# DESIGN A WORKFLOW FOR YOUR TEAM

@hectorsector
@beardofedu

# AGENDA

- Discussion of what makes Git special
- Deep dive into branching and merging with Git
- Review existing branching strategies
- Beyond branching
- Hands-on with integrations
- How to design your own workflow

# BRANCHING IN GIT



You probably came to a workflow discussion at GitHub Universe thinking we would spend a lot of time talking about branching. You're probably right. But why?

# BRANCHING IN GIT

- Cheap



* cheap: creating a branch requires not much more than the creation of a pointer. Repository files and history isn’t cloned for each new branch.

# BRANCHING IN GIT

- Cheap
- Easy



* easy: as simple as `git branch` and no straightforward ways to control that.

# BRANCHING IN GIT

- Cheap
- Easy
- Merging is simple



* merge simply: when you have to bring history together, the merging of two or more branches is relatively simple and, when conflicts occur, can be intelligently resolved.


* disposable: can delete when no longer needed, can rename or create a new one. Branches in Git aren’t restrictive or limiting.

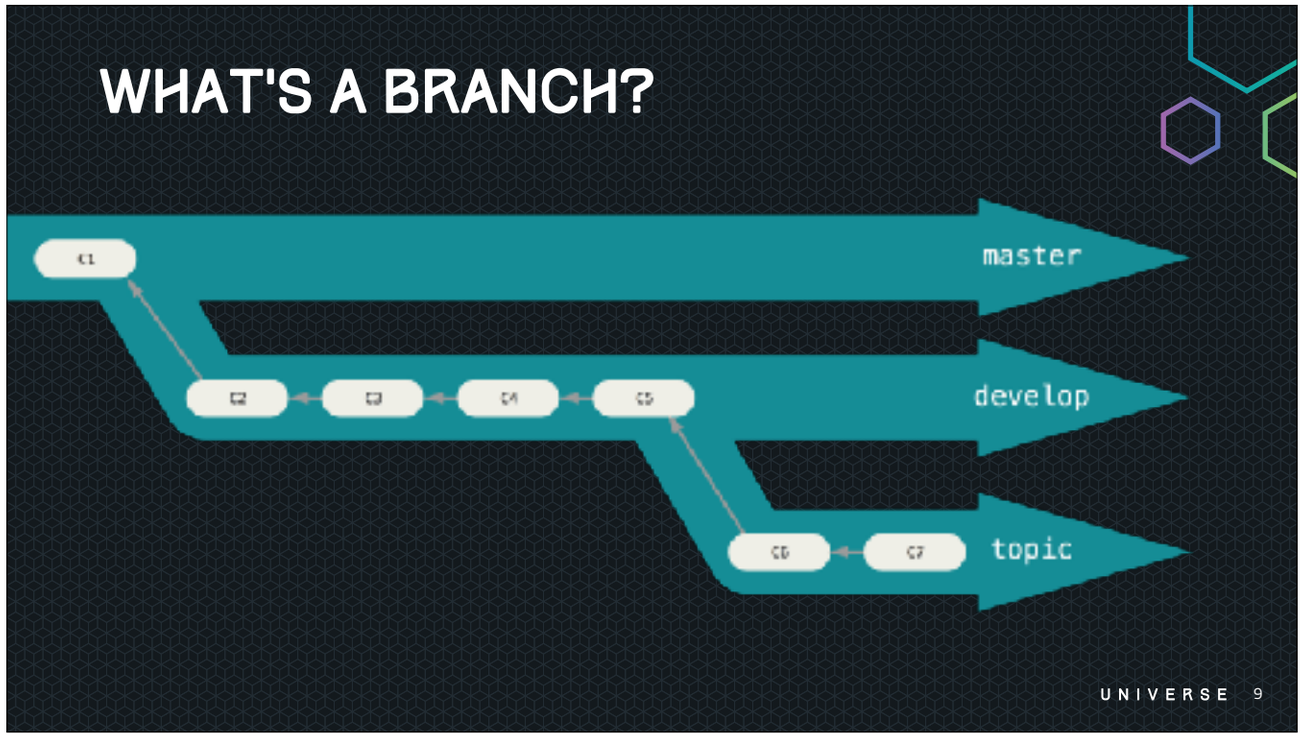
Do not fear branching. Much of the fear I hear about comes from teams that are coming over from a legacy version control system. In ClearCase, for example, a single file may be stored as multiple copies if it is part of the history of different branches. This happens enough in other VCS that we’ve designed our workflows around the limitations.

# BRANCHING IN GIT

- Why the push back?

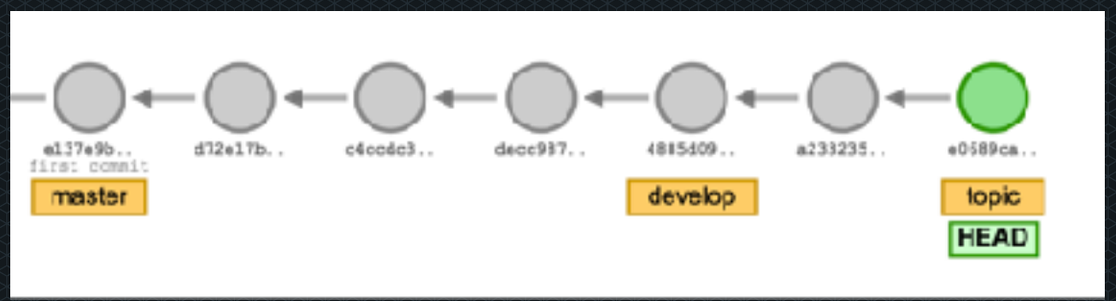


Do not fear branching. Much of the fear I hear about comes from teams that are coming over from a legacy version control system. In ClearCase, for example, a single file may be stored as multiple copies if it is part of the history of different branches. This happens enough in other VCS that we've designed our workflows around the limitations.




You’re probably familiar with this representation of a branch:

# WHAT'S A BRANCH?

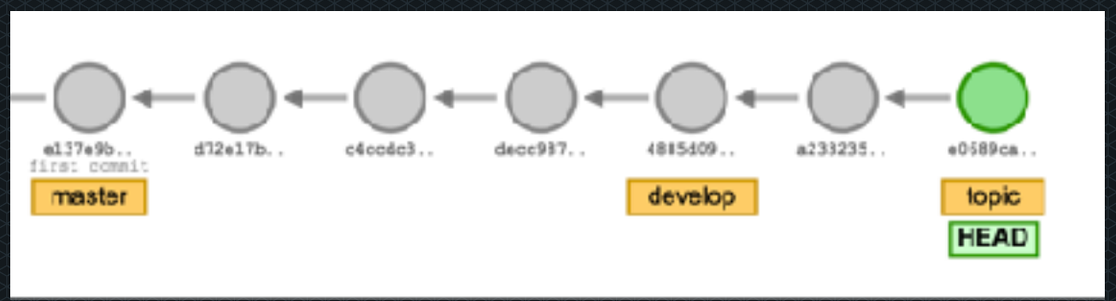





But this isn’t an accurate representation of how Git implements branches. It’s a device we use so we can project the branching models we’ve already got mental maps for, and so that we can have intelligent discussions around branching.

This is a better illustration of how Git implements branching. This is the history of an entire repository. Including all its branches.

# WHAT'S A BRANCH?

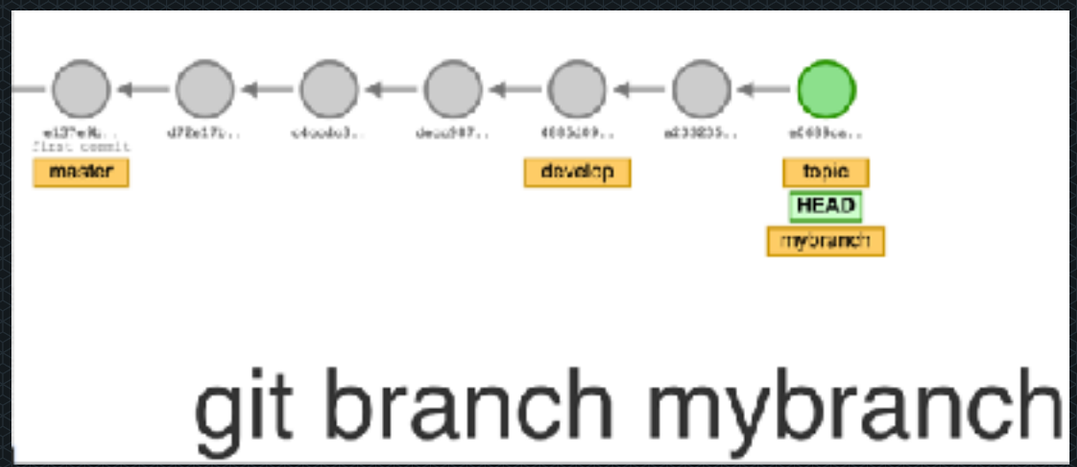





The circles are commits, and in Git commits are complete snapshots of the codebase. Those commits have parents — that's what the arrows represent — and branches are nothing more than a pointer, or a label, anchoring down a part of that commit history.

# GIT-SCHOOL.GITHUB.IO/ VISUALIZING-GIT/

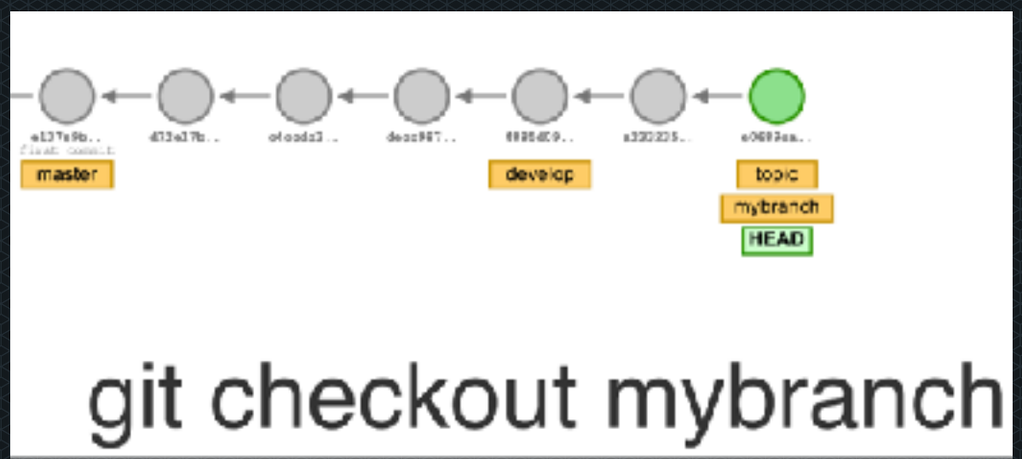
# CREATING A NEW BRANCH






When you create a new branch, this new label emerges .Notice the repository history stays pretty much exactly the same.

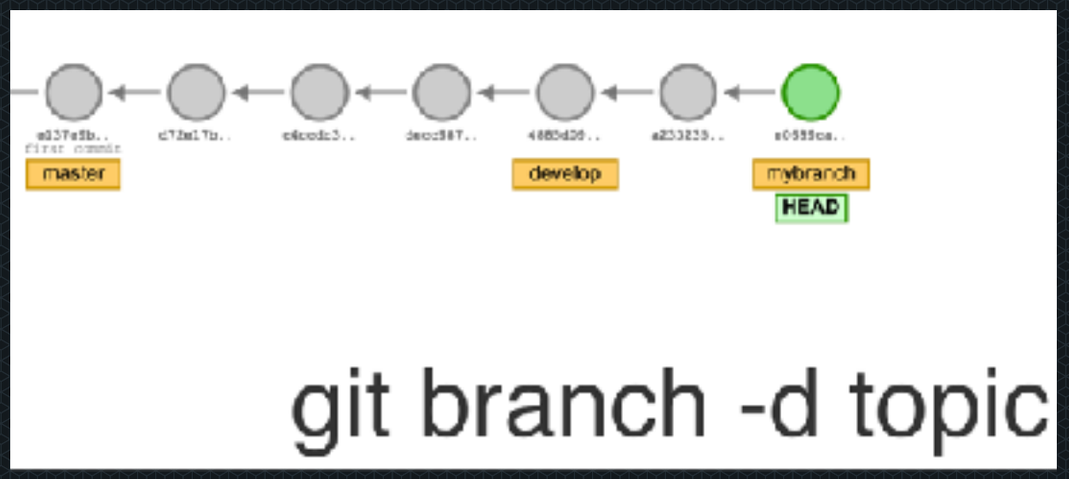
# WORKING ON A BRANCH






When you’re checked out to a branch, and you create a new commit, the branch pointer moves forward with you.

# WORKING ON A BRANCH






When you delete a branch, that pointer gets deleted. If the history on that branch is anchored down by some other branch, it stays. Otherwise, it’s garbage collected by GIt.

# ARE ALL BRANCHES CREATED EQUAL?

- Mostly yes.



Mostly: yes. To Git, anyway. But we pesky humans like to overcomplicate things. So we use terminology that isn’t necessarily built into Git.

# ARE ALL BRANCHES CREATED EQUAL?

- Mostly yes.

