**Interconnection between VCS workflow and CI/CD**

Let’s review the main things we need CI/CD for:

* Running tests.
* Deploying/releasing.

But wait, we can do all this manually, so what is the real reason to use it? The reason is that we can specify **conditions** under which these things will be done **automatically**.

This brings us these great benefits:

* Tests are guaranteed to run, which brings us more confidence in the released product and more confidence for a developer in his code.
* Because running tests is no longer the developer’s responsibility (in most cases), there are fewer things for him to keep in mind. This allows him to be more focused and less stressed.
* The feedback loop can be as fast as we want it to be, which allows us to quickly fix problems.
* Releases involve much less work, preparation and risk because deployment is automated and all the problems/conflicts are resolved beforehand.
* Since releases become less difficult to manage, they can be much more frequent. This brings us fast iterations and quick user feedback.

Sounds great, right? But let’s recall the reason to use CI/CD we talked about. I mentioned two main words — “conditions” and “automatic”. Guess, who is responsible for the first? It’s your VCS! Or rather its workflow.

Your VCS model defines which conditions for running various automated tasks can be possibly set up. Of course, this is also dependent on the feature set of the chosen CI solution. But they tend to provide similar functionality, only in different forms.

Here’s how the questions that you have to ask yourself when designing your development workflow might look:

* At what point do you want to push out new releases? How quickly do you need to iterate? Do you have multiple distributing channels for your application (e.g. alpha, beta, stable)?
* Do you have specific tests that you want to be run only for certain states of your codebase? Or just bigger and smaller test suites that you want to run less and more frequent, respectively?
* How often do you want to run tests? Do you need a fast feedback loop with frequent tests for reliability?
* Or is your priority to move forward quickly, and so, developers need to spend less time waiting for CI builds to finish?
* Do you have any time or capacity constraints for running your builds? Is there a way you can optimize these things using some specific VCS setup?

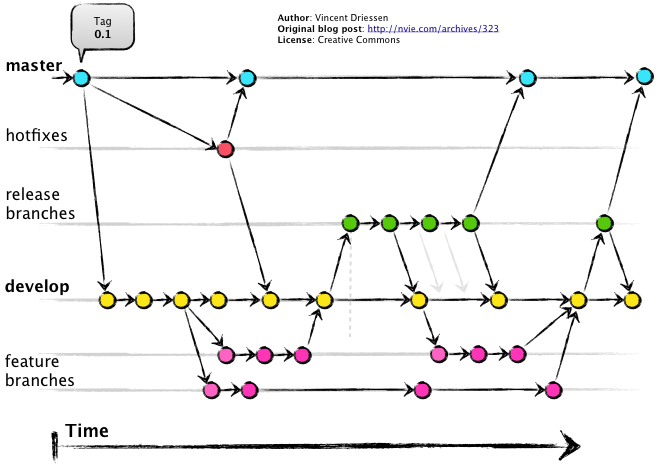
Keeping all this in mind and understanding the importance of your workflow design, let’s look at the choices we have when using the most popular VCS — Git.

**Dissecting Git workflows**

Here we are only going to look at various branching models and how they go along with CI/CD. Discussions pertaining merge vs rebase vs s̶q̶u̶a̶n̶c̶h̶ squash are out of the scope of this article as this doesn’t really affect our CI capabilities.

[**Gitflow**](https://nvie.com/posts/a-successful-git-branching-model/)

https://miro.medium.com/max/45/1*HIJnX63BtWVYibcRWaei5A.png?q=20



Gitflow, Image by [Vincent Driessen](https://nvie.com/about/)

