work.
5. Delete the hotfix branch after it has been merged to `main`.

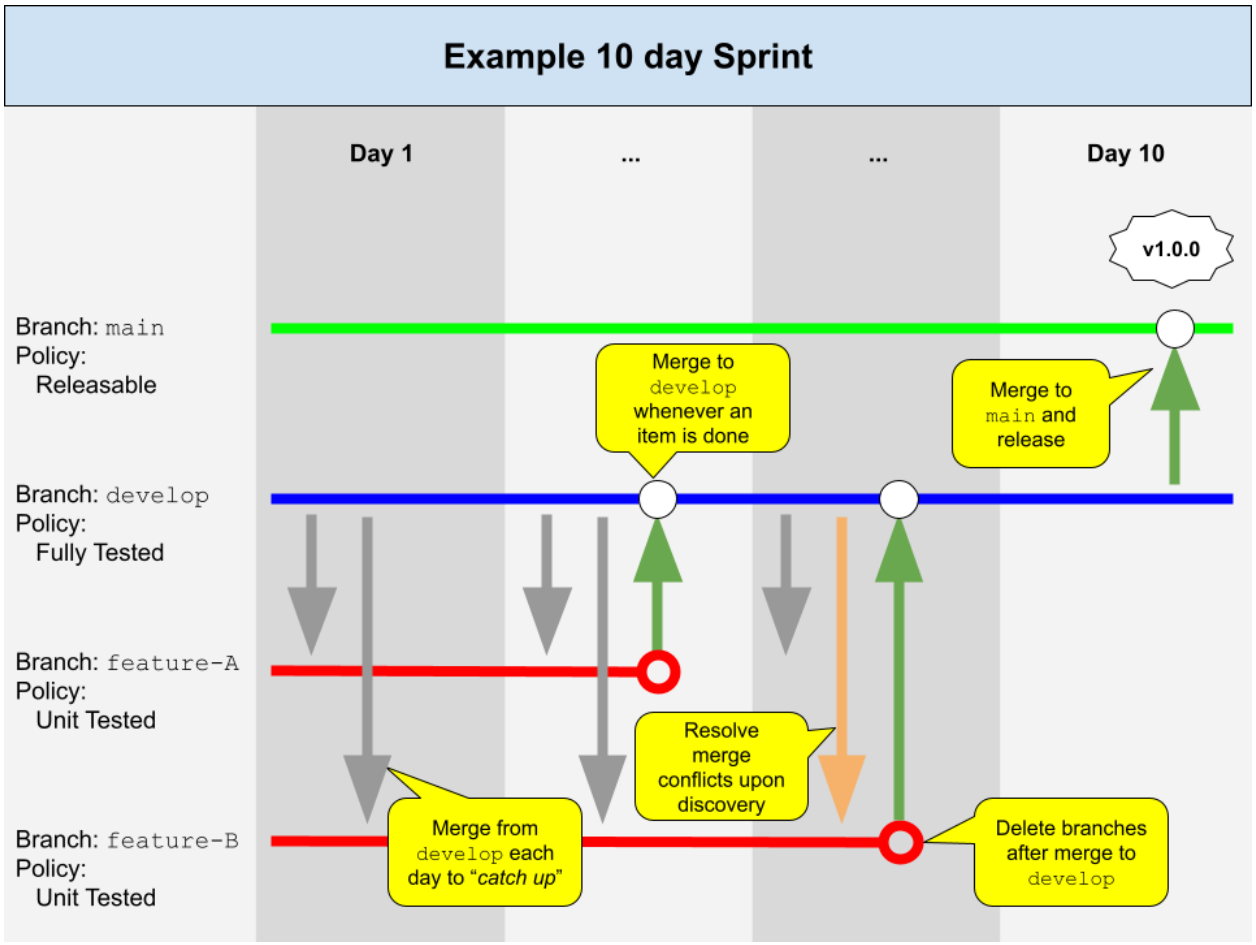
Working Agreement: Create and agree on naming conventions for branches. All teams that work on the code base follow the same convention.

```
git checkout v1.0.0                    # Checkout the previous release tag
git switch -c hotfix-branch-name       # Create the hotfix branch
git push -s origin hotfix-branch-name  # Push this branch to the remote
...                                    # Do work and make commits

git checkout develop                   # Checkout develop
git merge hotfix-branch-name           # Merge your hotfix into develop

git checkout main                      # Checkout main
git merge hotfix-branch-name           # Merge your hotfix into main
git tag v1.0.1                         # Tag this commit with a new version
git push --tags                        # Push the tag to the remote

git branch -d hotfix-branch-name       # Delete the local branch you merged
git push -d origin hotfix-branch-name  # Delete the remote side as well
```



The big picture

Ok...so let's see if we can put this all together.