- Topic
- Feature
- Short-lived
- Long-running



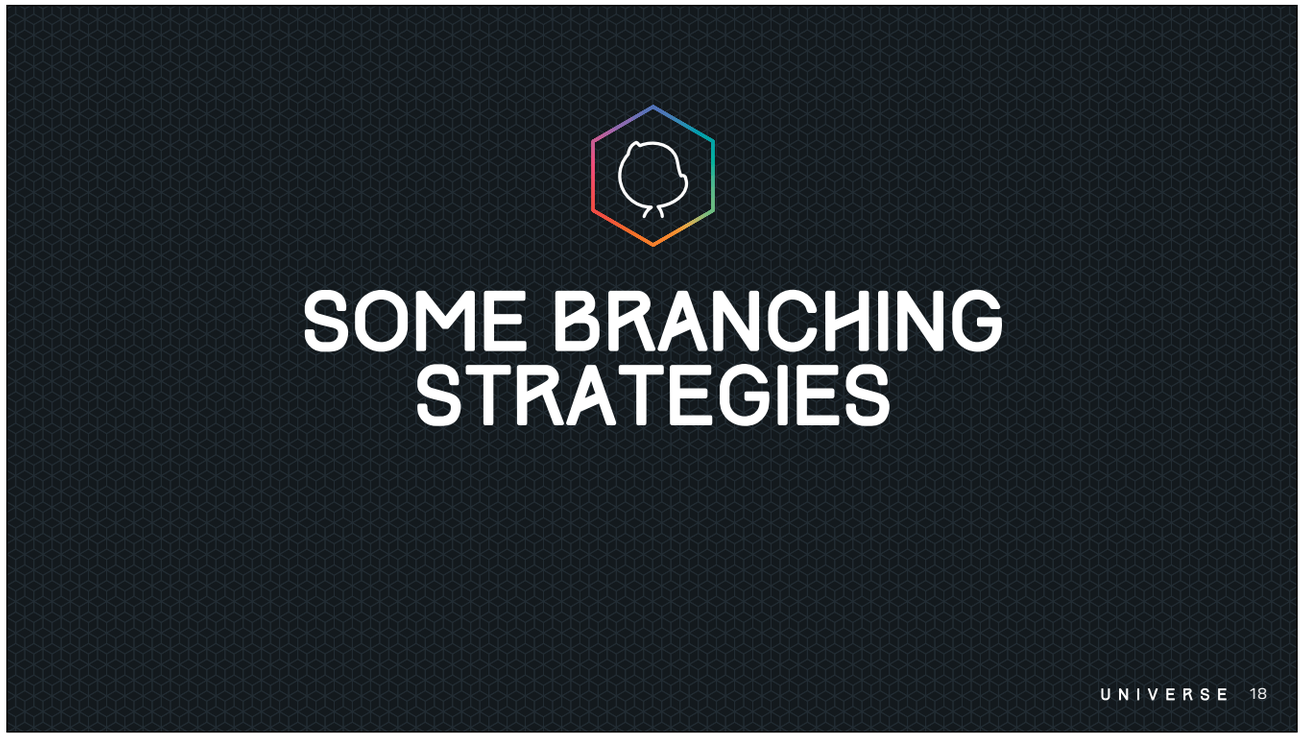
Words like:

Topic
Feature
Short-lived
Long-running

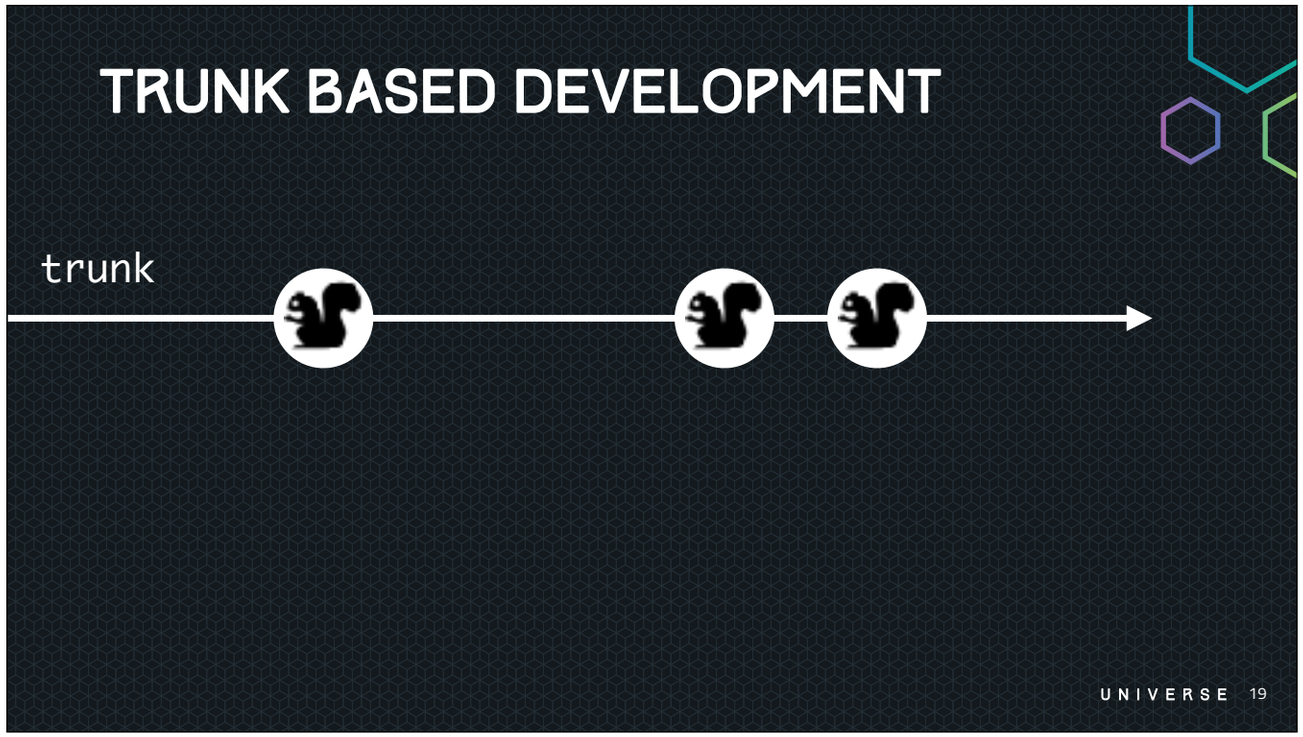
So let’s dive into some of these. And let’s be clear: Git doesn’t really care much which of these names you employ. Git defaults to a `master` branch, but in reality you can call it whatever you like. On GitHub, even, you can indicate your preferred default branch.

Generally speaking this default branch, let’s go with `master` for now, is kept around forever. Let’s call this a long running branch.

Some teams choose to create other long running branches alongside `master` to have greater flexibility. But, again, this behavior isn’t necessarily natural to Git.



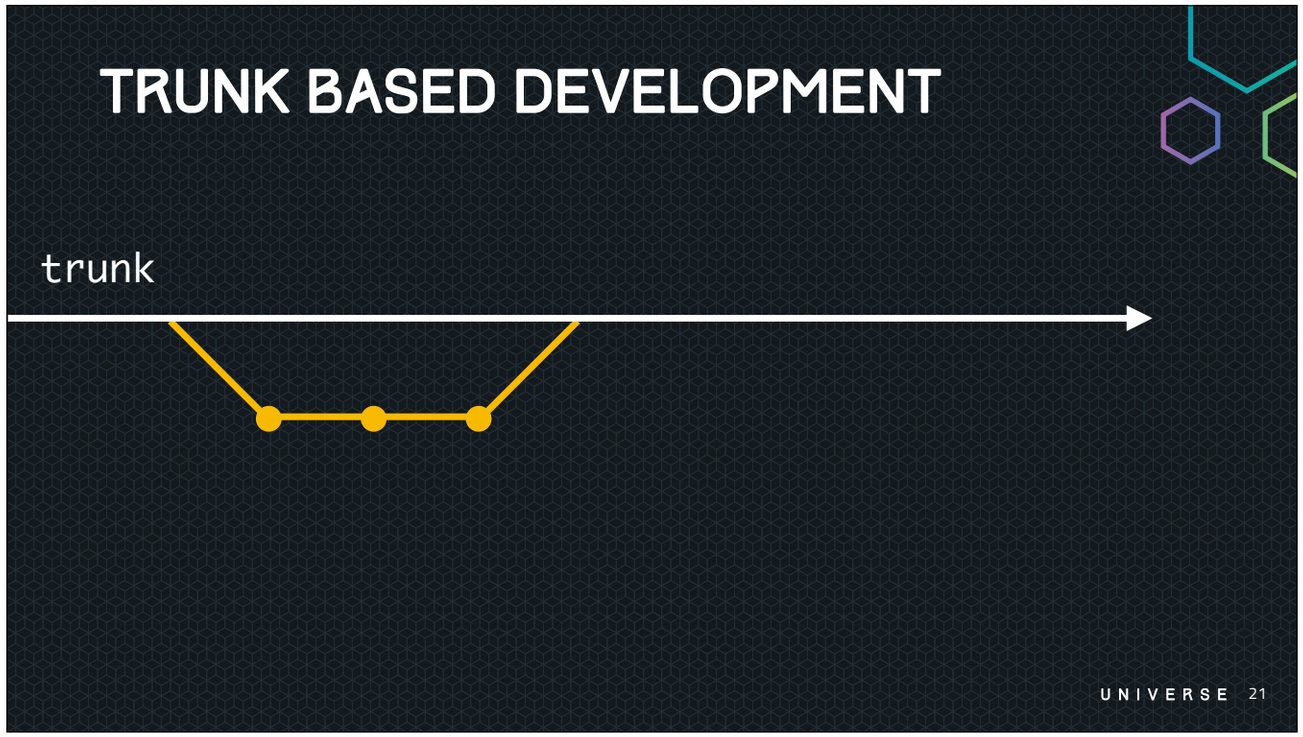

Now that we’re pretty savvy, let’s look at what other folks are doing with their branching strategies and see if we can make some sense of why a team might choose each strategy.




In trunk based development, all work centers around the trunk (in our case, `master`, but remember Git doesn’t care). The trunk is always considered deployment ready.

In pure trunk based development we don’t use branches.


All work is based on the trunk.



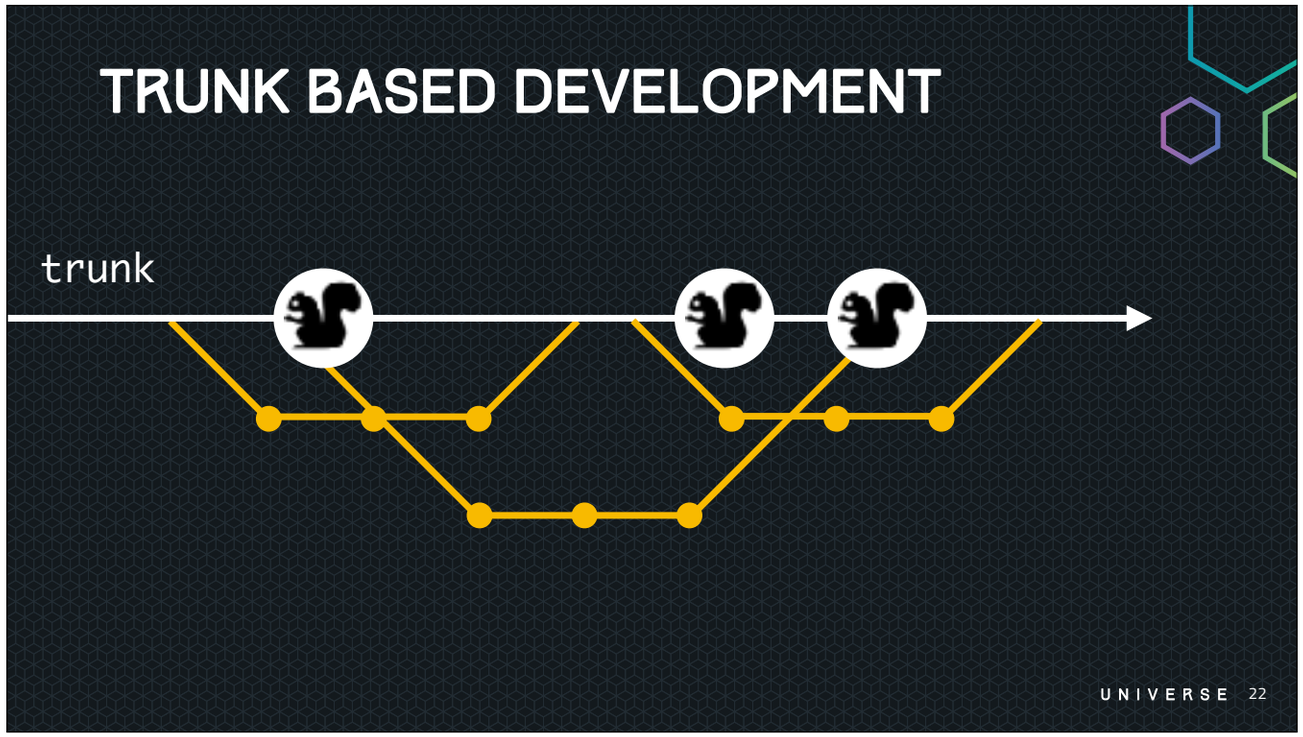

And is merged back into the trunk.

There are some benefits to this workflow like:
* one source of truth, the trunk
* easy to understand, harder to follow through with
* deploy directly from the trunk

All branches that we create from the trunk are temporary. They’re short-lived.

But to implement trunk based development, we must be absolutely certain that anything that makes it into `master` has been fully vetted. This means that our discussion of branching needs to encompass more than just branching. Trunk based development typically requires that each commit be built and tested. This makes sense, if the commit will make it into `master`, which is deployment ready,

So our discussion of workflow has to extend past branching. In this case, we need CI to test each commit.



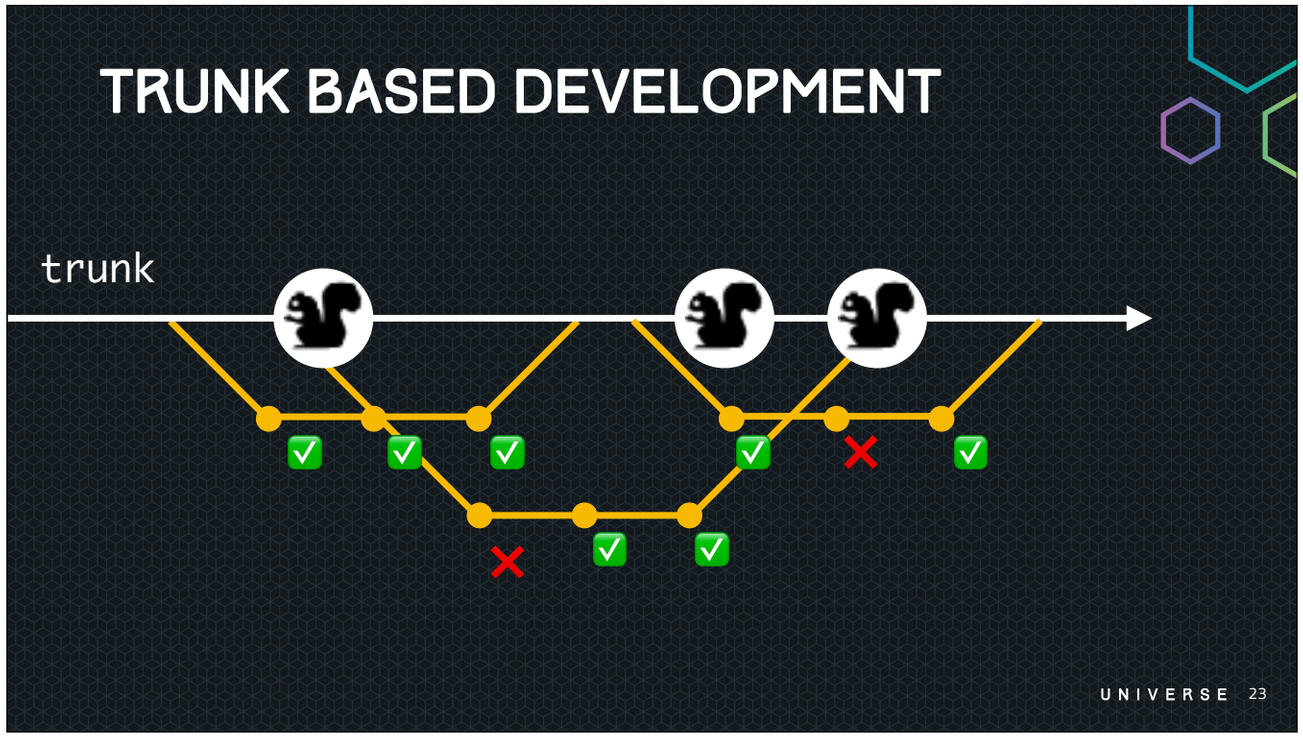

And is merged back into the trunk.

