## Leveraging Trunk Based Development

### Module Overview

Let's take a look at trunk based development. I'll highlight the benefits of this approach and under what circumstances it could be a good fit. And to provide you with a balanced viewpoint, I'll highlight some of the potential drawbacks, too. I'll also include what types of engineering teams would best be served by trunk based development. Before I dive into the content, let's talk about an important point of clarification. While the workflow we'll cover in this module is most commonly referred to as trunk based development, you may occasionally hear it be called centralized workflow in some older documentation. They are the same thing. Here's a quick overview of what we'll cover. First, I'll answer the most fundamental question. What is trunk based development? I'll also provide some high‑level pros and cons of this approach to be aware of. Then, we'll explore how to implement this approach using Git on the command line. Next, we use our knowledge of trunk based development to determine what type of team would benefit most from this workflow based on both the unique characteristics of the team and the workflow. And finally, I'll highlight some of the key ways that you can advocate for and eventually implement trunk based development on your team. Let's focus on some key characteristics of trunk based development from a process standpoint. As the name suggests, there's a single branch, the trunk, usually called main, that contains all of the code. Engineers commit their code directly to main, and if your process requires short code approvals prior to production, you can have a really short‑lived branch, and that's fine, too. And these branches are designed to exist for less than 24 hours, and they only contain the work of a single engineer or a mob or pairing team on them, and they're not intended to represent large chunks of work or to be long living. A key feature of this workflow is that main is always ready to be deployed to production and that the code is always reflecting production's current state. In order to support that reality, a robust automated continuous integration and continuous deployment pipeline is a must. Feature flags also make this process much easier because they allow engineers to ship work in progress to production without impacting the product or the user experience. Throughout this course and in the industry in general, Git workflows are often represented with branch diagrams. I'll quickly highlight some of the key parts of this graphic so that you're familiar with them. First things first, diagrams in this course will always start on the left and move to the right. Most diagrams will have one or more branches, and it's typical to have the primary branch, in our case, main, at the top. Commits are indicated by dots. Here I'm referencing a commit on the main branch. And in addition, I'll add these boxes to indicate when there's a deployment to production. Because there's only a single branch, the entire flow is represented on that one branch with commits being added one after the other. As I mentioned earlier, one of the goals of trunk based deployments is to have a Git repository always match the production environment. To make this happen, teams that use this workflow will almost always deploy to production after every commit. This requires a very robust CI/CD pipeline. We're going to take a look at this and some of the other pros and cons next.

### Trunk Based Development Pros and Cons

Having a good understanding of any workflow is critical. So let's take a look at the pros and cons of trunk‑based development. Before we begin, I do want to call out that your perspective plays a huge role in this discussion. A pro for me might be a con for you in your organization and vice versa. So I'd encourage you to not look at pros as inherently good and cons as inherently bad, but to think about how each statement would be perceived within your organization. Let's start with the positive aspects of trunk‑based development. As we saw in the previous clip, the workflow couldn't be simpler. Each commit is pushed onto the main branch right after the previous one. Because of this approach, every commit must keep production stable so it can automatically be deployed at any time. The flip side of each commit automatically being deployed is that it greatly speeds up addressing production issues. You know exactly when new code was introduced to production and what that build contained. Ease of maintenance and lack of a burdensome process are significant improvements for the engineer experience. And finally, because commits are small, the chance of merge conflicts happening are greatly reduced. Next, let's take a look at some cons of this approach. As I mentioned earlier, these items are all about the context within your organization, and just because they're on my con list doesn't mean that they're on yours. For starters, trunk‑based development requires a robust CI/CD environment, including automated testing and a really good deployment pipeline. While this is the gold standard for most organizations, some of us just aren't there yet. Next, since everyone is committing to the same branch that immediately goes into production, there's a natural cap on the number of engineers that can contribute code at once. I've seen this workflow work really well with teams of up to eight engineers, but beyond that would probably be pushing it. One way to make the repository more accessible to both engineers and to larger features is to use feature flags, allowing you to ship code that's not ready for prime time and therefore isn't visible to the user. But if your organization doesn't have those, that can be a problem. Next, there's an implication that this process works better with more senior engineers since the risk of them accidentally shipping nonfunctional code directly to production is lower. And as a result, teams may choose to focus on pairing and mobbing instead of using individual work patterns such as pull requests Finally, this process has far less traceability and auditability built in compared to more robust workflows and specifically in the area of code reviews. Next, I'll demo a repository setup with trunk‑based development workflow using the command line.

### Demo: Using Trunk Based Development via the CLI

Now that we have a basic understanding of trunk‑based development, let's try it out. We'll start by reviewing the sample repository I created on Bitbucket. More information on why I've chosen Bitbucket in a second. Next, we'll clone that repository to our local environment, and then we'll make a small change to one of the files, just enough to trigger a new Git commit. Then, we'll commit and push our changes back to the remote repository's name branch. And finally, we'll observe our change back in Bitbucket. For this course, I'm making an intentional choice to use Bitbucket because it allows us to see the visualization of the Git branches, which will be more important as the workflows become more complex. This is not an endorsement of Bitbucket over GitHub. And since the foundation of any Git workflow is Git, either platform handles them equally well. If you're comfortable using Bitbucket, I'd encourage you to sign up for free count. Otherwise, I'll include some information on how to use GitHub as well. All right, enough setup. Let's get to work. I've created this short link to help you get to the repository quickly so you don't have to type the long URL. Let's take a quick look at the code. This is a very basic repository with just an index page and some CSS to go along with it. The code that's actually in this repository doesn't really matter and neither do the changes that we'll need to make. They're merely to serve as a reference point for us to build Git workflows around. Next, let's check out the Commits tab. As I mentioned earlier, this is the functionality that Bitbucket has available out of the box. If you're not using Bitbucket, there are ways of getting this visual in other ways like a desktop Git client or a Chrome extension on GitHub. But because this is a trunk‑based workflow, there's only this single bridge displayed on the Commits page. And since this is a public repository, it is read‑only so we'll have to fork it. Let's head back to the source page of the repository to get this done. If you already have a free Bitbucket account or are willing to sign up for one, we can simply fork it from here, and you can find the Fork option under the three dots to the right. The options that you pick on the screen don't really matter for this course. And when you're done, click Fork repository, and it will take a few seconds to generate. You now have your own copy of the repository, so go ahead and clone it to your local environment. Before I do that on my system, I'll quickly provide the forking instructions for GitHub. So back on the source page of the original repository, copy the full link to the URL. It does need to be the full link, and you can't use the short link because GitHub doesn't process the shorter link redirect. With the link copied, head on over to GitHub and log in if you're not there already. Next, click the plus and select Import repository. And this is where you'll paste that repository URL you copied earlier. And go ahead and pick whatever options and name that you would like and click Import. And then, after a few seconds, you'll have your own copy of the repository in your GitHub account, and you can clone it to your local environment. I've got my local copy of the repository open in Visual Studio Code. So let's go ahead and make a quick change. One of Carved Rock Fitness's customers let the support team know that the home page still has the Google+ logo in the footer, and this is a pretty awkward miss, but we can fix it really easily. So I've got the index file open, and I'll scroll to the bottom. And we can see that pesky line right here, and we can quickly delete it. So those are all the changes that we needed to make, so I'm going to go ahead and commit and push our changes. Go ahead and do that using your preferred method. I'm going to quickly add and commit that change, and then I'm going to push it to origin. Another quick note here if you're using Bitbucket. If you're prompted for your password and this is not your regular password, but instead an app password that you'll need to create, you can do so under personal settings by clicking the avatar in the top‑right corner of any Bitbucket page. If you're using GitHub, you won't be able to follow along unfortunately. Let's head to our Commits tab, and we'll see our newly added commit. There at the top of the page, you can see our new commit, and you can also see that the branching diagram is still just a straight line. Now at this point, you're probably thinking, wait, isn't this just how normal Git works right out of the box? And the answer is yes. That's what's so great about trunk‑based development. So next, we're going to talk about which teams and which scenarios might benefit from trunk‑based development, and we're also going to cover how you can advocate for it and implement it if it's the right fit for you.