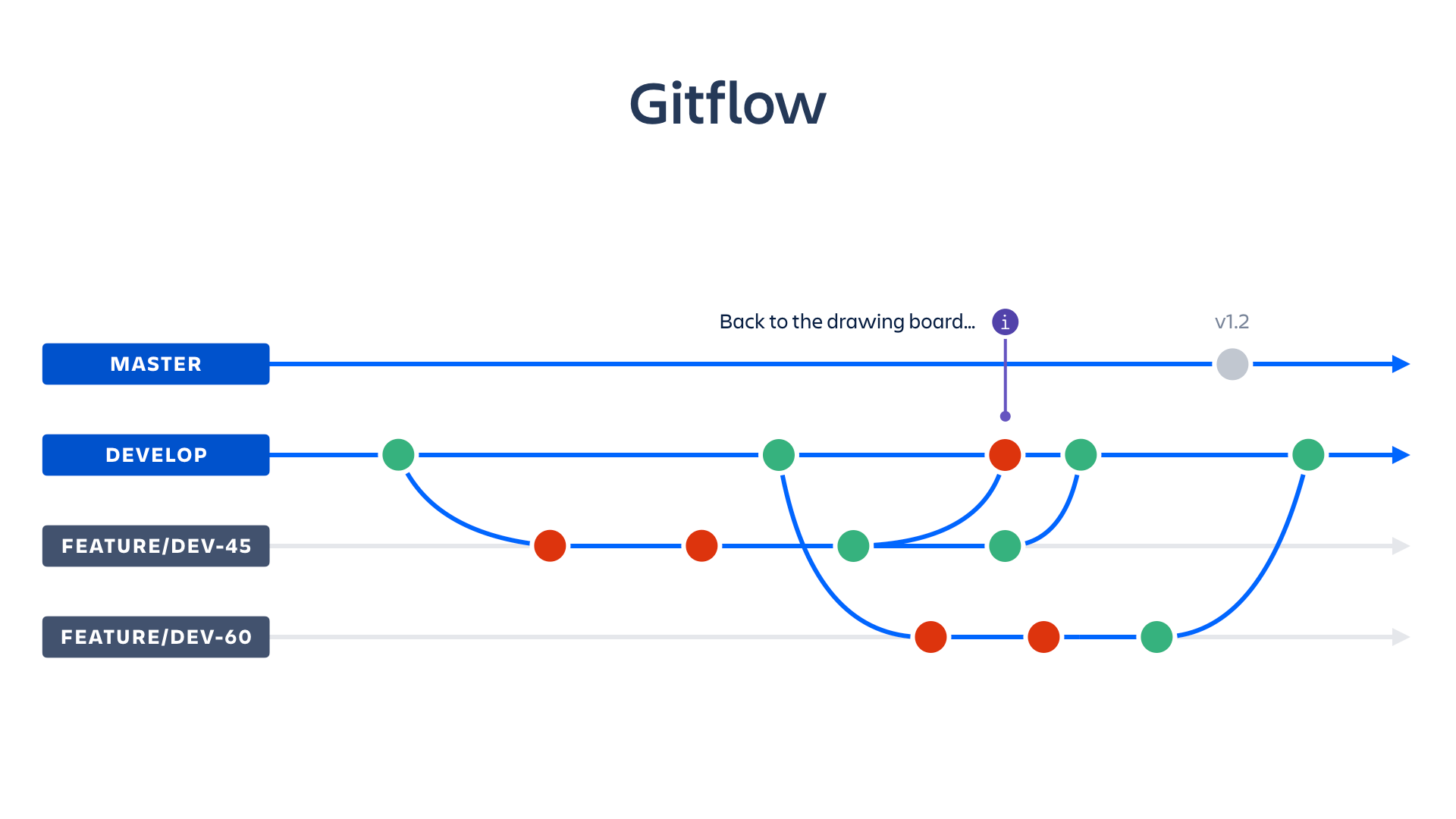
Gitflow is a very popular workflow that defines the following types of branches:

* **master:** contains production code.
* **develop:**contains the latest development changes that will be included in the next release.
* **feature:**a new branch is created for each new feature we work on. We start it from *develop*and once we’re done, merge back into it.
* **release:**starts from *develop* and signifies that there will be a new release once we merge this branch into *master*.
* **hotfix:**used when we need to deliver urgent changes to our production app but *develop*is not yet ready to produce a *release*branch. Starts from *master* and merges into both *master*and *develop*.