There are some benefits to this workflow like:
* one source of truth, the trunk
* easy to understand, harder to follow through with
* deploy directly from the trunk

All branches that we create from the trunk are temporary. They’re short-lived.

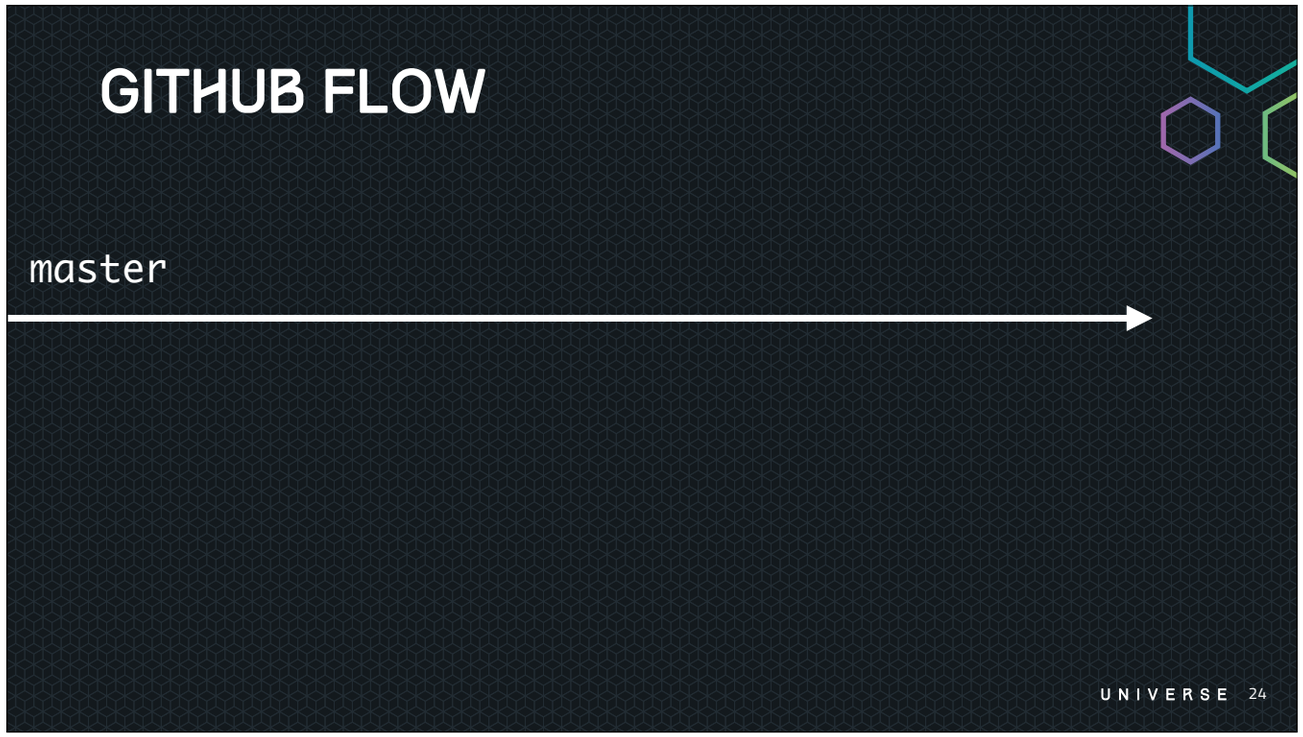
But to implement trunk based development, we must be absolutely certain that anything that makes it into `master` has been fully vetted. This means that our discussion of branching needs to encompass more than just branching. Trunk based development typically requires that each commit be built and tested. This makes sense, if the commit will make it into `master`, which is deployment ready,

So our discussion of workflow has to extend past branching. In this case, we need CI to test each commit.

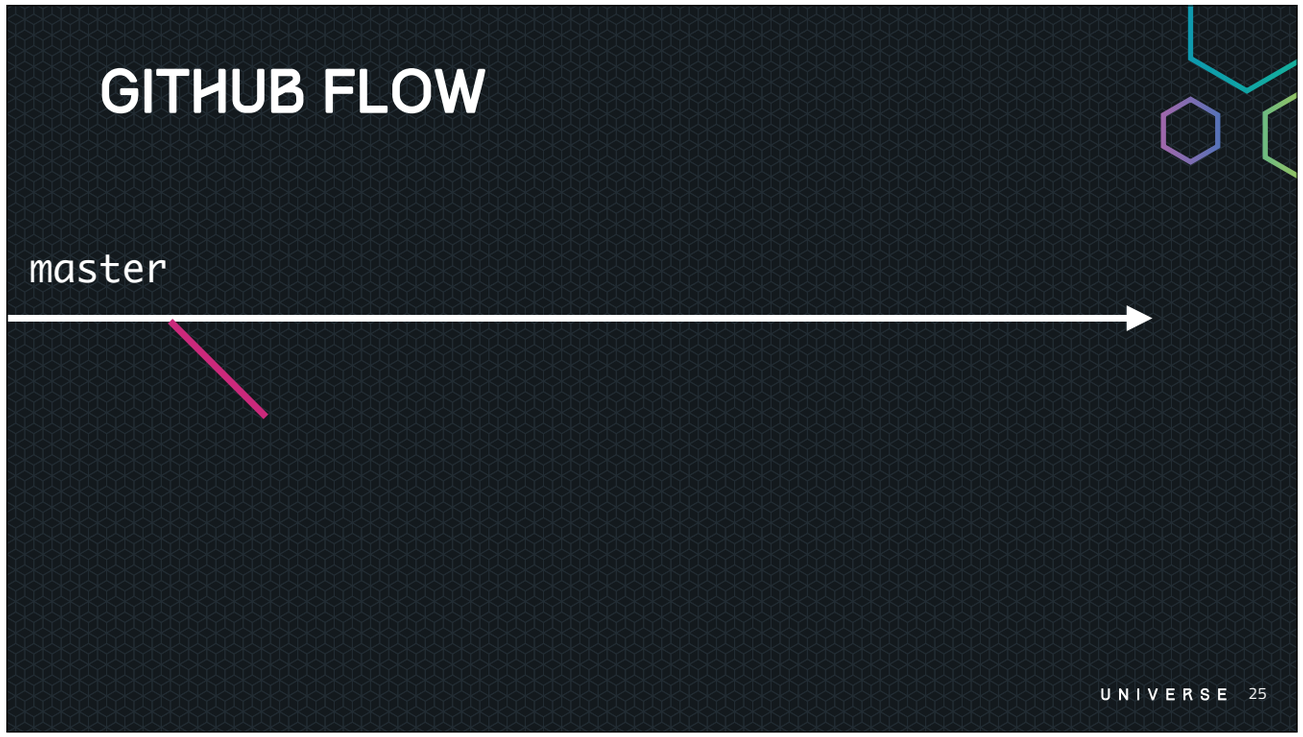



But to implement trunk based development, we must be absolutely certain that anything that makes it into `master` has been fully vetted. This means that our discussion of branching needs to encompass more than just branching. Trunk based development typically requires that each commit be built and tested. This makes sense, if the commit will make it into `master`, which is deployment ready,

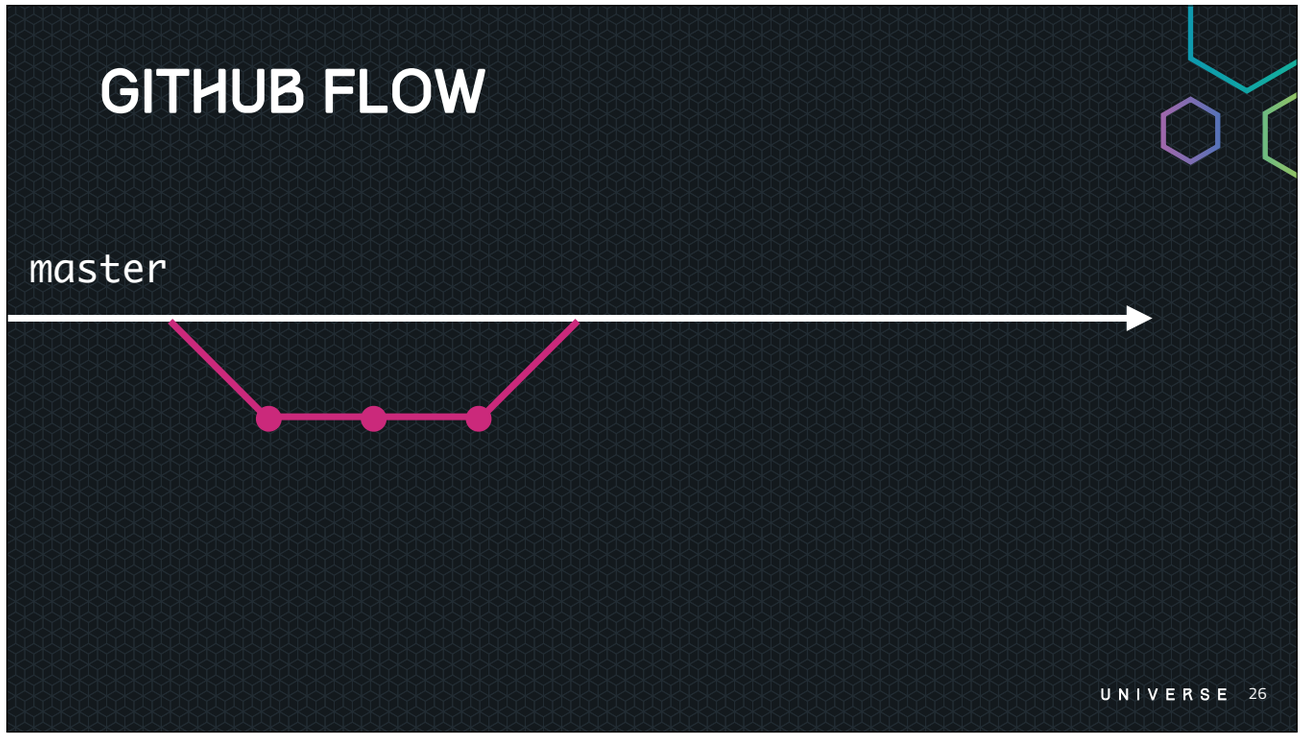
So our discussion of workflow has to extend past branching. In this case, we need CI to test each commit.



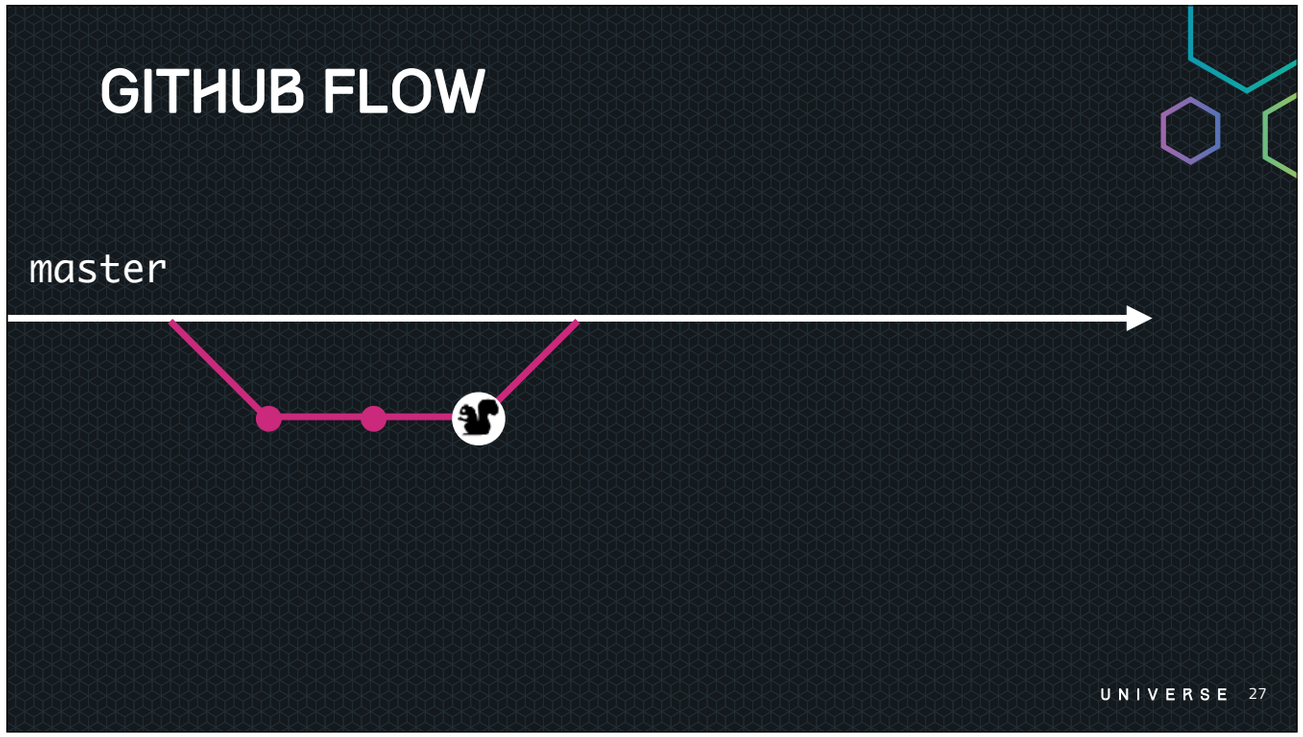

GitHub Flow follows a similar pattern to trunk based development. We consider `master` to be deployment ready