### Leveraging Trunk Based Development

Let's review what type of teams would benefit from trunk‑based development, and then we'll use some of the rationale from that decision to help you fashion some talking points for your leaders and teams. And finally, we'll talk through how to implement trunk‑based development in your space. Carved Rock Fitness is a large outdoor retailer with a significant IT workforce that includes several engineering teams. And while your organization most likely won't be exactly like Carved Rock Fitness, my hope is that you'll be able to relate to some of the challenges that one or more of the teams face, and I would also encourage you to think about how your teams are similar or different than the Carved Rock Fitness teams. Let's take a quick look at some of the unique characteristics of each of the teams. The first and biggest team is Carved Rock Fitness's Ecommerce team. This team is actually several smaller teams that has over 100 engineers total, and given the size of that team, there are many different experience levels that we need to contend with. Another big challenge for this team is that their code base is a single massive repository, or a monolith, that contains all of the functionality. And since this team deals with customer data and financial records, they have to follow SOC 2 compliance processes, and there's a lot of factors in SOC 2, but the most relevant one for this discussion is that the code has to follow a really strict process with proper sign‑ups including code review, testing, and deployment traceability. Next, we have the Innovations team, who focuses primarily on creating new customer experiences and they have minimal access to existing customer data. The team only has five engineers, so it's much easier for them to be on the same page about when they're going to release a feature and what they're working on right now, and they're also a big proponent of pairing and mobbing. And they are mainly a senior team, which allows them to move pretty quick. Another big benefit they have is that they're working on their own code base away from the monolith team. Because they're focused on creating value quickly, the team is small and nimble, and they have a really robust continuous integration, continuous deployment pipeline to help them ship code really quickly. Finally, we have the Content Management team, who is responsible for deploying all of the code on Carved Rock Fitness's public‑facing content management site. The team is a little bit more manageable, it's got 10 engineers on it, and the team is primarily senior, but there are a few earlier‑in‑career folks on there too. Fortunately, the Content Management team also has a single repository that's separate from the big monolith, that allows them to make changes easier. However, the team is distributed across the globe with half of the engineers on one continent and the other half on another, and there's a couple of different time zones in between them, so they only have a couple hours of overlap where they're both online at the same time each day. Now that we have the context about the different teams, let's take a look at some of the key features of trunk‑based development and see how they match up with each of the requirements for the different teams. As we go through this process, think about how you would answer these questions for your teams. As we saw in the demo, the workflow is very streamlined and simple. However, that simplicity works best when the team is smaller and has similar working hours. Because of that, the Innovations team is a good fit, but the other two are not. Since all of the commits are going directly to production, having a more senior team is beneficial. This rules out the Ecommerce team, is a plus for Innovations, and a maybe for the Content Management team. Next, we need to have a smooth, well‑built and automated CI/CD pipeline. Due to the Ecommerce team's auditing requirements, that's challenging. The Innovation team already has one built, so it's a great fit for them. And the Content Management team relies on the underlying content management system, which limits the number of deployments they can do, making this category a no for them as well. While having the last commit represent production is great for a small and nimble team like Innovations, it's a nightmare for a large monolith repository like Ecommerce, since that commit could represent any team. Similarly, it's a challenging proposition for a globally distributed team like Content Management. The Ecommerce team is way too big for this workflow, Innovations is just right, and the Content Management team is right on the border. The larger the team, the harder trunk‑based development becomes to manage. As I mentioned, this workflow is best for teams that prefer pairing or mobbing, like the Innovations team. Even if parts of the team follow this approach on the other two teams, there's still considerable risk of miscommunication that can be lost on the way to production. And finally, the limited traceability is a nonstarter for the Ecommerce team's audit requirements. Neither Innovation nor Content Management teams have a requirement, so they're compatible with this. So operating under the assumption that your team is a good fit for trunk‑based development, let's take a look at some ways that you can advocate for it. Highlighting that it's a simple and clean workflow is a great place to start. It's literally Git at its most fundamental level. Building a good continuous integration and continuous delivery pipeline takes time and effort. This workflow is the best way to take full advantage of that investment, and because each commit is shipped directly to production, customers can receive value much quicker. Also, if we find a bug in production, tracking down the commit that caused it is quick and easy too. And finally, something everyone can get on board with, engineers spend less time jumping through hoops and more time writing code. Think about some additional ways that you can think of advocating for trunk‑based development in your environment. Fortunately, implementing trunk‑based workflows is really straightforward. First, make sure that your CI/CD pipelines and feature flags are in working order. Next, have informal conversations and workshops with your engineers. Consider setting up a practice repository that deploys automatically to a non‑production environment for them to practice in. And once everyone feels comfortable, you can merge and then delete any outstanding branches. Sometimes old habits can be hard to break, so keep an eye out for branches that pop up unexpectedly, and keep having those coaching conversations to help prevent it from recurring. Let's summarize what we talked about. First, we became familiar with how trunk‑based development works, as well as its pros and cons. Next, we walked through a really simple demo which helped us to realize that trunk‑based development is just Git out of the box. We talked about which Carved Rock Fitness teams would benefit most from this workflow with the goal of helping you to think about how this approach could work in your environment. And finally, we finished up by talking through how to advocate for and implement trunk‑based development. Next, we'll take a look at Git flow. Git flow is a really robust and pretty complicated workflow, so I hope you'll join me.