A basic CI/CD setup with this workflow would look like this:

* Run automated tests on all branches except master.
* Deploy after every merge into *master.*

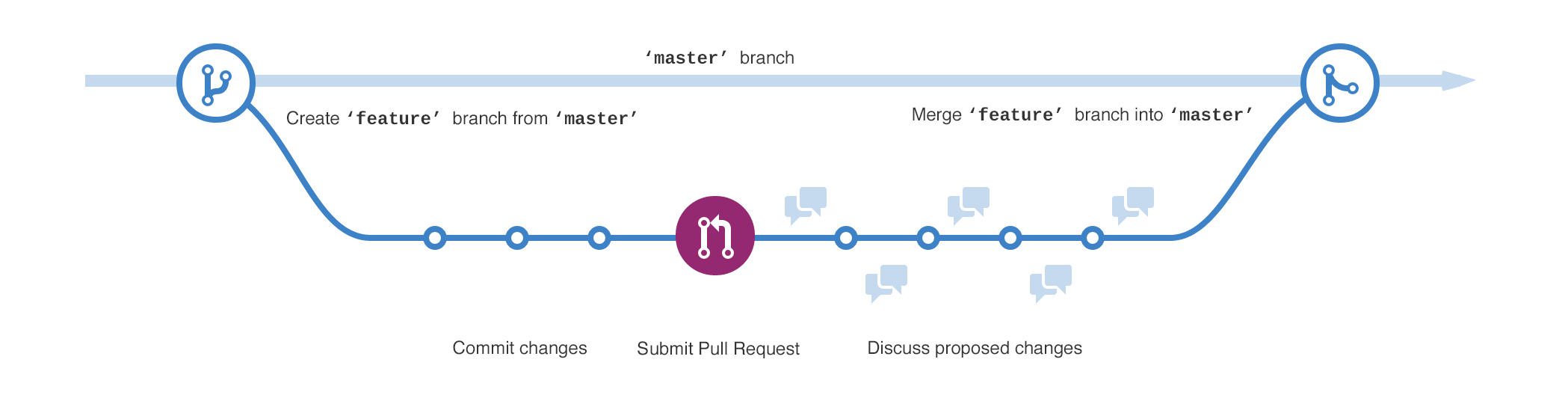
https://miro.medium.com/max/45/1*sA61YWkC2CQZLBMdHIobjg.png?q=20



Gitflow + CI/CD, [original article](https://www.atlassian.com/continuous-delivery/principles/workflows-with-feature-branching-and-gitflow) by Sarah Goff-Dupont

[**GitHub Flow**](https://guides.github.com/introduction/flow/)**(Feature Branch)**

https://miro.medium.com/max/45/1*iHPPa72N11sBI_JSDEGxEA.png?q=20



GitHub Flow

Gitflow is a solid model, albeit not the most simple one. It takes time to learn it, and it takes time to move your codebase from one state to another because of the number of the necessary operations.

With the rise of GitHub, [they shared their own workflow](https://githubflow.github.io/). It simply states “for every new feature create a new branch from *master* and merge back into it”. This way, *hotfix*becomes just another *feature*branch, *develop* branch is omitted and *release* is either non-existent or also a *feature,* depending on the way you prepare for a deployment.