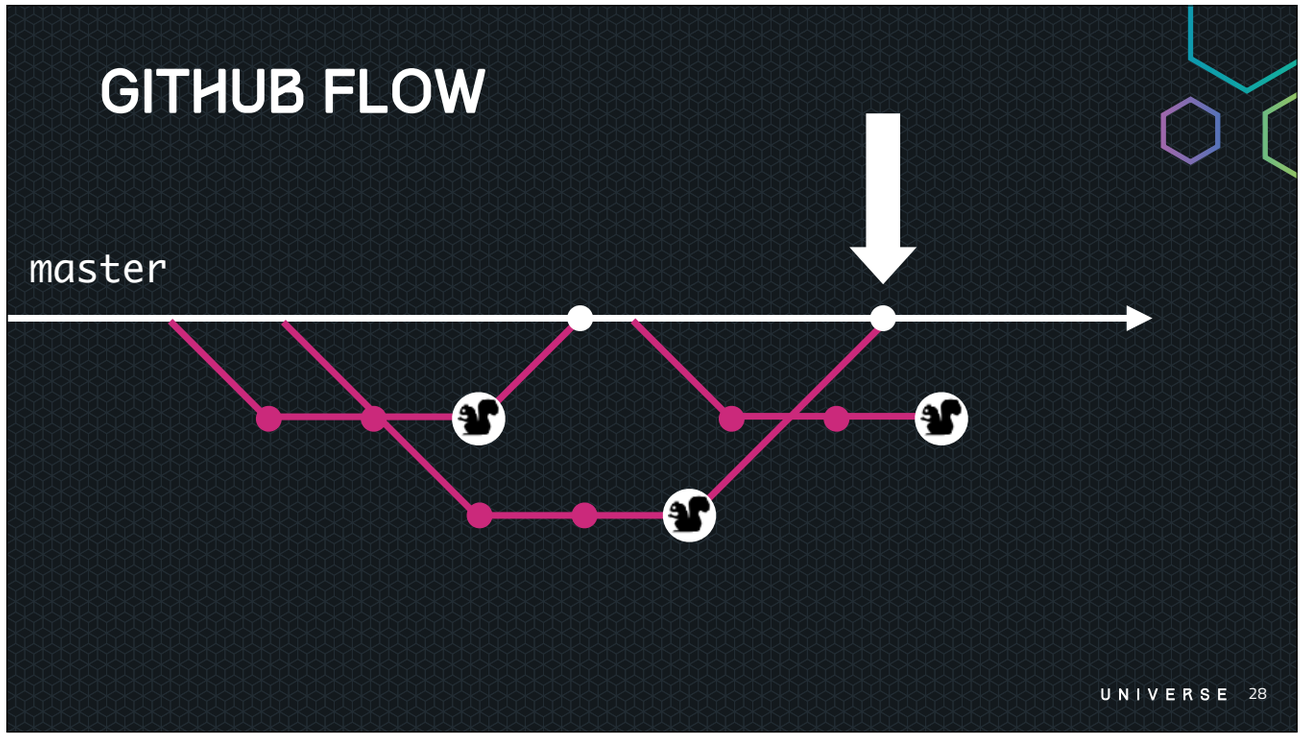

We base all our work on it




and merge back into it. All branches are short lived, and are disposed off when merged into `master`.

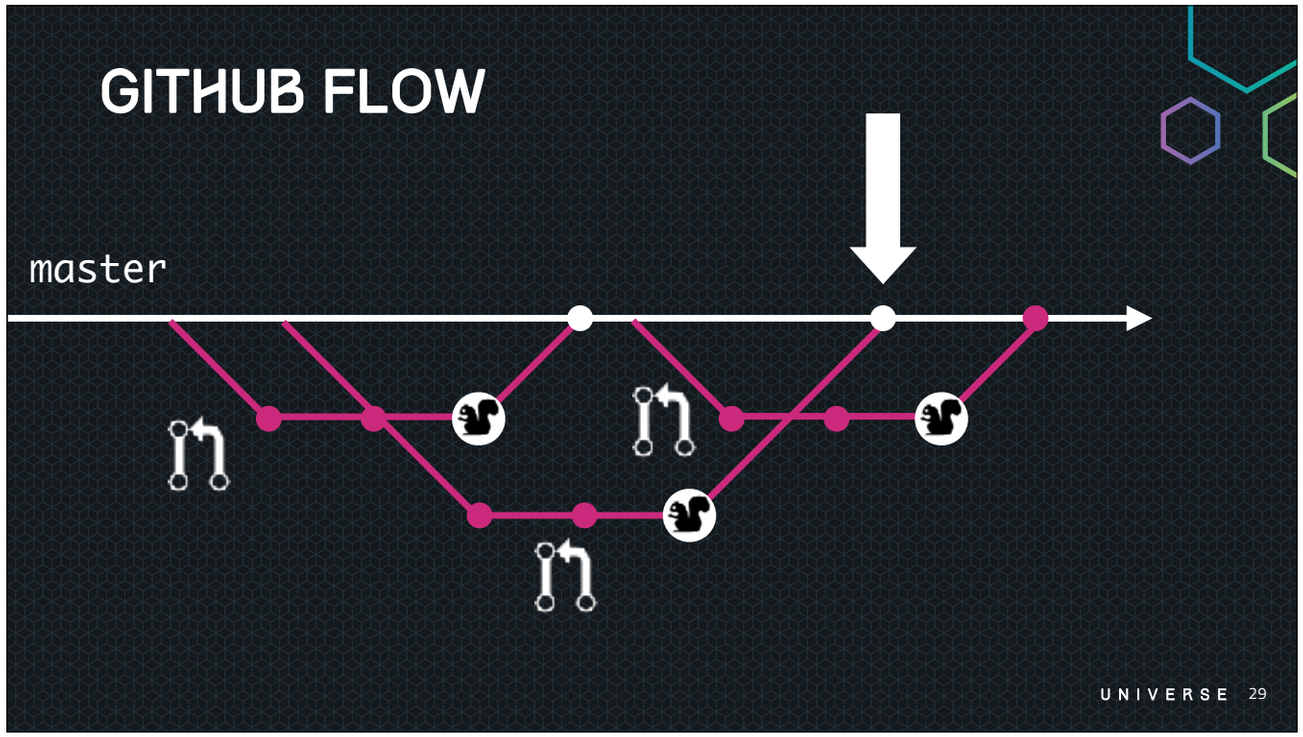



But in GitHub Flow, we can deploy right from branches.




The benefit here is that if something we deployed on the branch bugs out, we can always deploy the most recent commit on `master` to undo our mistake.

But, again, branching alone does not define this work flow. The Pull Request, for example, is used for every branch that goes through code review at GitHub. So, to see a more complete picture, we need to include the PR.

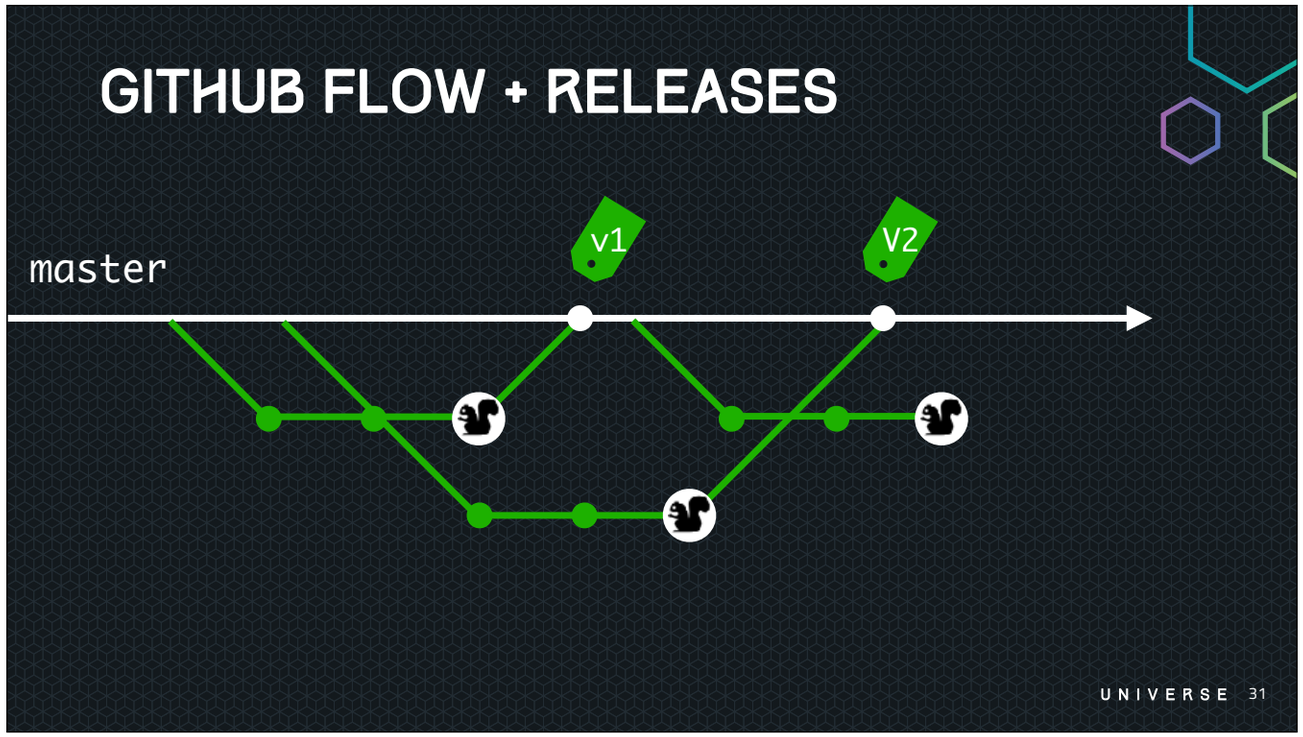



But, again, branching alone does not define this work flow. The Pull Request, for example, is used for every branch that goes through code review at GitHub. So, to see a more complete picture, we need to include the PR.

# GITHUB FLOW + RELEASES



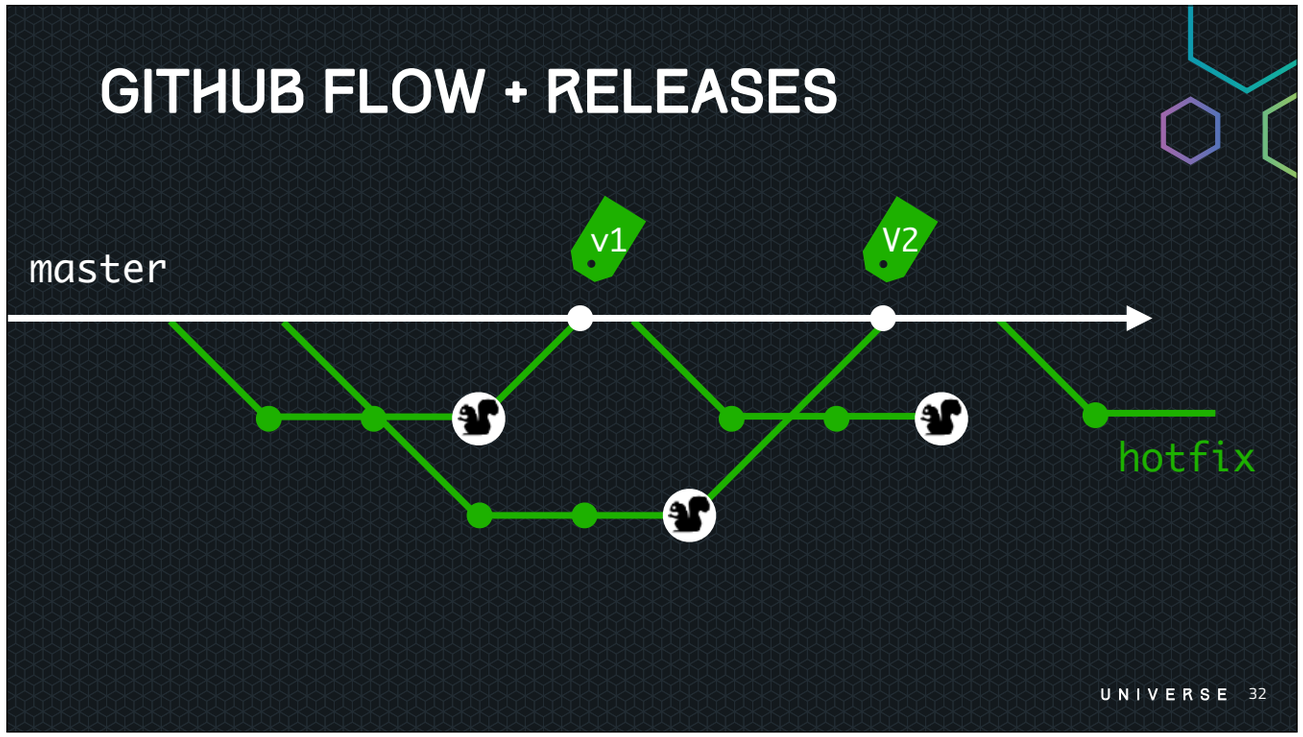
What if I was mostly happy with GitHub Flow, but that I wanted the ability to cut some releases?




Git has a built-in way of doing this, with tags. A tag works much like a branch, in that it anchors down a portion of the history by attaching itself to a commit. But a branch and a tag differ in that a tag is stationary, it marks the commit and doesn’t update if new commits are created. A branch, on the other hand, follows any new commits as long as you’re checked out to it.