## Understanding Git Flow

### Module Overview

Let's walk through Git Flow. This is the most involved of all of the Git Flows out there. But don't worry. I'll take plenty of time to walk through it. I'll highlight the benefits and potential drawbacks of it, and we'll examine the circumstances that it could be a right fit. I'll also include what types of engineering teams are best served by the Git Flow pattern. So here's a quick overview of what we'll cover. First, we'll start by answering the question what is Git Flow? And then we'll also address its benefits and drawbacks. I'll leverage Bitbucket's Git Desktop tool, Sourcetree, to implement Git Flow in a demo. And then we'll repeat that same process via the Git command line. Next, we'll use our knowledge of the Git Flow pattern to determine what type of team would benefit most based on their unique characteristics. And then we'll spend some time discussing how to advocate, coach, and finally implement this complex workflow. And finally, I'll highlight some variations of Git Flow that have become popular in the recent history. Let's take a look at some of the high‑level characteristics of Git Flow. For starters, in addition to its strong traceability, auditability, and testing support, Git Flow is great for software products that have specific releases. In addition to this robust release process, there are also established ways of deploying urgent bug fixes to production. And unlike other workflows, Git Flow has multiple long‑living branches. The main branch still represents production, but is restricted from receiving direct commits. Most hosted Git repositories allow you to enable branch protection to make this happen. Next, there's develop this branch contains the code that will go into the next release. Another way to think of it is is that the develop's branch represents production's future state after the next deployment. Develop is often set up to not allow direct commits as well. In addition to those two long‑living branches, there are also three categories of shorter‑living branches. Feature branches represent the code for a feature that a team or individual engineer is working on. These branches are always branched off of develop. When the feature is completed, it's merged back into develop, and the feature branch is deleted. When a critical mass of features are merged into develop or a cutoff date approaches, it's time to create a release branch. This branch is cut from develop and represents a snapshot of a point in time on develop that's ready to be deployed to production. For auditing and testing purposes, it's best to avoid changing this release branch as much as possible, limiting direct commits to bug fixes and similar tasks related to that specific release. Once all the testing and other release tasks have been completed, the release branch is merged into main, signaling the deployment of new code and is tagged with a version number matching the name of the release branch. The release branch should also be merged back into develop to catch any bug fixes that were committed to the release branch only. And once you've done all of that, you can delete your release branch. As I mentioned earlier, Git Flow also has a specific process to address critical bug fixes. Rather than going through the whole process I just described, the hotfix branch is cut directly from main, and all of the bug fixing happens on that branch. Once the fix is completed, it's merged directly into main, again tagging with a virtual number. And the hotfix branch is also merged down into develop so that the change isn't overwritten during the next deployment. That was a lot of information on a pretty complex topic, so let's visualize it. I'll be using a Git branch diagram, which is a pretty common industry visual out there. This diagram will have time moving from left to right. Most diagrams will have one or more branches, and it's typical to have the primary branch, in this case main, at the top of the diagram. Commits are indicated by dots. Here I'm referencing a commit on the main branch and one on develop. In addition, I'll add these boxes to indicate when the deployment to production happens. Note that they also have a version number. So let's get started. A team is working on feature A so they create a new feature branch off of develop. Curved lines leading from one branch to another indicate that a new branch has been created based off of the commit at the start or left side of the line. This process is called branching. The team is really confident in their work and gets everything done in a single commit, and they're ready to send their code back to deploy. Moving from one branch back to the original branch is merging. And to indicate that we're using pull requests to complete this process, I'm using a thicker line. Once the team's work is merged, the feature branch can be deleted. And because this is an important feature, we want to get it out right away and create a release from develop, which has the feature in it already. And then during testing, a bug is found in the release, and the team commits the fix directly to the release branch. The release is then approved by all of the necessary parties and is merged into main via pull request. The main branch is then built and pushed to production. And so to ensure that that bug fix that we found in the release cycle isn't overwritten in a future release, the release branch is also merged back into develop, usually with a pull request for auditing purposes. So now that the code is in production, an urgent issue is discovered, and the team decides to create a hotfix to address it. A hotfix branch is created off of the main branch, and the team addresses the issue. And once the fix is tested and code‑reviewed, it's merged back into main via a pull request. The code is then deployed with a new version tag. And just like the fixed during the release cycle, the hotfix is merged back down to develop, also with a pull request to prevent it from being overwritten. While all this is going on, a different team has diligently been working on their own feature, B, which they split off of develop. And as they continue working on their feature, they're adding more and more commits. As they're getting ready to finish their feature, they hear about the hotfix that the other team made and decide to pull it into their branch before attempting to merge. This allows them to handle any merge conflicts on their own terms without blocking develop for everyone. Once they fix the merge conflicts, they merge their feature into develop via a pull request and delete their branch. And if there are any additional features ready to go, they would follow the same process. Now it's time for a new release branch to be created, developed, and tested. This release doesn't have any issues that need to be addressed, so it can just be merged right to main via a pull request. And just as before, main is deployed to production and now accurately reflects the code on these servers. And finally, even though there weren't any additional changes, the release branch is merged back into develop and closed.

### Git Flow Pros and Cons

Having a good understanding of a workflow is critical, so let's look at the pros and cons of Git flow. It's important to look at them through the lens of your organization and decide if they're properly categorized. You shouldn't look at a pro as inherently good or a con as inherently bad, but think about how each statement is perceived in your organization. Let's get started with the positive aspects of Git flow. The primary reasons that organizations choose to use Git flow is its strong fit to their existing compliance and testing requirements. Git flow is very popular in highly regulated industries like healthcare and finance. As the number of engineers contributing to a single repository grows, it becomes harder to track who is doing what. Being able to manage work in progress across a large team is one of Git flow's strengths. Similarly, Git flow makes it easy to track the progress of features being developed, since each new feature has its own branch. And finally, Git flow allows for multiple versions to be in the pipeline. This is great for strict release cadences and other challenging release scenarios like submitting to the Apple Store for iOS apps. Next, let's look at some of the cons of Git flow. As I mentioned earlier, these items are all about context within your organization. Just because they're on the con list, doesn't mean that they're necessarily a detriment in your environment. First, the learning curve for Git flow is very high. When I work with engineers starting on Git flow for the first time, helping them to understand what happens to their code once it leaves their machine is pretty challenging. A big reason why Git flow is so hard to learn is because it's very complicated and complex. While it provides a lot of structure, understanding and following that structure is a challenge. Speed to production slows way down on Git flow compared to other approaches. However, to be fair, organizations that leverage Git flow often have lengthier change management processes that greatly overshadow the time that Git flow itself takes. As the team and number of features in a repository grows, many organizations find themselves needing a dedicated role to manage the inner workings of Git flow. This person creates releases, decides what goes into them, and how to handle hot fixes and similar issues. Another challenge in Git flow is frequent merge conflicts as features are merged into develop or added to releases. For organizations with release managers, handling these conflicts is often a requirement for that role. I'll highlight a helpful tool that will make working in Git flow much more manageable, next.