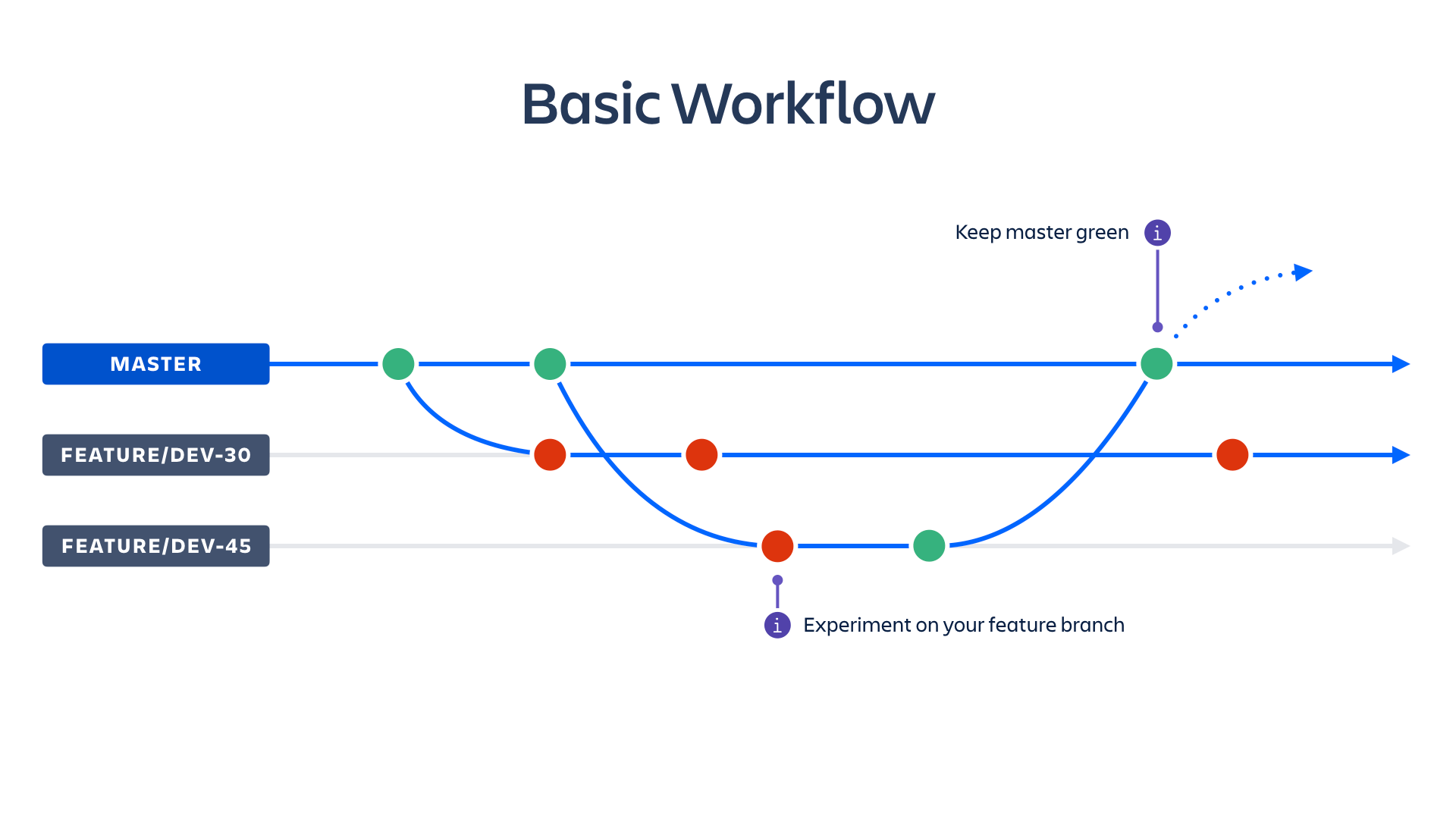
This model might not be as elaborate as the previous one, but it brings one killer feature —**simplicity**.

Another place where GitHub Flow has a big advantage is open-source software. Open-source projects using Gitflow are left with two poor choices —show its visitors and contributors by default either the rarely updated *master* branch or the non-production-quality *develop* branch.

A basic CI/CD setup with this workflow would look like this:

* Run automated tests on all branches.
* Deploy after every merge into *master* or every time you create a designated [*tag*](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-tag)*.*

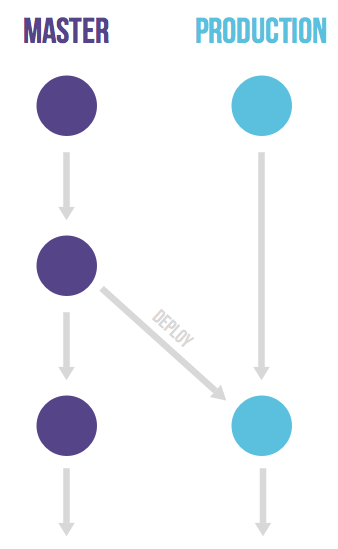
https://miro.medium.com/max/45/1*JMTOcsscPABr3rkd9RjhmA.png?q=20