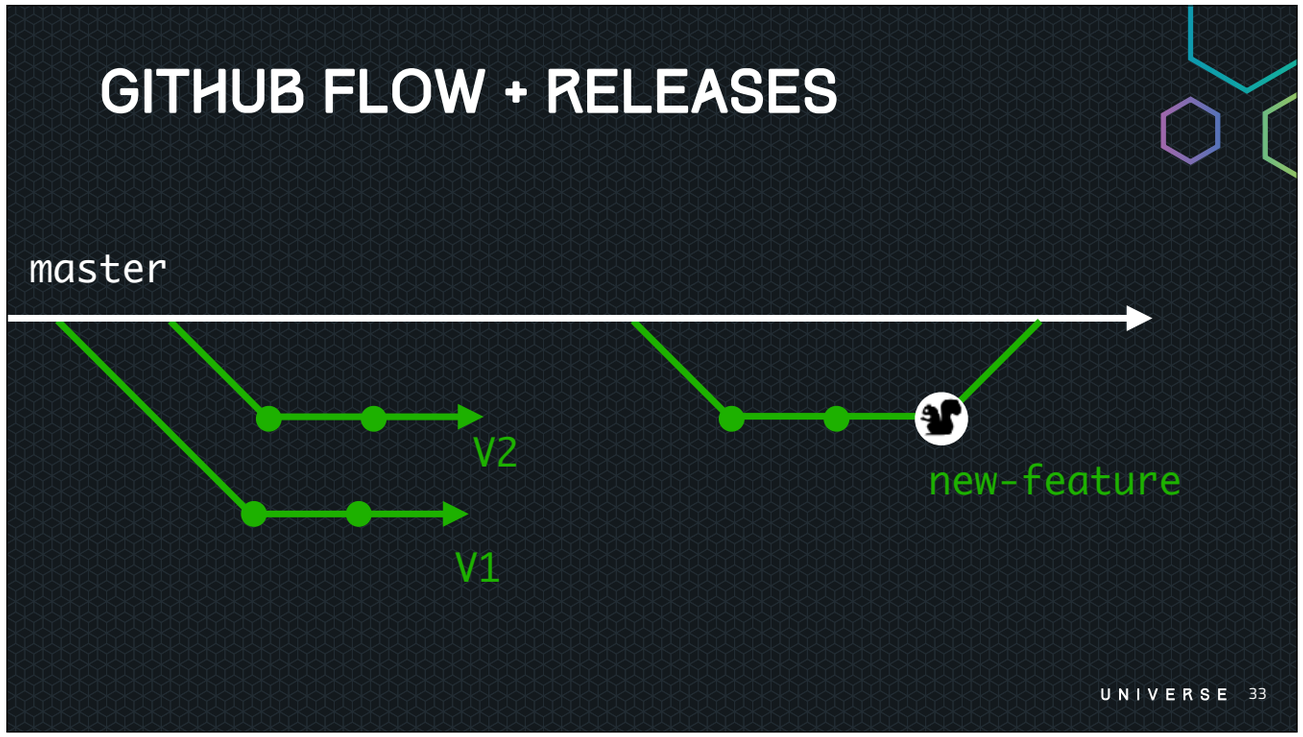
But what happens if once we cut a release, we then need to be able to introduce a hot fix that only applies to that release. For scenarios like this, we may need to introduce some new long-running branches.

But when should releases be created? Who is allowed to merge into one of the releases? And who is allowed to merge into `master`? As you’re beginning to guess by now, branching once again doesn’t tell the whole story. In this case, we would benefit from protected branches with CODEOWNERS, a GitHub feature that allows us to indicate a branch as important, and require reviews from specific individuals.

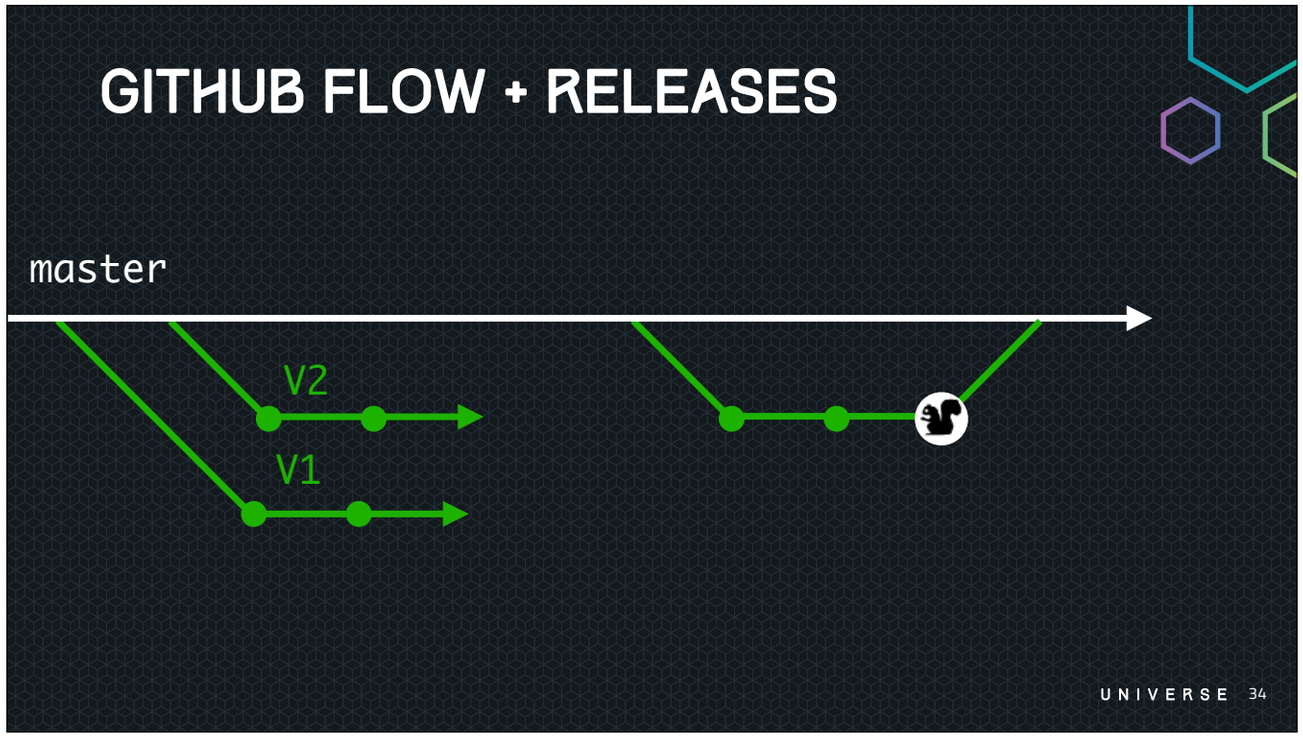



But what happens if once we cut a release, we then need to be able to introduce a hot fix that only applies to that release. For scenarios like this, we may need to introduce some new long-running branches.

But when should releases be created? Who is allowed to merge into one of the releases? And who is allowed to merge into `master`? As you're beginning to guess by now, branching once again doesn't tell the whole story. In this case, we would benefit from protected branches with CODEOWNERS, a GitHub feature that allows us to indicate a branch as important, and require reviews from specific individuals.

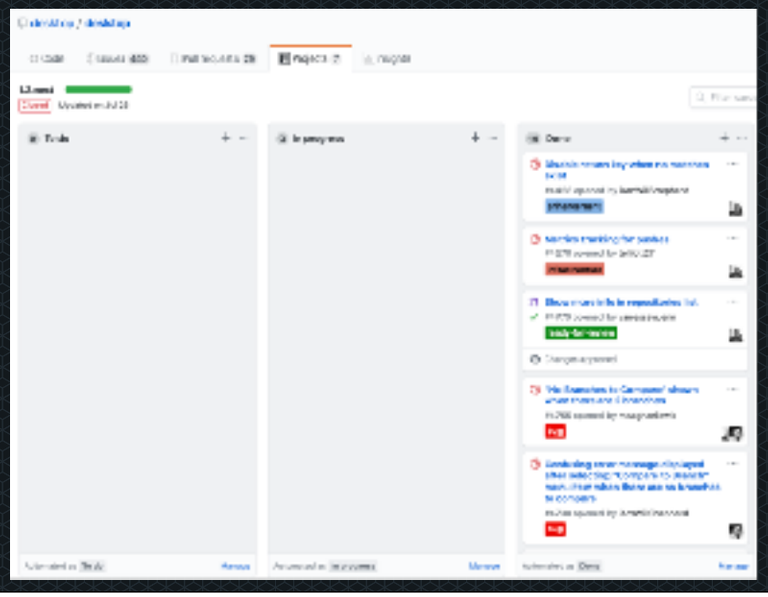



But what happens if we need some time to craft what a particular release looks like, without slowing down development on `master`? We need to be able to satisfy both of these use cases in one branching strategy. For scenarios like this, we may need to introduce some new long-running branches.



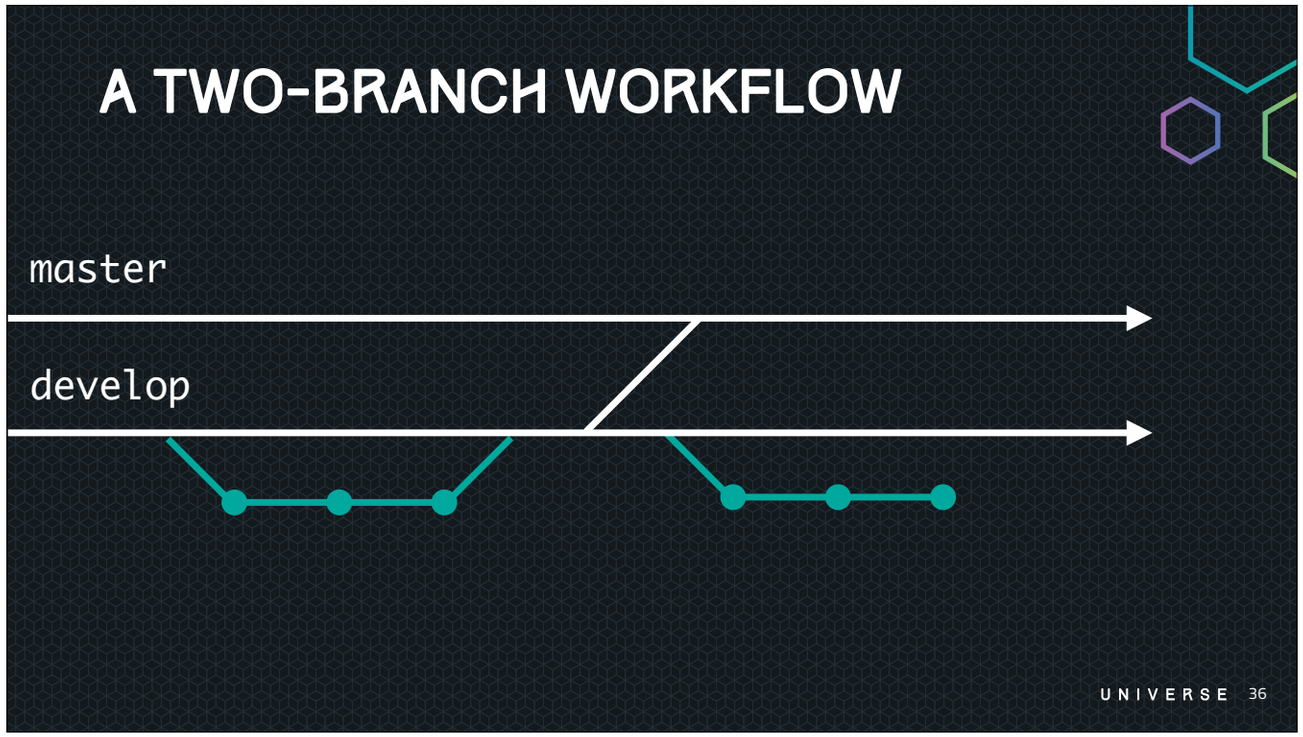

But when should releases be created? How are the features for a given release determined? As you’re beginning to guess by now, branching once again doesn’t tell the whole story. In this case, we would benefit from some project management, either using

# GITHUB FLOW + RELEASES

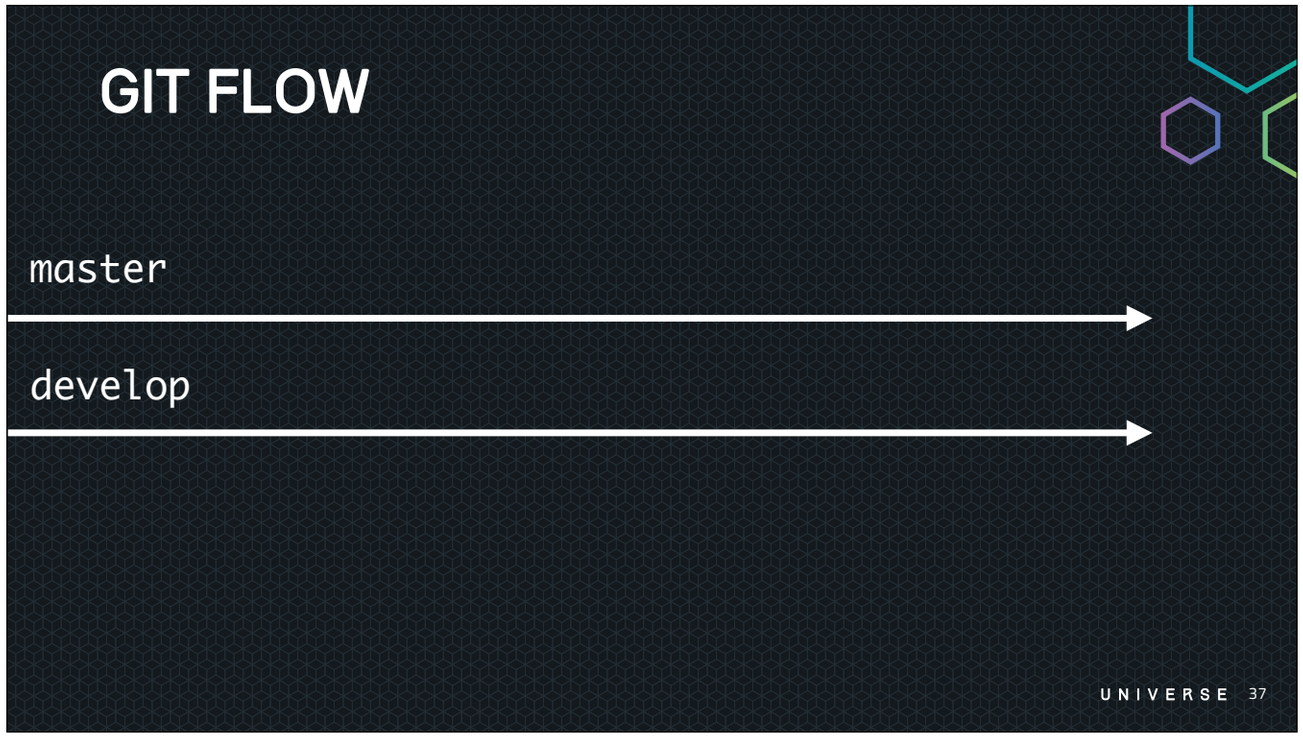




But when should releases be created? How are the features for a given release determined? As you’re beginning to guess by now, branching once again doesn’t tell the whole story. In this case, we would benefit from some project management, either using