### Using the Git Flow Extension

As we are walking through the branching diagram for Git Flow, you're probably thinking of all of the commands you'll have to write, and there are a lot of them. Fortunately, there's a great extension that automates most of this, so let's jump right in. A quick Google search for Git Flow extensions reveals a couple of different options, so let's talk through which one you want. Git Flow AVH, or a virtual home, is the current version. You can find installation instructions on their GitHub page. But if you're using Git for Windows, it's already included in there automatically, and it's also included in many of the leading Git GUIs, such as Sourcetree. The original Git Flow extension hasn't been updated since 2012 and lacks many of the new features like the ability to have custom hooks and naming your primary branches. So if this is the URL you're on, it's the wrong version, and you want the new one. Now that we know where to get the extension, I'll walk through the commands that it offers and what they do. Just a quick note, I'm using the command line version for simplicity. If you're using a GUI, the same options will be available to you, but in a menu structure instead. First, we'll need to initialize the repository with init. This command is run on an existing Git repo and will ask us which branches we'd like to use for our main and develop functions. We'll also be able to specify how we want hotfix, release, and feature branches to be prefixed. Typically, it's best to just leave the defaults. With our repository ready to go, we can start working on our first feature. The command feature start along with the name of our choice will create a new branch from develop. You'll then be switched to that branch, and you can begin working on your feature. Using the publish command and the branch name, you can push your features to origin for others to access. Honestly, I just use git push origin. The extension doesn't really add any special value here. If you're not using pull requests or protected branches and you can merge directly into develop, the feature finish command will merge your feature branch into develop for you. Switch your local environment back to development and delete your feature branch. This approach does require every engineer to have the ability to write directly to develop, which many organizations choose not to do. If that's the case in your organization, you can manually create a pull request into develop and then do the local cleanup yourself. A quick note that this extension does allow you to create custom hooks, so you could write a hook to create this pull request for you. The releases commands follow a similar pattern as the feature ones. Creating a new release branches off of develop with the name of your choice. And just like the feature commands, you can publish it for others to consume. And finally, if your organization allows merging directly into main, you can use the release feature command. Because this action breaks many of the common change management controls, it's probably unlikely that this is the case for you. In that case, you would have to manually create a pull request into main from your release branch and do the cleanup on your own again. Hot fixes follow a similar pattern in that the star command is far more useful than the finish command. The hotfix star command simply branches off of main, creates a new branch with the name you specified, and then switches your local environment to that branch. The finish command is also not very useful in pull request environments since it tries to merge your code directly into develop, as well as into main. If while you're watching this course you thought that this extension isn't particularly valuable or doesn't save you that much time, that's totally fine. I know lots of engineers working with Git Flow that never use the extension or use it initially and then switched to plain Git commands once they are more comfortable with the workflow. This extension is helpful for some, but completely optional. Next, I'll walk through initializing Git Flow on a repository and creating and releasing a feature using Sourcetree, a free Git UI tool.

### Demo: Using Git Flow with SourceTree

Git Flow is one of those processes that's really hard to explain in the abstract. So I'm just going to do a realistic example instead. Just a heads up that I'll be using the free Get UI Sourcetree to make these changes. In the next demo, we'll be using the Git CLI. We'll start by forking the repository on Bitbucket. More on why I've chosen Bitbucket in a little bit. Next, we'll clone that repository to our local environment, and then we'll use Sourcetree's Bit and Git Flow menu to initialize our repository to use Git Flow. Then we'll create a new feature and push some code to it. Once we've made our code changes, we'll finish the feature and push our changes back to Bitbucket. And finally, we'll create a release that has our feature in it, and we'll finish that too. For this course, I'm making an intentional decision to use Bitbucket because it allows us to see the visualizations of the Git branches, which will be important as this workflow becomes more complex. This is not an endorsement of Bitbucket over GitHub, and all of the workflows that we'll cover in this course work equally well on either platform since the underlying structure is just Git. If you're comfortable signing up for Bitbucket, I'd encourage you to do so since the accounts are free. You can still get some of the functionality in GitHub, but just not all of it. If you haven't already, download and install Sourcetree onto your machine. I've included a short link here for your convenience. Now let's get started. I have the repository we'll use pulled up, and here's a short link to get you there quickly. Because we're learning about Git, the functionality of the code and the repository isn't particularly important. It's a simple web page with some CSS. Now let's check out the commits for this repository so far. There's only a single line of commits, but we're going to expand this tree with multiple branches as we implement Git Flow. While this is a public repository, it is read‑only so we'll have to fork it. We'll head back to the source page to get this done. Now if you have your free Bitbucket account or you're willing to get one, you can simply fork it from here. You can find that option under the three dots to the right. Next, you'll be able to assign or create a project that will house the copy of your repository. Pick whatever name you would like. It's not important for this course. And you can also leave the repository public or private. It doesn't really matter for this course either. When you're done, click fork the repository, and it'll take a few seconds to generate. If you'd prefer to stick with GitHub, copying the repository there is pretty straightforward too. Back on the source page of the original repository, copy the full link to the repository, and it does need to be the full link. You can't use the short link because GitHub doesn't process the redirect. And if you open up GitHub, click the plus, and import the repository. That's where you can paste the repo URL and choose whatever name you'd like. Again, you can choose whether you want the repository to be public or private. It doesn't matter. And then click Import. So let's use Sourcetree to clone the repository in our local environment. Since Bitbucket and Sourcetree are both owned by Atlassian, there's a handy clone in Sourcetree button here that will do the cloning work for us. If you're using GitHub, you can copy that repo URL and manually clone it into Sourcetree. In Sourcetree, changed the path to where you want to save the repository if you'd like and then click Clone. If you haven't used Sourcetree before, I'd encourage you to pause the video and take a look around. Now let's start working on Git Flow. Let's initialize the repository first. You can find that in the Repository menu, Git Flow, and then Initialize Repository. We're prompted with a window that allows us to pick branch names for all of the different stages of the Git Flow. We'll want to change our production branch from master to main in order to match our current repository structure, and we can leave all of the other defaults in place and proceed. You can select show output if you'd like to see the commands that Sourcetree is running on your behalf. I'd encourage you to do so until you're comfortable with the different Git commands that the Git Flow extension executes for each step of the process. Once it finishes, you can see that there's a new develop branch and that we've been switched to it. Let's push our new develop branch up to our remote repository too. We'll click on Push, check the box next to develop, and then click Push again. Now let's do some feature work. The marketing team wants to launch a new promotion so we can make a new feature with Git Flow. Back under the Repository menu and then Git Flow again, let's select Start New Feature. This will launch a new dialog allowing us to name our new branch, which we'll call new‑promo. We now see our new branch on the left here and the feature prefix in front of it. We're just going to make a few minor tweaks to the index page. What we change isn't important. We just need something so that Git has a change to track. I'm going to bump this discount percentage up slightly, and then we'll save the file. Let's head back to Sourcetree. In the File Status section, we can see that we have one changed file ready to commit. Let's stage that file and commit it. With that done, let's finish our feature, which we can do back in the Git Flow menu. Notice that a branch is automatically selected for deletion in an attempt to keep our Git tree tidy. The Sourcetree GUI does not have the publish feature option in Git Flow, so I'll have to manually push develop back to the origin. So let's do that now. Notice our main branch is now behind develop. This is correct since it reflects reality. We have code that's been merged from a feature branch in to develop, and it hasn't been released yet. So let's fix that by creating a release called 1.1 from the Git Flow menu. In the output, we see that the new branch release /1.1 was created from develop, and there's some helpful reminders on what to do next. Because we don't need to add anything else to this release, let's go ahead and finish it. The finish release window has a number of helpful options, including adding a tag. It's common practice to tag releases with the version number since tags persist even after branches are deleted. We can also push these changes directly to the remote repository, in our case Bitbucket. Finishing releases does the most work behind the scenes, so let's check out the output to see all of the Git commands that are executed. First, Sourcetree checks if we need to pull any other updates to main. Then it switches to main and merges our release in. Then it determines if someone else had pushed the same release at the same time. With the main branch in a good state for now, Git Flow focuses on develop and confirms that the code in the release was already in our local branch. Remember how we checked the Push changes to remote box? Well now, Git Flow is going to push not only develop and main to origin, but the new tags for 1.1 as well. And then the final action Git Flow completes is to delete our local release branch. Sourcetree gives us a summary of all of the actions it completed and then switches us back to develop. So let's review the commit graph that Sourcetree provides for us. The blue line represents the develop branch, and we can see all of the commits that predated it using Git Flow. The next commit was merging the new promo feature. Because we only had a single commit on that feature, the commit message of updating percentage was used in the develop merge commit. Next, we have the commit that merged the release into main. Sourcetree is representing all activity on main in pink. Notice the separate pink line that starts from the pre Git Flow commit. This matches what happened when we first set up Git Flow. We created a new branch called develop off of main and switched it to the primary branch. The diagram looks like it's in reverse order, but that's because it's representing the new reality of develop being the repository's primary branch instead of main as it was before Git Flow. Next, notice how the main commit is coming up from the developer branch after our updated percentage. This subtle distinction highlights that a release branch facilitated this merge into the main branch. And finally, we can see that the main branch was merged back into develop. We've now gone through the entire development process using Git Flow, but there's one more lingering question. We are able to do all this because we're able to push directly to main and develop. And in most organizations looking to leverage Git Flow, branch restrictions are in place, and direct commits into protected branches like develop and main are blocked. Pull requests must be used instead. So in the next demo, we'll address the scenario using the Git command line. So I hope you'll join me.