GitHub Flow + CI/CD, [original article](https://www.atlassian.com/continuous-delivery/principles/workflows-with-feature-branching-and-gitflow) by Sarah Goff-Dupont

[**GitLab Flow**](https://docs.gitlab.com/ee/workflow/gitlab_flow.html)

https://miro.medium.com/max/27/1*DC42a-yL0O7KMto4IYYQ7Q.png?q=20



GitLab Flow — basic version

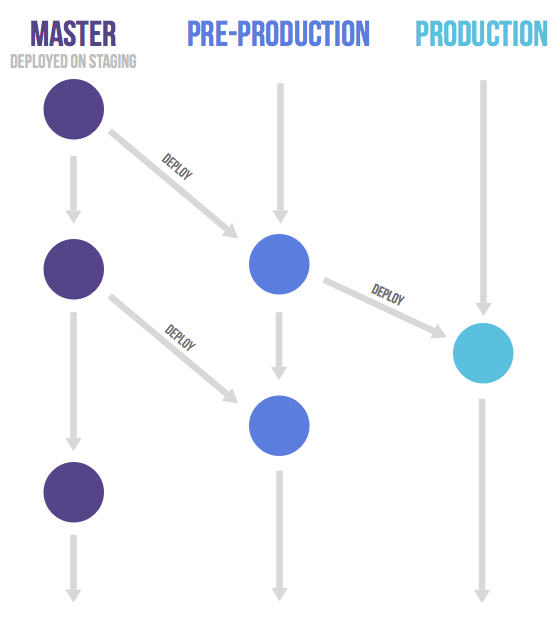
Not long after the emergence of GitHub Flow, it started to seem that every respectable web service for VCS should have a workflow named after it. And so [the GitLab Flow was born](https://about.gitlab.com/2014/09/29/gitlab-flow/).

It builds upon the GitHub Flow and is designed to address the need in more complex deployment logic than just doing it every time the code is merged into the *master*branch.

These requirements can be:

* Inability to deploy at will because of some external conditions — app reviews (App Store), deployment windows.
* Multiple environments (staging, pre-production, production) or multiple distribution channels (alpha, beta, stable).

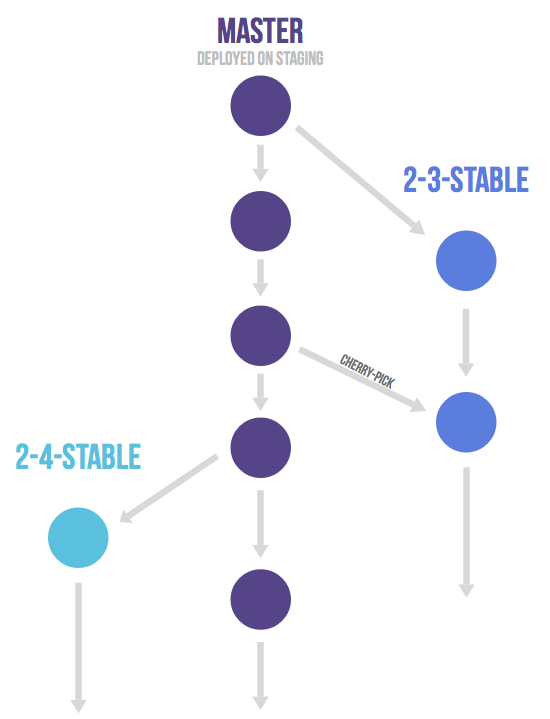
https://miro.medium.com/max/41/1*1JTyIZPRUKSkQAPV3FYCkg.png?q=20



GitLab Flow — multiple environments

* Supporting several application versions at once.

https://miro.medium.com/max/35/1*s8HOQJF4lO-dosEYBPbALg.png?q=20



GitLab Flow — supporting multiple versions

A basic CI/CD setup with this workflow would look like this:

* Run automated tests on all branches.
* Deploy depending on the exact strategy — it may be from the *master*branch, from the *production*branch, or from multiple branches.