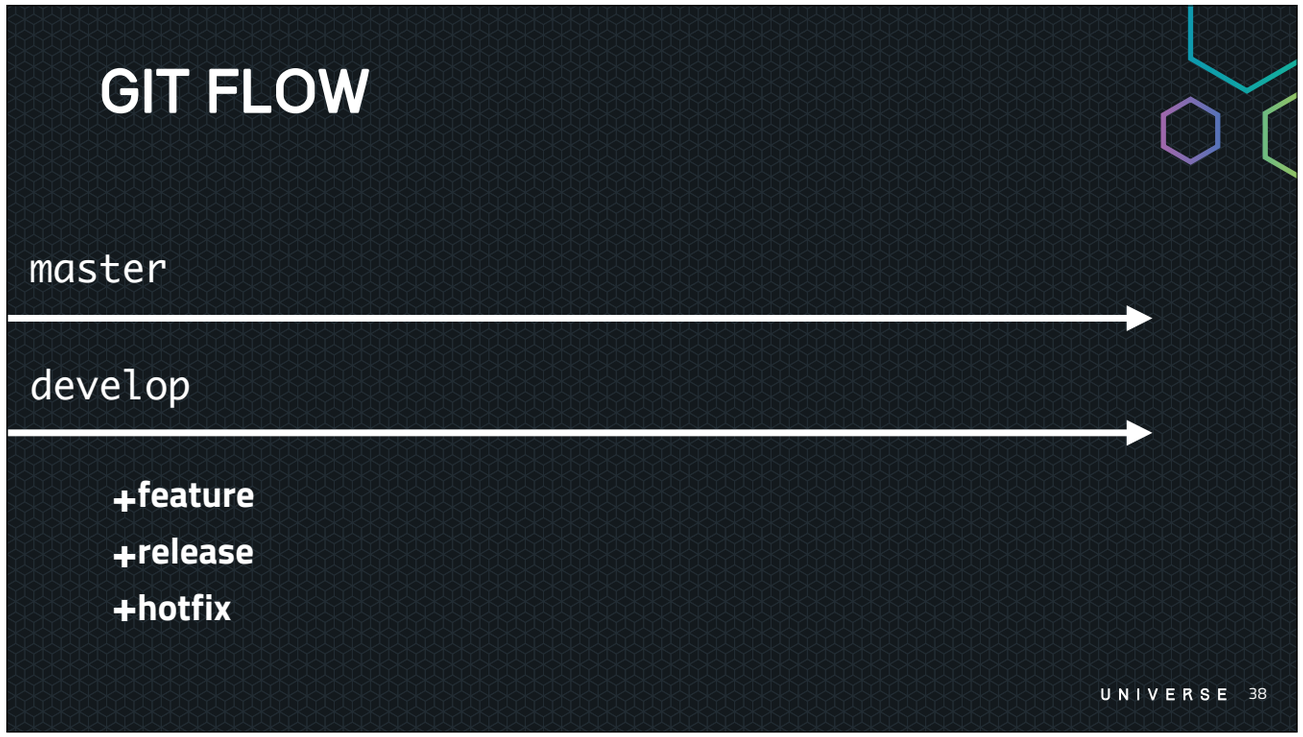



We’ve seen teams that want to control code that makes it to the deployment-ready `master` branch, but want developers to feel empowered to merge multiple times a day. A long running `master` and `develop` branch may solve this problem. Developers can feel empowered to continuously merge into `develop`, and some extra attention can be paid when the work from `develop` should make it into `master` for deployment.

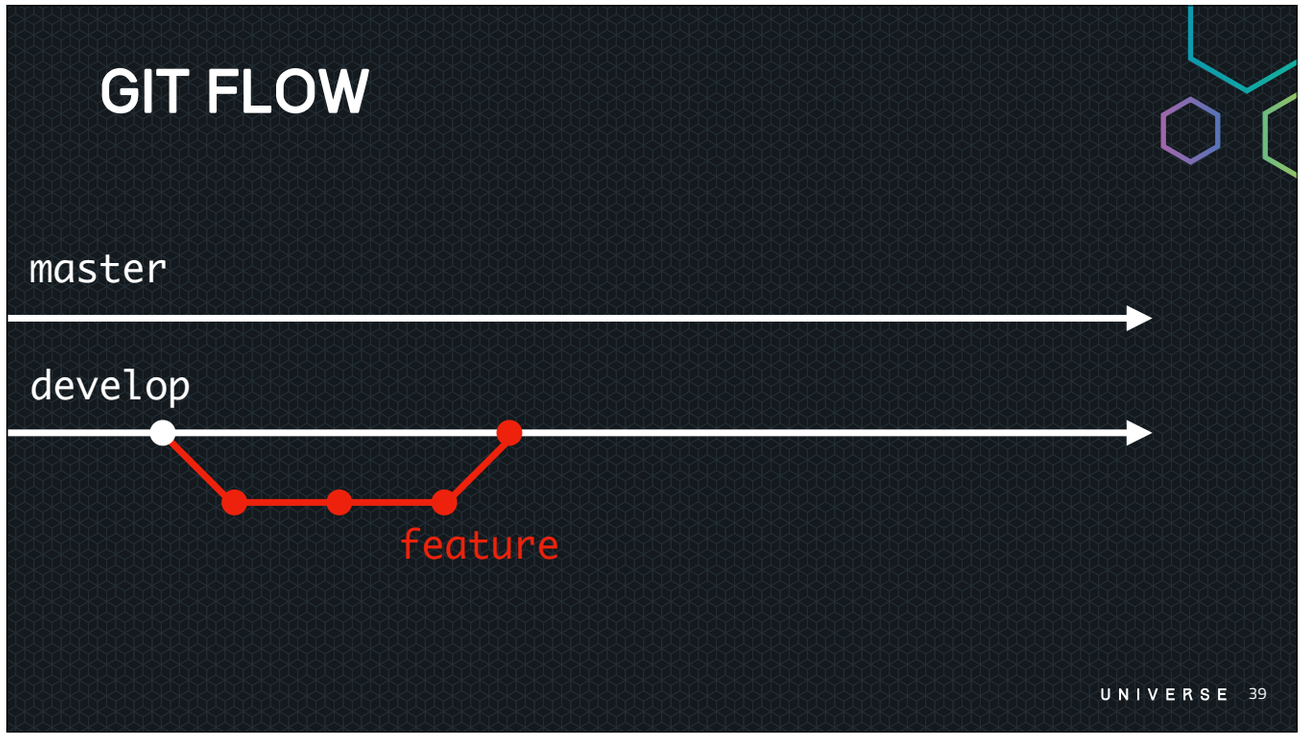



When there’s a lot of complexity in the review mechanism, and when deployment can be tricky, many teams resort to Git Flow.

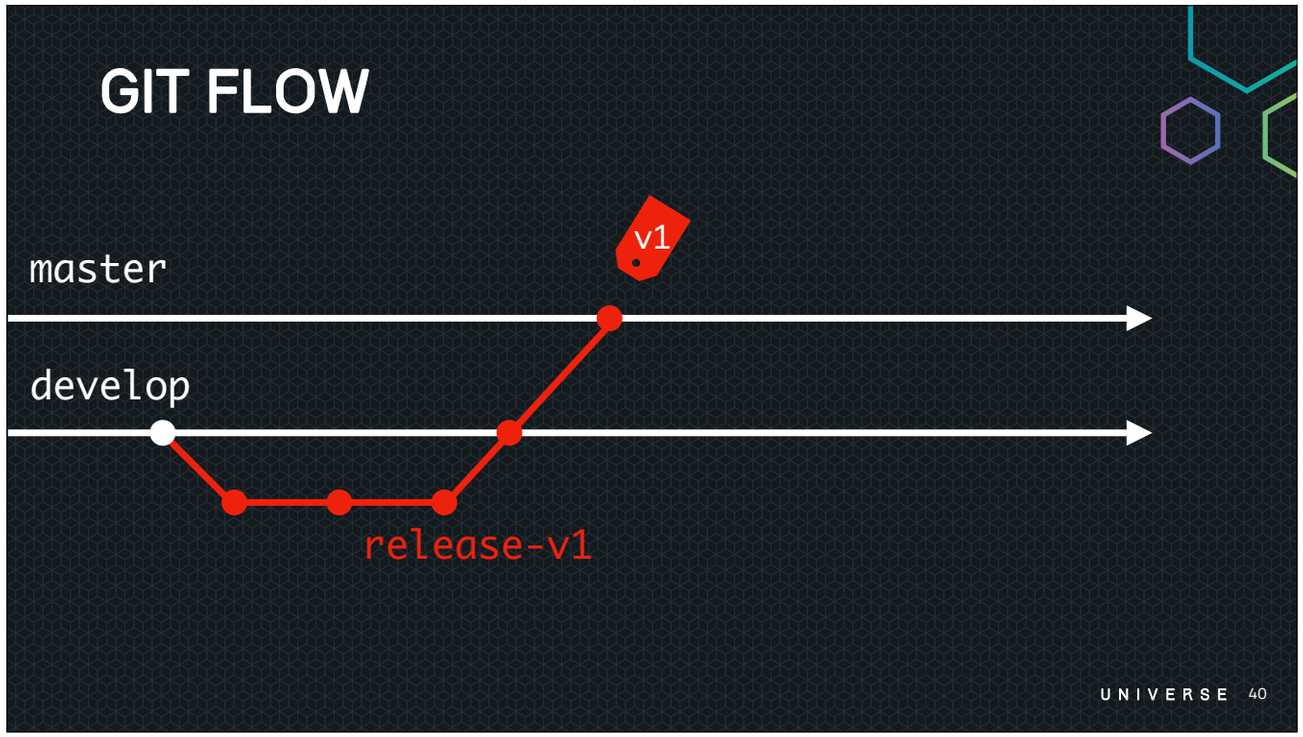
Git Flow is similar to our two-branch strategy and is heavy on long running branches. The branches `master` and `develop`  serve much of the same purpose as in the two branch workflow we just reviewed.



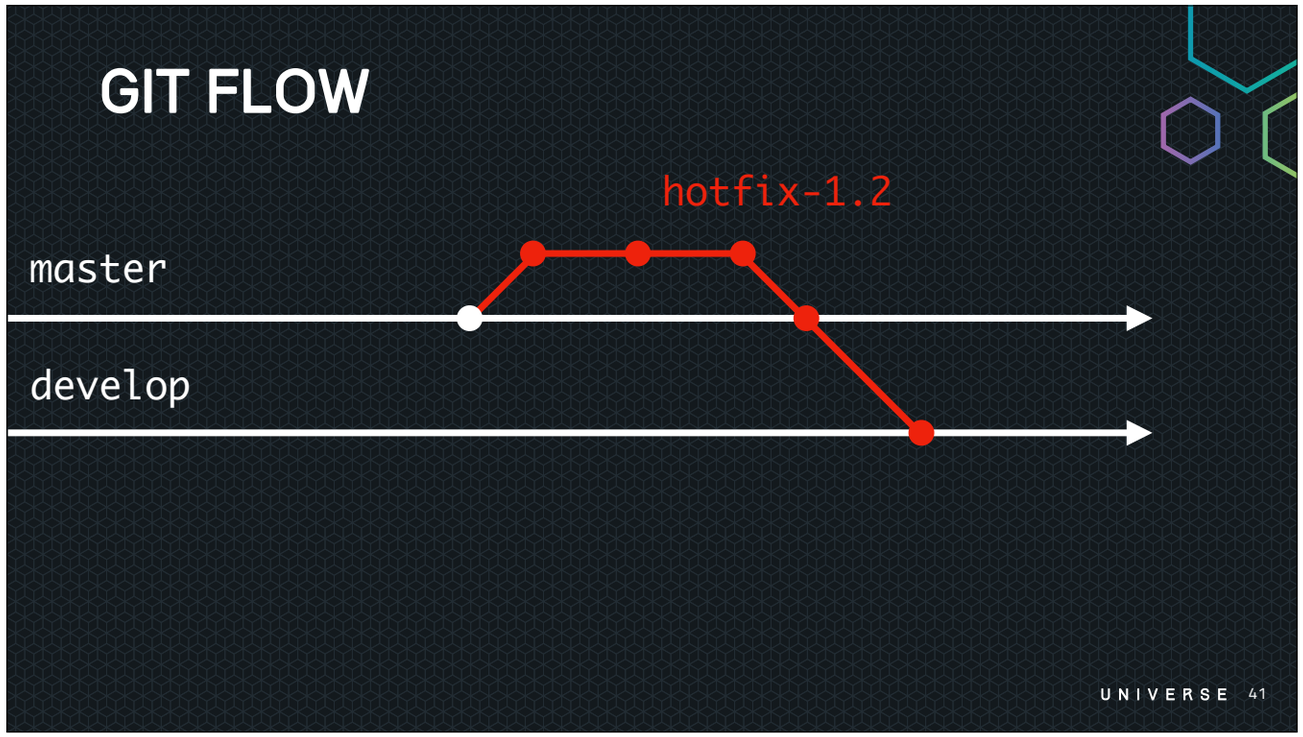

Git Flow also employs some supporting short lived branches: feature, release, and hotfix.




Feature branches are used for, well, feature development. They branch off, and merge back into develop. The name of a feature branch is up to the developer.




Release branches are used for preparing a release. For this reason, they branch off of `develop`, where the most recent work has been merged. Some minor work may need to be completed on the release branch so that it is actually ready for deployment, and at the time that it is, will be merged into `master` where, by definition, all releases are reflected. Release branches must start with the word `release`.  Once a release branch is merged into `master`, a Git tag is created to identify and locate that point in time. On GitHub, these tags can also be used to attach executables or release notes.