### Demo: Using Git Flow with the CLI

Before we begin, if you skipped the previous demo, I'd encourage you to go back and watch it so that you can get your repository set up correctly. We'll be picking up where that clip ended. First, we'll start by leveraging the Git flow extension in the terminal. While this means we'll need to remember the names of the commands we want to run, it will give us more flexibility. Most organizations leverage in Git flow because they have specific controls on how code can be merged into production branches. So to represent that, we'll configure our repository to use branch restrictions to support this workflow. Similarly, the example of a single feature happening at a single point in time is pretty simplistic, so we're going to increase the complexity and make it more realistic by working on two features at the same time. Next, because our repositories develop and main branches no longer allow direct merges, we'll have to finish our feature using pull requests. We have also been informed that there is a production issue that we need to address immediately, so we'll handle this through a hotfix. And finally, we'll create a release containing our features in it and get it to production with a pull request. Let's start in Bitbucket where we'll configure our branch protections. Before we begin, a quick note for our GitHub users. Branch protections aren't supported on private, non‑enterprise, and non‑team accounts. Let's configure some settings to make this repository easier for Git flow to use. First, we need to change the default branch to develop. This will make tracking and consuming the repository easier for future engineers. Next, Bitbucket allows us to specify a branching model, which conveniently matches our Git flow perfectly. We'll specify main as our production branch, and now we can add some branch protections. We'll do develop first. Bitbucket allows us to add restrictions by branch type based off of our branching model. We don't want anyone to be able to write to it directly, but we do want to allow everyone to merge to it via pull requests. And we'll follow the same process for our Production branch too. Many organizations restrict access to merge pull requests to Production to certain individuals or teams, which can be configured here. We'll leave this as is for the demo, but know that it is a feature that does exist on Bitbucket and in a slightly different fashion on GitHub as well. Now that our repository matches what most organizations have for compliance purposes, we can get back to our local copy. With that in place, let's head there now. I'm using the built‑in terminal for Visual Studio Code, but any terminal will work. Before we start working on new features, let's make sure we're in a good starting point. First, confirm that we have Git flow installed and working in the terminal. If this command doesn't work for you, check out the clip on the Git flow extension on where to get it. Next, we'll confirm that there is nothing outstanding to commit and that we're on develop and in sync with our origin. Looks like we're in a good spot, so let's get started. We need to end the promotion that we created in the previous clip, so we'll start a new feature for that. We'll pass feature start and the name of the new feature to the Git flow command to kick us off. Just as in Sourcetree, the Git flow extension provides a summary of what it did on our behalf. Let's do a quick git branch check to see what branches we have locally. As we'd hoped, we still have develop and main, as well as our new feature branch, which we're currently on. Let's revert that discount percentage back to 20%. In real life, obviously, we'd be making more substantial changes in a feature, but this is good enough to trigger a change in Git. We've just been informed that we also need to work on a new feature to update some site contact to reflect a new spring promotion, so let's create a new feature for that work as well. First, we'll verify that we're looking good with git branch. Everything checks out, so we'll do git status too. Well, this is awkward. Looks like I forgot to commit our discount percentage change for our promo ending feature. Fortunately, git flow is just a layer on top of regular Git, so I can just switch back to our other feature branch and then do a commit. And finally, another get status to confirm that I'm good before I switch back to our spring‑promo feature. Sorry about that detour. Let's get back to our second feature branch now. Some of the details of the new promotion are still being worked out, but we know that it will be the UPCOMING SPRING, so let's update that text for starters. And then we'll commit this change until we have some more specifics on what else needs to be updated. We just got a message that the discount percentage in the spring will be 45%, so we'll update that now. And that's the extent of the promotion changes, so let's go ahead and commit them. Now because we can't merge directly into our develop branch, we'll have to publish the feature and then handle the merging via pull request. Passing feature publish and the branch name to Git flow will take care of that for us. Once the command executes, check out the output, and you'll see another summary of what was done. What's even more interesting is a little higher up. The extension has formed a link to Bitbucket for us to create a pull request, so let's go ahead and click that and check it out. We can see this pull request has all the information we would expect to see including the titles of the commits in the description. We'll delete the branch after it's merged since the feature will be completed then. The diff and the commit history look good too, so let's go ahead and create that pull request. Now let's address our other feature. It's exactly the same process. I'd encourage you to do it on your own without following along for practice. One quick note before you begin. You don't need to be on the promo‑end branch to publish that feature, so no need to switch to it unless you want to. So I'm going to go ahead and quickly repeat that process for the other feature, and then I'll create the pull request in Bitbucket too. With our features merged to develop, let's take a quick look at the commits page so we can check out the visualization of the commit tree. As you can see by the red line, the promo‑end feature was branched off of develop, which was represented by the green line. And then you can also see that it exists separately while on the blue line, which is where the spring‑promo feature was being worked on. And this is the power of Git flow. Both features were easily managed concurrently without any blocking of repositories or the team. All that's left for this feature work is to update our local environment back in the terminal. Unfortunately, we can't use the Git flow finish feature since it will try to merge it back into develop, which we can't do. But fortunately, the commands we have to run by hand aren't too cumbersome. First we'll switch to develop and pull all the commits from our merge pull request down. Next, all that's left to do is to delete the two feature branches. Now, if we do another git status, we can see that we're up to date with our origin repository and git branch shows us that we're back to our original state with the main branch and develop being cloned locally and develop as the active branch. Things are in great shape for our feature work. Unfortunately, one of the Carved Rock Fitness' customers let the support team know that the home page still has a link to Google+ in the footer. Let's create a new hotfix which branches directly off of our main production branch. Because of this, hotfixes typically have a version number as their identifier rather than specific feature or issue names. Let's go ahead and call this hotfix 1.1.1, since our last production release was 1.1. Passing 1.1.1 and hotfix start to Git flow will get us going. So this is going to be a pretty easy fix to make. We'll scroll to the bottom of the index file, and we can see that problematic line and delete it right away. So I'll add and commit the change and then push it to origin. Here again, we see another handy pull‑request link, so let's click on that. Notice it goes to the main production branch since that's where it branched from. We'll need a second pull request to merge into develop, so let's not delete the branch just yet. I'll go ahead and create and merge this now. And let's take a look at our commit tree on the Commits page. We can see quite a bit of activity on the Git tree, so let's break it down. The green line represents our main, or production branch, and we can see that the hotfix, the red dot, originates and ends back on the green line. We can also see that the develop branch is in blue and is hanging out on its own detached from the changes that the hotfix introduced. This is the second part of the Git flow hotfix finish command, merging the change down to develop. Let's create a pull request from our hotfix branch to the develop branch in order to fix this. We'll set the source to the hotfix branch and leave the destination branch as develop. Since we already emerged this hotfix into main, we can now delete this branch. And we'll go ahead and create and merge this pull request. Let's see how we're doing on this commit tree now. We now see that develop, represented in green, has the hotfix merged into it. The only thing left manually to do from the hotfix finish process is to tag the merge commit into main with the version of the hotfix, in this case, 1.1.1. We can do this directly on the commits page. Both Bitbucket and GitHub have the ability to add tags on the site and via the command line, but they're implemented a little differently. If you're using GitHub, I'm confident you'll be able to find a way to add tags on your own. For Bitbucket, however, I can just add the tag here for our 1.1.1 hotfix. We now need to clean up our local repository before we can continue working. First, we'll switch off of our hotfix branch to develop. Next, we'll pull the latest from origin, and we'll repeat this process for main. Then, let's delete our hotfix branch since it's been merged in. And finally, we'll check what branches we have left in our repository. It looks like we just have develop and main locally, so we're in good shape. With our hotfix taken care of, it's time to create that release to get our features out to production. We'll name our release 1.2, so we'll pass that along with release start to the Git flow command. We can see in the output that a new release branch based on develop was created and that we've switched to it. And next, we'll publish our branch so that we can create a pull request. We'll once again scroll up and use the handy PR generator link. And we can see that our PR is correctly populated from the release branch to the main branch. It didn't happen in this scenario, but it is common for teams to commit bug fixes directly into a release branch, so we need to make sure that we merge it back to develop. However, since this isn't the case in our scenario, we can just mark this branch for deletion and create it and merge it. Let's head back to our Commits page. Everything looks good except we still need to tag our release being merged into main. We'll add that and then head back to the Commits page. And there's quite a bit of activity on our commit tree, so I'd encourage you to trace the starting and ending point of each branch to help you understand their functions. And I'd also encourage you to get your local repository back into a good state as we did after the hotfix and do some more experimenting by yourself. Next, we'll talk about some of the situations and teams that are a good fit for Git flow.