SCM Policy

- `main` will serve as the default release branch in CICD
- `main` represents only DONE work; it must always be releasable
- The build must be passing at all times; stop and fix otherwise
- No direct commits allowed. Everything goes through Pull requests
- `develop` is our primary integration branch, and serves as the mainline
- `develop` should be ready for review at all times
- All work pushed to `develop` must be complete with passing functional tests
- Must not have persistently failing build (frequent red to green movement expected)
- Feature branches are created by a team when they start a work item
- Feature branches will be named `feature`* where * is a descriptive name of the work item
- Work pushed to feature branches will compile, build, and pass unit tests
- Feature branches are deleted after they have been merged

- Hotfix branches are created as needed to patch a previous release while new development is in progress
- Hotfix branches will be named hotfix* where * is a descriptive name of the work item
- Hotfix branches are deleted after they have been merged

Working Agreement:

- Each type of branch has a policy and owner(s), no exceptions.
- A Definition of Done is agreed to and enforced by everyone working on the code
- Only create a new branch when you have something you want to check in, and there is no existing branch that you can use without violating its branch policy.
- Synchronize your code with others as often as possible, without violating branch policies.
- Merge from `develop` to your work branch at least once a day
- Resolve conflicts on the branch that is least stable
- Merge from feature branches to `develop` on a regular basis, whenever an item is done. Don't wait until the end of the sprint!
- Whomever checks in first to `develop` wins. All other teams are responsible for pulling changes from `develop` into their working branches and resolving conflicts, though the originating team will help as needed.
- Create and agree on naming conventions for branches. All teams that work on the code base follow the same convention.

Variations

Teams can have their own branch instead of feature branches

Teams can decide to keep a long-lived branch for their team and integrate all their work there, rather than creating a branch for each work item. It can simplify branching, but without some of the clarity that isolating features can give you.

Release when done

Our examples here illustrate staging a release at the end of a sprint, but it's even better if you can find ways to release as soon as things are done. Release early and often. Working software is the primary measure of progress!

Run long-running regression and end to end tests separately

If some of your tests are long or complicated, you can choose to run those tests on a nightly schedule, or to run as part of the pull request to `main`. Any delay to testing and integration increases the risk and expense of resolving issues, but it may be worth it if it encourages your teams to check-in and build frequently.

Release branches

Let's say your group has 15 features in this sprint. Your Product Owner, however, has let you know that 7 of those features will be deployed right away, but 8 features should be staged for a future deployment. Maybe they want to do some marketing or training for their users in a pre-prod environment before those features go live. Maybe those features need some protracted user checkout or end to end testing with other systems. None of these situations is ideal, but they can and will occur. Even though every feature should be "releasable" at the end of the iteration, not all of them will be ready for deployment. We could use feature toggles to deploy all of the features and only turn certain ones "on." That solution may work, but it comes with complexity and perhaps your system isn't set up or ready for that type of deployment. Another option is to continue to use `develop` as illustrated above, to integrate all changes together for fast feedback, and to use release branches to group and stage the features that will be released and deployed together.

Take note: the need for release branches is an indicator that something is not ideal. For example, release cycles are very long, or testing is manual, or the definition of done is weak. There is no simple solution to a complex, messy problem. The more release branches you have, the more complicated and messy it becomes. There are few shortcuts to fix that. While this strategy will help you cope, resolving the core issue is the only way to simplify your development and become more agile.

So...how do you do it?

1. Create two release branches, one for each feature set that will deploy together. Follow a naming convention such as release-booking and release-invoices.
2. As work reaches "done", merge it to `develop` for complete integration testing and fast feedback. Do not, however, pull down from `develop` into the feature branches, since you'll end up merging in code from features you will not release with.
3. As work reaches "done", merge it into the release branch it belongs with for integration with the featureset that will be released together. You can and should merge from the release branch into your feature branch to resolve any integration issues on your feature branch prior to pushing to the release branch.
4. When a featureset is ready for release, tag and open a pull request from the release branch into `main`. We recommend merging it to `develop` as well, just in case any fixes or patches were applied to the release branch and it is now ahead of `develop`.
5. Delete the feature branch after it has been merged to `main`.

Definitions

These are not the only definitions of these words, but they do help clarify what *we* mean when we use them.

Policy - A published set of rules for a branch that everyone agrees to follow

Working Agreement - A collection of rules, practices, and behaviors that people agree to follow.
Root branch - A branch that has no parents. It's created entirely on its own and has no previous history.
Integration branch - A branch whose primary purpose is to allow people to combine their work together.
Eternal branch - A branch that is intended to be very long-lived
Working branch - A branch created for a person or small group of people to complete a unit of work in isolation
Local branch - A branch that exists only on a single person's workstation.
Remote branch - A branch that exists on a git server. Your local workstation is able to synchronize with it.
Feature branch - A branch that is created to isolate a unit of work while developers work on it, with the intention of merging that work back to an integration branch when it is done. We are using the naming convention from GitFlow and calling this a feature branch, even though "feature" is only one kind of work that can be done this way. Bug fixes, enhancements, and even refactoring can be done using a feature branch.
SCM - Source Code Management. A system for storing, maintaining, and versioning artifacts.
Definition of Done - Borrowed from Scrum, the DoD is the agreed to list of things that must be true for the development team to consider an item complete. Across a code base, teams should agree to a baseline shared definition. Individual teams can extend, but not modify it.
Backlog item - A backlog consists of work to be done. A backlog item is one piece of work. It might be a new feature, an enhancement, a resolution to a defect, or even resolution of technical debt.
Increment - Each time you complete a change to your product, you've created a new increment. It's called an increment, since you're adding to it through incremental change.
Iteration - A small, repeatable, time-box where work is done. In Scrum, the Sprint is an iteration.
Release - To bundle and label a set of changes that are ready for deployment
Deploy - The act of moving a release into an environment
Tag - A reference to a specific commit. Tags are often used to annotate a point in time. We recommend tagging prior to release with a semantic version number.
Hotfix - A change that must be made to a previous release and cannot wait for the next one. This normally occurs due to some emergency defect.
Featureset - A collection of changes that will be released together.