A hot fix branch is used when bugs are found in production. Therefore, they are based off of `master`, since that's where the bug would've been found. Once the hot fix is finished, that work gets merged into  `develop`  so that any active development doesn't also contain the bug, and `master`, so we create a new release with the bug fix. Alternatively, if there's a release currently being drafted, we could merge it into that release branch as well.  Hot fix branches start with the word `hotfix`.

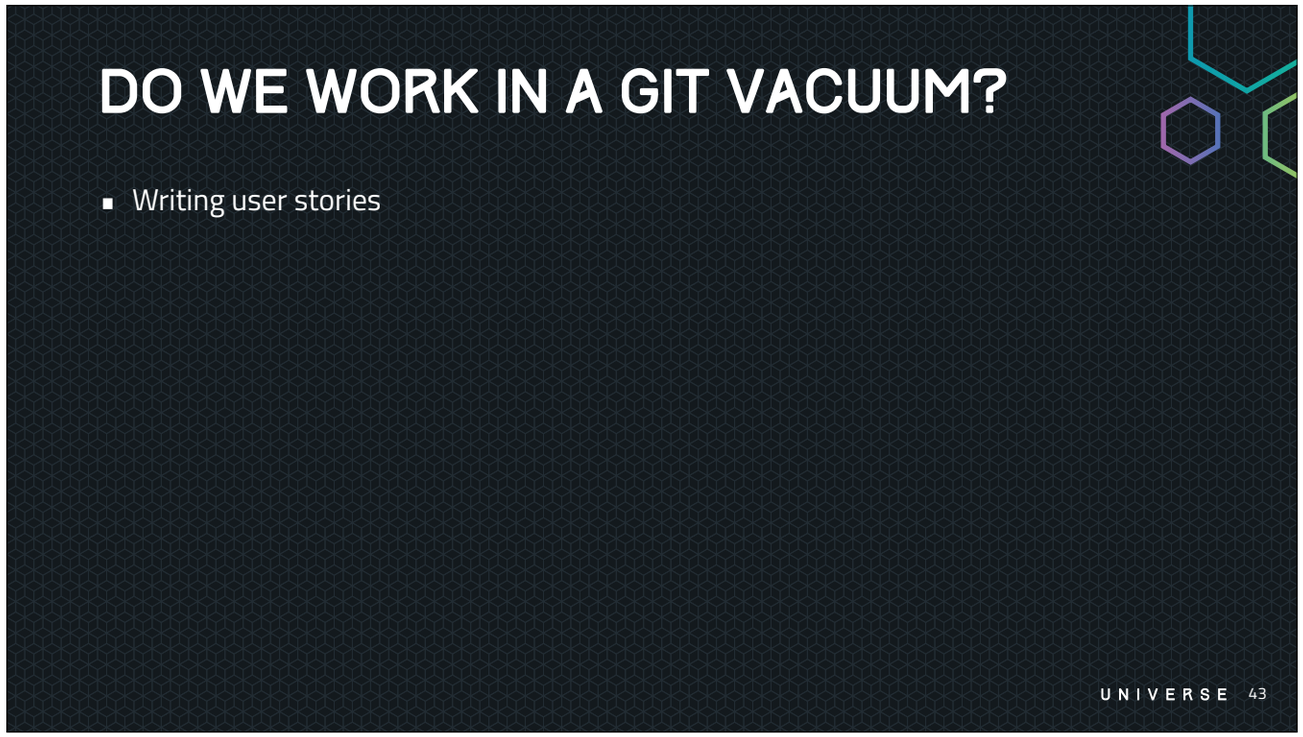
Git Flow requires a lot of history synchronization, increasing the potential for merge conflicts and branches to be out of date with one another.

**Use a single branching picture, add to it**


Is git the only thing in our toolchain? No. From writing user stories to scheduling tasks, to continuous integration, to code coverage, Git and branching are only part of the story. GitHub has this amazing capability of plugging into pretty much any one of these tools. So our conversation about workflow has to be inclusive of those other dimensions of our workflow.

Why wouldn’t we? We’d be cheating ourselves if we kept them out and, remember our goal, is to do our live’s best work. How can we do that if we have to continuously context switch?

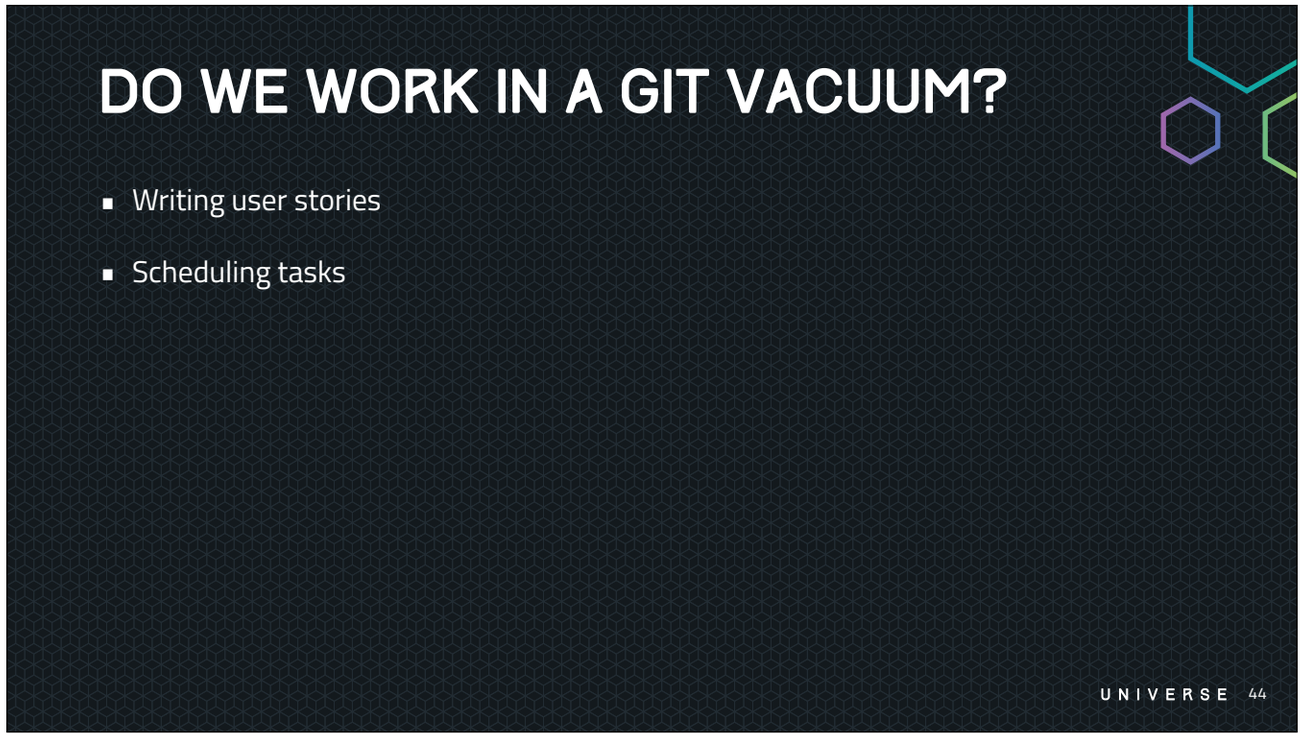
Branching only addresses one part of the equation. It doesn’t fully encapsulate concepts around planning, design, testing, acceptance, and scaling.




From writing user stories to scheduling tasks, to continuous integration, to code coverage, Git and branching are only part of the story. GitHub has this amazing capability of plugging into pretty much any one of these tools. So our conversation about workflow has to be inclusive of those other dimensions of our workflow.

Why wouldn’t we? We’d be cheating ourselves if we kept them out and, remember our goal, is to do our live’s best work. How can we do that if we have to continuously context switch?

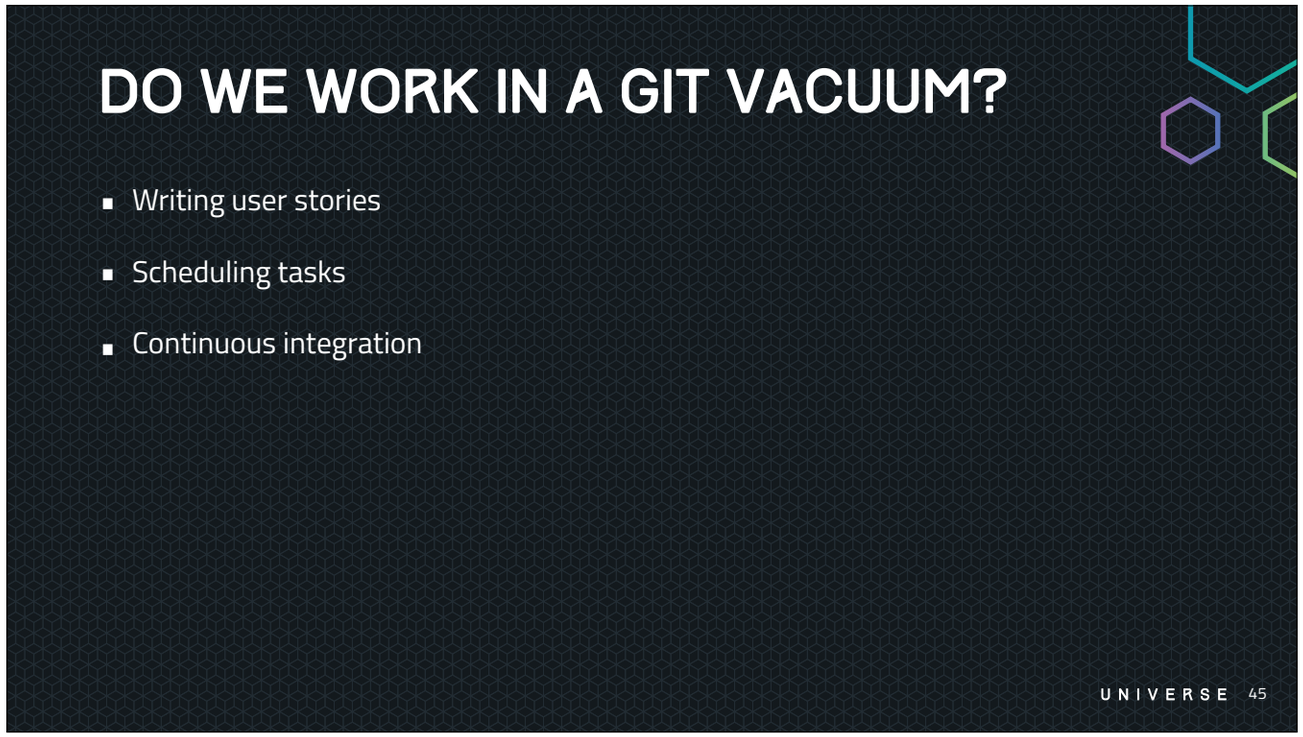
Branching only addresses one part of the equation. It doesn’t fully encapsulate concepts around planning, design, testing, acceptance, and scaling.




From writing user stories to scheduling tasks, to continuous integration, to code coverage, Git and branching are only part of the story. GitHub has this amazing capability of plugging into pretty much any one of these tools. So our conversation about workflow has to be inclusive of those other dimensions of our workflow.

Why wouldn't we? We'd be cheating ourselves if we kept them out and, remember our goal, is to do our live's best work. How can we do that if we have to continuously context switch?

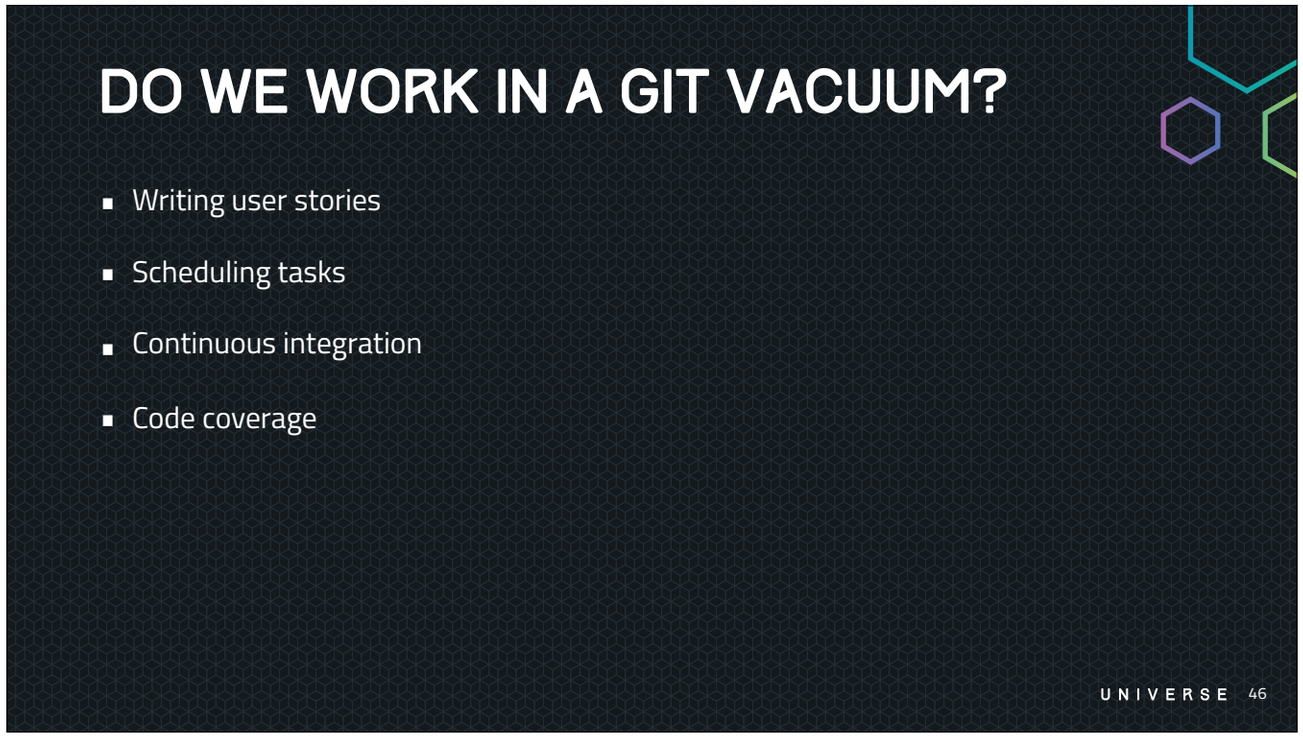
Branching only addresses one part of the equation. It doesn't fully encapsulate concepts around planning, design, testing, acceptance, and scaling.