### Understanding the Ideal Use Case

Let's take a look at what types of teams would benefit from Get Flow, and I'll be sure to highlight some unique team characteristics that would either strengthen or weaken the case for doing so. Carved Rock Fitness is a large outdoor retailer with a significant IT workforce, and it has several different engineering teams. Now I don't think that your organization is going to be exactly like Carve Rock Fitness, but my hope is that you'll be able to relate to some of the challenges that one or more of those teams faces, and I'd also encourage you to think about how your teams relate to or are different from Carved Rock Fitness. Let's take a look at each of the unique characteristics of the teams. The first and biggest team is E‑commerce, and this team's actually several smaller teams that all roll into one big team of over 100 engineers. So given the size of this team, there are many different experience levels to contend with. And even though they're split across different subteams, the scale is still pretty difficult, and it makes a lot of this work really hard. Another big challenge for this team is that their code base is a single massive monolith. And because this team deals with customer data and financial records, they have to follow SOC 2 compliance processes. And there's a lot of things that go into SOC 2, but the part that's most relevant for deciding a Git Workflow is that the code has to follow a really strict deployment process and have a lot of signoffs and code reviews and testing and traceability. Next, we have the innovations team, who focuses primarily on creating new experiences, and as a result, they don't have a ton of access to existing customer data. It's also a much smaller team at only five engineers, and they're a big proponent of pairing and mobbing. One more thing about this team is they're mostly senior engineers, which allows them to move really quickly. The team, fortunately, has their own code base, and it's not a part of the big monolith repository that E‑commerce has. And because their focus is on creating value quickly, it's a small and nimble team that's built a robust continuous integration continuous deployment pipeline to help them ship code quickly. And finally, we have the content management team, which is, as you may have guessed, responsible for Carved Rock Fitness's content‑management system. This team is a little bit bigger than innovation, but still really manageable at around 10 engineers. And it's more senior, but there are a couple of earlier and career team members as well. And fortunately, the content‑management system is in its own repository, separate from the other organizations, making it pretty manageable. The team, however, is distributed across the globe, with half of them on one continent and the other half on another. And there's multiple time zones in between them, and they only have a few hours a day where both of them are online and working at the same time. So now let's take a look at some of the key features of Git Flow and see how that matches to the needs of each of these teams. And as we're going through this, think about how you would answer some of these questions for the teams you're looking at. First things first, Git Flow is great for large teams because it reduces where and when engineers can bump into each other. And since only the E‑commerce team is really big at over 100 engineers, Git Flow's a perfect fit here. With the other two teams, it's probably a little overkill. And because the E‑commerce team also has several subteams, it's a great fit for Git Flow's feature process because each subteam can be working on one or more feature at a time. The innovations team really doesn't benefit from this functionality, every now and then, maybe. And the content management team has a limited number of content environments to test this in, so this feature isn't that helpful for them either. So remember how I mentioned that Git Flow is great for SOC 2 and similar audit processes? Well, that's a good fit for the E‑commerce team, and the other teams aren't, so for them, they don't really need this heavy process if they don't have to have it. So because we have all of these pull requests that slow down the release process, having a manual deployment pipeline isn't that big of a deal because it's already pretty slow without that. So it's great news for E‑commerce and content management since their deployment processes are already pretty manual. So especially when we have branch protections in place, it's pretty hard to accidentally merge bad code into production. So, of course, mistakes can happen and we, you know, might have something slipped through during a pull request, but accidentally pushing directly to main is something that's blocked, and this is great news for teams with less experienced engineers like E‑commerce and to some extent content management too. So the more concurrent development you have going on, the higher the change of merge conflicts becomes. And so because of this, the E‑commerce team is so big and has that monolith, it's really a big concern for them, and the other two teams with their smaller repositories and fewer engineers, it's less of an issue for them. And finally, as you saw during the demos, Git Flow has a pretty steep learning curve. And this isn't really great for any team, but because the E‑commerce team is so big, you have the ability to really create some experts who can kind of coach newer engineers or folks who are learning Git Flow for the first time and create some standardized training around it too.