**Custom Workflow**

Don’t be afraid to come up with your own bespoke workflow! If you feel that you need something else than any of the models you found online, then go ahead and modify the existing or create a new one from the ground up. No one on the Internet will know your team’s and project’s requirements better than you!

**Devising a Workflow**

Let’s go through considerations when creating a workflow for an application, specifically, an Android one.

**Starting off**

A good place to start is one of the Git models we explored earlier.

Generally, **GitHub Flow** might be a better choice in such cases:

* You expect to release frequently (every day or more in general, every couple of days for Android).
* You have an open-source project.
* You’re not sure about your requirements. It’s always easier to start simple and expand afterwards than to begin with a complex system which will be hard to dismantle later on.
* You just like to [KISS](https://en.wikipedia.org/wiki/KISS_principle) :)

Consider **GitLab Flow**if:

* You have constraints on the time of deployment.
* You have several app flavors (e.g. alpha, beta, stable) that you need to release independently.
* You need to support several app versions simultaneously.

And, finally, think about **Gitflow** when:

* Your releases should be infrequent (weekly or more rare).
* You might often need to wait for some time for release candidates to be merged in *master,*butcontinue working on the new features in the meantime.
* You need to support several app versions simultaneously.

**Optimizing**

We always want to reduce CI builds execution time as much as possible to quickly see feedback and move forward. Sometimes, the reason for this is also the build time limit. For example, if you’re on the free tier. And since Android builds can take obscene amounts of time, this problem becomes even more acute.

There are several ways to spend less time on builds:

* Don’t trigger them at all when it’s not necessary.
* Find tasks that take a lot of time but can be omitted in certain cases.
* Customize tasks for specific situations. Sometimes, we can set up different configurations of the same task that take different amounts of time.

Now for the concrete optimization examples:

* **Run CI builds only for pull requests** as we want to build only meaningful code. Usually, this setting can be enabled in CI settings.
* **Skip CI builds for branches that never contain untested code**. For example, you can skip the *master* branch if you don’t need to deploy every time it changes. Just make sure to rebase any new branch onto the *master* before merging.
* **Cache your Gradle files**. You will save you a lot of time by avoiding downloading dependencies while they don’t change. If you have multiple Gradle build files, look at [this advice from Chris Banes.](https://medium.com/@chrisbanes/circleci-cache-key-over-many-files-c9e07f4d471a)
* [**Run Espresso tests with Robolectric**](https://medium.com/androiddevelopers/write-once-run-everywhere-tests-on-android-88adb2ba20c5) instead of virtual/physical devices, it will be much faster.
* If you still want to run tests on “real” devices, you can set them to execute only before a release. For example, on the *release*and *hotfix* branches in Gitflow.
* **Generate Bundles only before publishing** to Google Play, if you use them. For other cases use assembleRelease and assembleDebug instead of bundleRelease and bundleDebug. You will save some time because generating APKs is faster than Bundles.
* **Don’t use check to run tests.** It will run corresponding tasks for all build types, like lintDebug and lintRelease, testDebugUnitTest and testReleaseUnitTest. You can choose to run these tasks only for one build type and specify them manually. This will run your checks **2x faster**.
* If you use Firebase Performance SDK, only enable it for production builds to save from 20–30 seconds to several minutes. Learn how to do it in [my other article](https://proandroiddev.com/hidden-costs-of-firebase-performance-and-how-to-avoid-them-a54f96bafcb1).

**Other best practices for a workflow**

* Always use pull requests.
* Always write [nice commit messages](https://chris.beams.io/posts/git-commit/).
* Use [Semantic Versioning](https://semver.org/).

**Conclusion**

Let’s outline the gist of what we have learnt.

VCS and CI/CD are integral parts of a modern development process. We’ve discussed why and how they influence each other, basics you need to know to use them, as well as some Android-specific tips to remember. So, keeping all that in mind, before taking on a project, you should always take some time to design a workflow which will reflect the requirements of your product and your team.