From writing user stories to scheduling tasks, to continuous integration, to code coverage, Git and branching are only part of the story. GitHub has this amazing capability of plugging into pretty much any one of these tools. So our conversation about workflow has to be inclusive of those other dimensions of our workflow.

Why wouldn't we? We'd be cheating ourselves if we kept them out and, remember our goal, is to do our live's best work. How can we do that if we have to continuously context switch?

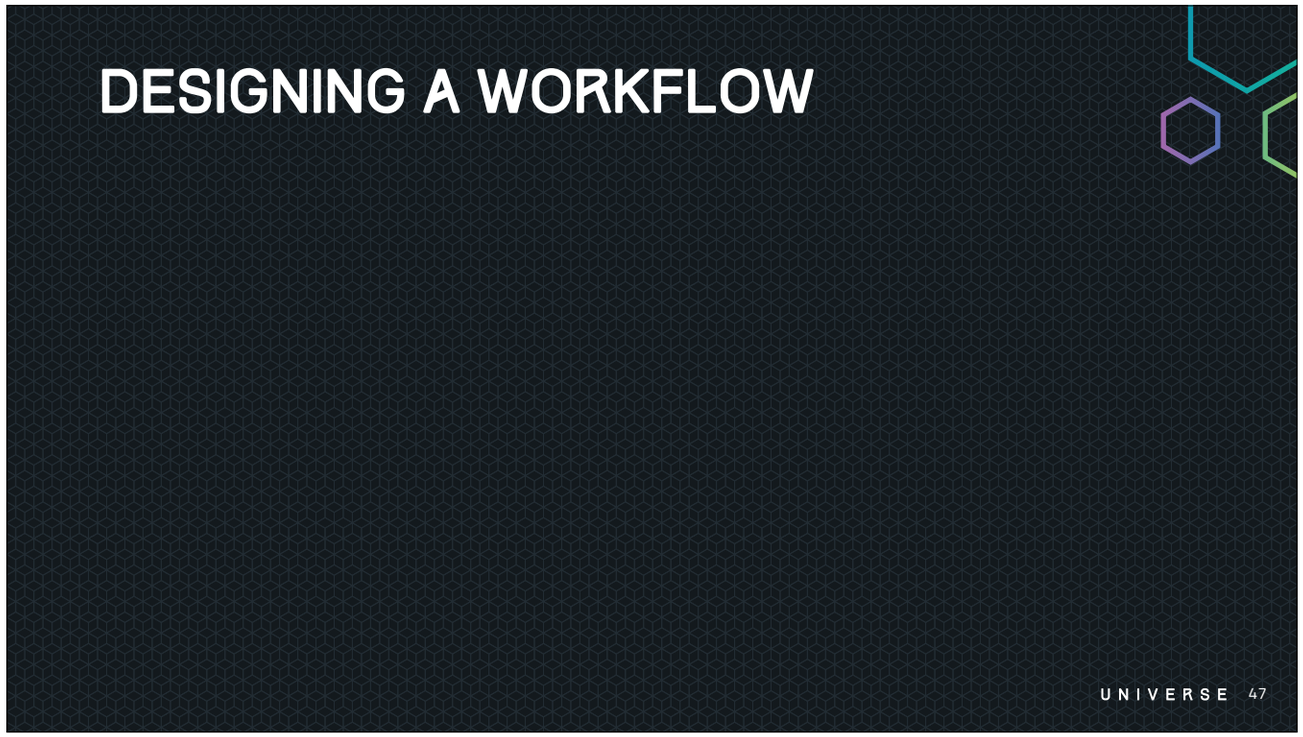
Branching only addresses one part of the equation. It doesn't fully encapsulate concepts around planning, design, testing, acceptance, and scaling.




From writing user stories to scheduling tasks, to continuous integration, to code coverage, Git and branching are only part of the story. GitHub has this amazing capability of plugging into pretty much any one of these tools. So our conversation about workflow has to be inclusive of those other dimensions of our workflow.

Why wouldn't we? We'd be cheating ourselves if we kept them out and, remember our goal, is to do our live's best work. How can we do that if we have to continuously context switch?

Branching only addresses one part of the equation. It doesn't fully encapsulate concepts around planning, design, testing, acceptance, and scaling.



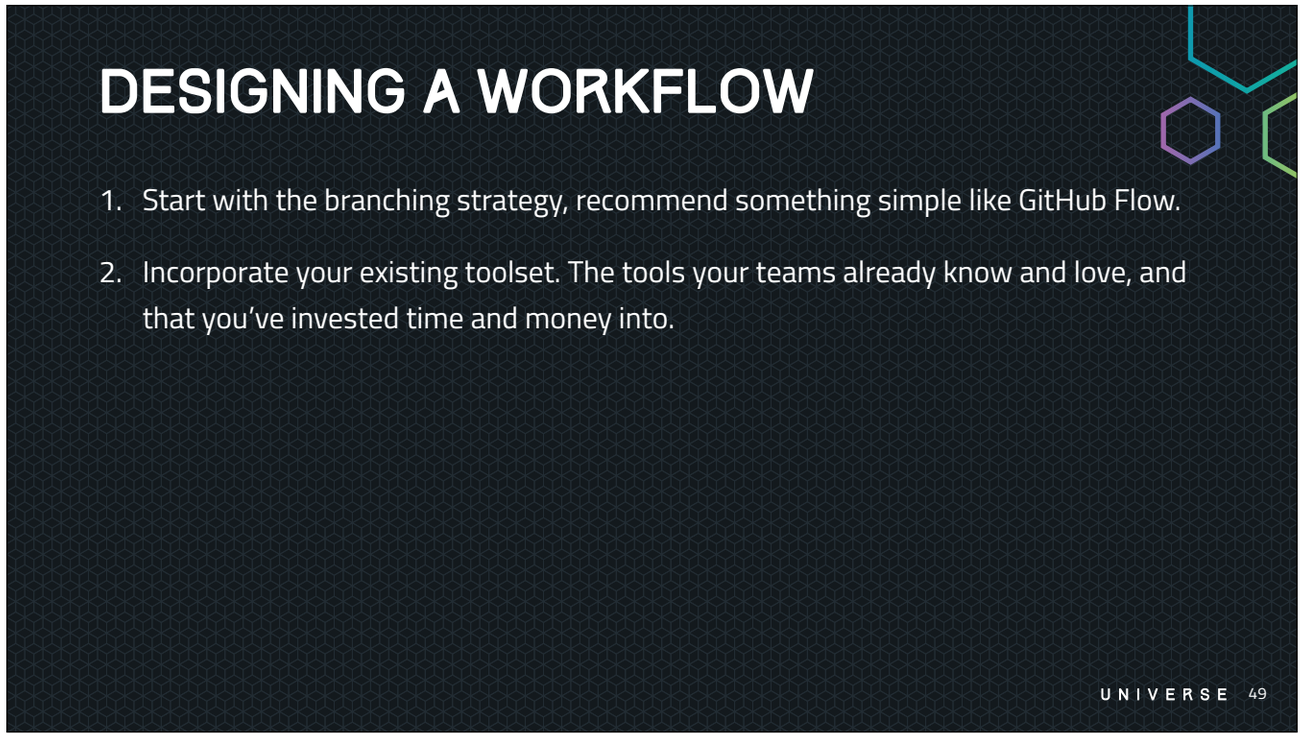

When we design a workflow, we must give everyone a seat at the table. This includes Project Managers, Developers, QA, leadership. When each of these roles uses their own tools, the isolated nature of the tooling leads to silos, chucking problems over the walls.

# DESIGNING A WORKFLOW

1. Start with the branching strategy, recommend something simple like GitHub Flow.



When we design a workflow, we must give everyone a seat at the table. This includes Project Managers, Developers, QA, leadership. When each of these roles uses their own tools, the isolated nature of the tooling leads to silos, chucking problems over the walls.




When we design a workflow, we must give everyone a seat at the table. This includes Project Managers, Developers, QA, leadership. When each of these roles uses their own tools, the isolated nature of the tooling leads to silos, chucking problems over the walls.

# DESIGNING A WORKFLOW

1. Start with the branching strategy, recommend something simple like GitHub Flow.
2. Incorporate your existing toolset. The tools your teams already know and love, and that you've invested time and money into.
3. Explore integrations that put you where you want to be.



When we design a workflow, we must give everyone a seat at the table. This includes Project Managers, Developers, QA, leadership. When each of these roles uses their own tools, the isolated nature of the tooling leads to silos, chucking problems over the walls.

# DESIGNING A WORKFLOW

1. Start with the branching strategy, recommend something simple like GitHub Flow.
2. Incorporate your existing toolset. The tools your teams already know and love, and that you've invested time and money into.
3. Explore integrations that put you where you want to be.
4. Iterate.



When we design a workflow, we must give everyone a seat at the table. This includes Project Managers, Developers, QA, leadership. When each of these roles uses their own tools, the isolated nature of the tooling leads to silos, chucking problems over the walls.

# LEVERAGE THE POWER OF GITHUB AS A PLATFORM



There are a few different ways to integrate with GitHub:

# LEVERAGE THE POWER OF GITHUB AS A PLATFORM

- Webhooks



1. *Webhooks* are GitHub’s notification mechanism. Actions on GitHub generate a payload that can be delivered to a web server. You could use Webhooks to, for example, trigger a new build on Jenkins every time someone pushes new commits.

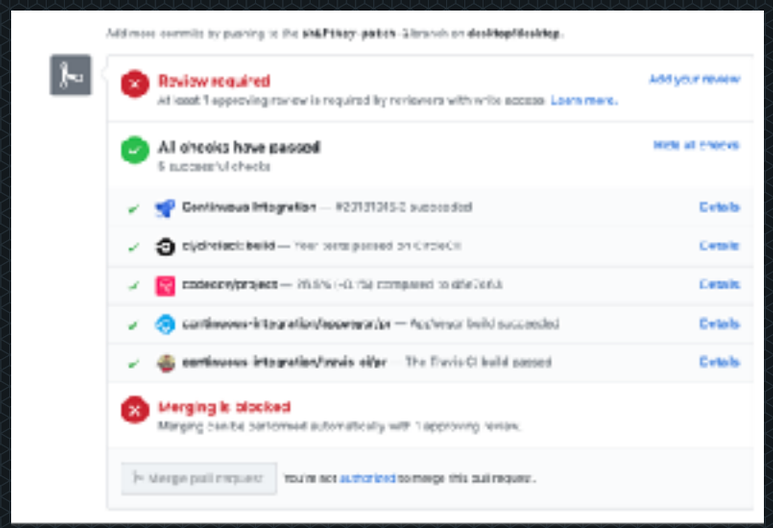
# LEVERAGE THE POWER OF GITHUB AS A PLATFORM

- Webhooks
- Apps



*Apps* are integrations that can also use the GitHub API. Some of GitHub’s API that is of interest to us here is

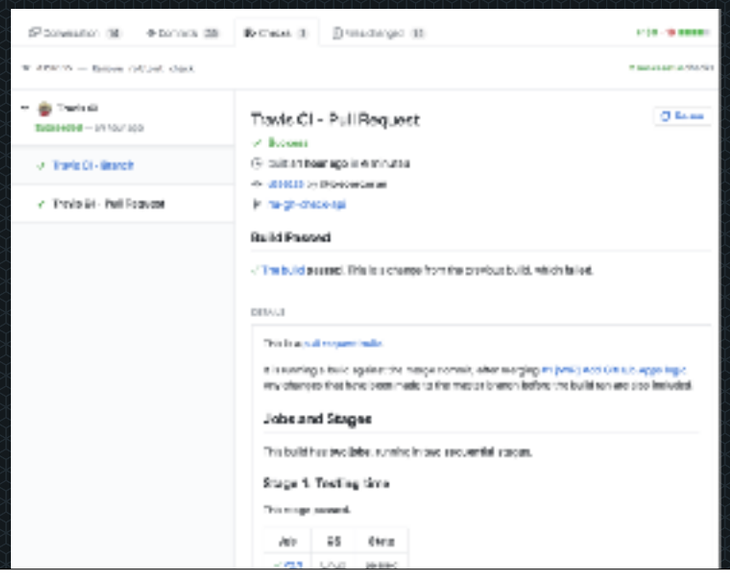
# LEVERAGE THE POWER OF GITHUB AS A PLATFORM






* statuses (show picture) which can tag a commit with one of three statuses: pending, fail, or success

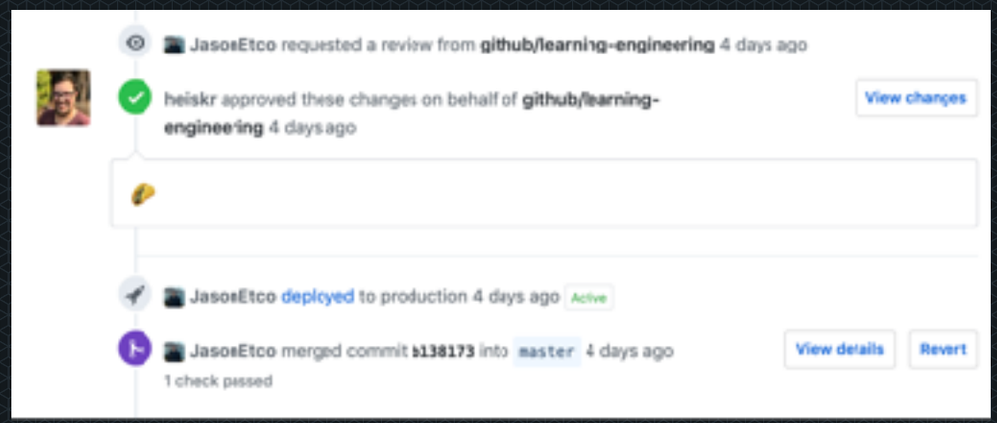
# LEVERAGE THE POWER OF GITHUB AS A PLATFORM






* checks (show picture) which levels up the amount of information and makes it context-sensitive

# LEVERAGE THE POWER OF GITHUB AS A PLATFORM






* deployments (show picture) which can automate deployments to your infrastructure.


We’ve got some demos set up that allow you to test out these