### Teaching and Implementing Git Flow

As you've seen thus far, there's a lot to Git Flow, and it can be challenging to get people on board, let alone teaching them how to use it. So first, let's talk about advocating for Git Flow, of course, assuming that you've determined that Git Flow is the right approach for your team. I've broken down some key talking points by audience. So let's go through some common groups of people that you'll need to work with and address their potential concerns. The first and biggest group are the engineers themselves. And while the process itself will seem burdensome, many engineers have struggled through a painful line‑by‑line merge conflict resolution, and the thought of having to do that less often will surely be welcomed. Another significant concern around Git Flow is getting in the way when time is critical, specifically during production outages. But because of the hotfix functionality, we can get fixes to production fast and safely. Next, let's talk about engineering leaders. And one of the biggest concerns that they're going to have is being able to prevent bottlenecks with large numbers of engineers committing to the same repository. And as we've talked about, Git Flow is designed perfectly for this with its feature and release strategy, especially when there's multiple subteams. And many organizations will likely already have a designated person that handles some of the more complex parts of Git Flow, such as release scheduling or some of the trickier emerges. This role, often called a release manager, may already be somebody informally in your organization. If there's someone who's always running point on wrangling the repository, some formal structure in clarifying that process and their role in the organization could be really helpful. And finally, Git Flow significantly reduces a risk of their team failing an audit. And let me tell you, as someone who's had to sit across from auditors for hours at a time while they painstakingly go through your team's changes for the last year, I'll take anything that makes that experience better. And speaking of auditors, Git Flow naturally aligns with much of what they're looking for too. Most audit processes require traceability, specifically the ability to tie a code change from its initial request or ticket all the way to commit in production. And using Git Flow, teams can easily tie a request to a single feature and ensure that traceability. And additionally, because there are multiple steps and optionally pull requests required to get into production, the risks of an accidental or malicious change getting into production are greatly reduced. Either during or after your advocacy phase, you'll need to start thinking about how you want to teach Git Flow to your team. First, think about the ideal state of your repository as a Git Flow repository. What will be your development branch? What about your production branch? And how does that differ from your current configuration? Next, think about your organization's processes and practices. Are your stories the size of a feature that takes a team several days to complete? Or are they smaller so that you'd want to bundle several of them together into a single feature branch? Are your releases scheduled on a cadence or when you hit a critical mass of code changes to release? These are the answers that will be critical on how you communicate Git Flow and how it fits into the workflow your engineers are already familiar with. Once you've gathered that information, consider cloning the repository and setting it up to match the future state so people can practice in it. If you're too worried that it looks too similar to the actual repository, just create a dummy one instead. And finally, recognize that teaching this change is a large effort that will need to happen incrementally. It may be helpful to host sessions where engineers can perform the different tasks, and those are great opportunities for engineers to practice submitting feature pull requests to developing them and then rotating to the approver role for others. Now let's talk about how to implement Git Flow in addition to the implementation steps we already walked through earlier. There are a few technical items to get out of the way first. Specifically, will your organization require branch protection on your develop branch or just the production branch or both? Additionally, as we've discussed, different Git hosts have different levels of built‑in Git Flow Support. Confirm that what you're planning to do matches with what they offer. Next, merge and delete any outstanding branches. Ideally, you'd be able to start this process with a single branch remaining. And from that single branch, split off a branch and designate one of them production and the other one development. And finally, if you've determined that you need branch protection, go ahead and add those. You should now have a good foundation on how Git Flow works. Next, I'm going to talk about two recent variations of Git Flow that have come up in the last couple of years, GitHub Flow and GitLab Flow. If you're not sure about Git Flow, I'd highly recommend you check this clip out to see if either of these processes is a better fit for your needs.

### Git Flow Mutations (Github Flow and GitLab Flow)

As you've researched different Git workflow patterns, I'm sure you've come across a bunch of different variations, probably in particular to Git Flow. So, I'm going to briefly address two of them. First, let's talk about GitHub Flow. This approach only has two branches, specifically, the main branch, which always represents production, and next, a series of feature branches just like Git Flow, each containing a single feature. The difference, however, is that feature branches are just merged directly into main, and thus production. Now, if you're thinking this sounds an awful lot like feature branching, you're right. GitHub Flow is just feature branching with a different name, and because of this, I'm going to hold off on digging too deep into it because we're going to tackle that in the next module. So the next and more involved mutation is GitLab Flow, and one of the key features of this approach is that it easily supports the existence of a separate testing environment, such as staging or QA. Similar to Git Flow, it has multiple long‑living branches. The primary branch that engineers merge features into is the main branch, and this branch auto‑deploys to a shared development environment. The main branch is then periodically emerged into the testing branch, which automatically deploys into a staging or QA environment. And finally, the production branch deploys to the production environment whenever the testing branch is merged into it. However, unlike Git Flow, there are only short‑lived feature branches. Many engineers appreciate that there are no hot fix or release branches to track and deal with. I'm going to use another Git branch diagram to explain GitLab Flow in more detail. I'm going to assume that you've watched the first clip of this module to understand the different components of the diagram, and if you haven't checked that out yet and some of this terminology that I'm using on this diagram is confusing, I'd recommend going back and watching it. As I mentioned, the main branch is where engineers start new features from, and it's also where new features are merged directly back into it. Typically, main is set to automatically be deployed to a shared development environment for early integration testing. Next, often on a scheduled cadence, the main branch is pushed to a testing branch, which deploys to a staging environment. Once the test build has been validated, the commit is pushed to the production branch, and a production build is created and deployed. And then the cycle repeats again, starting with a feature branch from main. The features merged back into main and deployed to the shared development environment. Once the build looks good, it's merged into the testing branch, and finally, it's merged into production and deployed. In the scenario of hot fixes or similar pressing issues, the same process would be followed. The feature branch naming doesn't have any special significance in GitLab Flow. Instead of calling it feature‑a, you could just as easily have bug fix removed dead link, or something similar. Now that we understand how GitLab functions, let's run through some quick benefits. First, this approach is much simpler than Git Flow and has a much lower learning curve. However, some of the beneficial features like protecting production from accidental commits and allowing multiple testing cycles still remain. On the other hand, because there's less separation between code for a hot fix and a bug release going out the door and general feature work, the likelihood of merge conflicts is much higher. Similarly, we lose some of the traceability that Git Flow offered by reducing the visibility of which feature is in which release and as a result was deployed to production at what time. Let's recap what we learned about Git Flow. First, we broke down the process and dove into the different phases. We also covered the pros and cons of it. I also shared the Git Flow extension and how their commands can save time, but have their disadvantages in a poor request‑focused environment. Leveraging both source tree and the Git command line, we developed a number of features, hot fixes, and releases. And using our example company, Carved Rock Fitness, we exampled what types of teams and scenarios would be the best fit for Git Flow. Next, I shared some strategies on how to advocate for and ultimately implement Git Flow. And, finally, we wrapped up by discussing some common Git Flow mutations, specifically GitHub Flow and GitLab Flow. We'll take a look at feature branching next. This workflow seeks to find a balance between the speed of trunk‑based development and the traceability of Git Flow. It's a great workflow option for many different types of teams, so I hope you'll join me.

## Using Feature Branching

### Module Overview

Let's examine the feature branch in Git workflow. We'll cover when this approach would and wouldn't be useful. And I'll also highlight the types of engineering teams that would benefit from feature branching. Here's a quick overview of what we'll cover. First, I'll answer the most fundamental question. What is feature branching, and how does it work? I'll also provide some high‑level pros and cons of this approach to be aware of. Then, we'll explore how to implement this approach using Git from the command line. And next, we'll use our knowledge of feature branching to determine what type of team would benefit most from this workflow based on their unique characteristics. Finally, I'll highlight some of the key ways that you can advocate and eventually implement a feature branching workflow for your teams. Let's focus on some of the key characteristics of feature branching from a process standpoint. Just like most Git workflows, there's a long, well, actually permanently living main branch. And from this main branch, short‑lived temporary feature branches are split off and then merged back in. Now one of the main benefits of this approach is that it allows multiple features to be worked on at the same time. And as feature development continues, an additional benefit is that the main branch is always production‑ready. And with a solid continuous integration and continuous deployment process in place, you can also ensure that the main branch always matches the current state of production. While feature branching seems pretty straightforward, let's go through a quick branch diagram just so that we're on the same page. And now if you're not familiar with Git branch diagrams, they typically start on the left and then move to the right as time progresses. As I mentioned, the feature branching workflow always has the main branch, and commits are indicated by dots. And here I'm referencing a commit on the main branch. In addition, I'll add these nice little boxes to indicate when a deployment to production happens. I'm assuming a solid CI/CD pipeline to make this happen. So the team decides that they want to create a new feature, and they branch off of the latest commit on the main branch. And they're free to iterate on this branch as long as they need, adding as many additional commitments as they'd like. Now in the meantime, a different set of engineers wants to start working on a separate feature. They also add a number of commits to their feature branch. Now back on feature 1, the team has decided that their code is ready for production and merges it back into main via a pull request. And at this point, the feature branch can be deleted, and another production deployment happens. And the team working on feature 2 hears about the successful deployment of feature 1 to main and decides to preemptively emerge main back into their branch so that they can handle any potential merge conflicts on their own branch. Now you don't have to do this, but it definitely makes life easier. And finally, the second feature undergoes the same process as the first. It's merged to main, the feature branch is deleted, and a new production deployment happens. So now let's take a look at some of the pros and cons of feature branching.

### Feature Branching Pros and Cons

Having a good understanding of any workflow is critical, so let's take a look. Keep in mind that many of the points I'll highlight are situational. A pro for me might be a con for you and vice versa, and I encourage you to not inherently think as pros as good and cons as inherently bad, but think about how each statement might be perceived in your organization. So let's start with the positive aspects of feature branching. While this workflow isn't as simple as trunk‑based development where every commit goes directly to main, feature branching is pretty straightforward, especially compared to Git Flow. Feature branching also lives up to its name by supporting multiple features to be developed simultaneously. Additionally, since feature branches only merge into main, you only have to worry about merge conflicts at that single spot instead of multiple places. And by having a clear delineation on when features are started and finished, feature branching also offers traceability and some basic auditing requirements. And finally, each feature can be tested in isolation from other in‑progress features before being merged into main. Next, let's look at some of the cons of feature branching. And just like the pros, these are highly situational. So because every feature is a freestanding branch that could be merged into main at any moment, scaling becomes an issue pretty quickly. While it might be easy to track a handful of feature branches, even a 5x increase in those features would greatly complicate things. And as a result, by being restricted to a lower number of features, this process inherently restricts the numbers of contributors too. Because features can take a long time to complete, significant deviation can happen between the main branch and a feature branch, and failing to keep your branch up to date with this main branch can create huge merge headaches. And finally, this workflow is most effective when you have a smooth CI/CD pipeline. While you can technically get away with manual deploys after each main commit, it does reduce the power of feature branching quite a bit. Next, we'll work on a repository using feature branching using the Git command line.

### Demo: Using Feature Branching via the CLI

Now that we have a basic understanding of feature branching, let's try it out. We'll start by reviewing and forking a repository from Bitbucket. More on why I've chosen Bitbucket in a moment. Once we have the repository cloned locally, we'll create a feature branch, and on that feature branch we'll make a small change. Finally, we'll push our branch back to origin and create a pull request to merge it back to main. Once the code is merged in, we can delete our feature branch both on the remote server and locally. I'll be using Bitbucket for this module so that we can see the feature branches on Bitbucket's built‑in branch diagram. This is not an endorsement of Bitbucket over GitHub, and feature branching works equally well on either platform since the underlying infrastructure is just Git. If you're comfortable using Bitbucket, I'd encourage you to sign up for a free account; otherwise, I'll include some information on how to use GitHub as well. Okay, enough setup, let's get to work. I've created a short link to get you to the repository quickly. The code in this repository doesn't really matter and it's just a simple HTML page and some CSS, and it's really only purpose is for us to be able to track changes in Git. The real action happens here on the Commit tab. On the left, we can see a Git branch diagram similar to the one we used earlier. Instead of representing time from left to right, this diagram is going from bottom to top, meaning our most recent commit is at the top and the oldest one is at the bottom. There's already a separate feature branch here seen in red, and I'll talk a little bit more about that later. Let's go ahead and fork this repository. If you're using Bitbucket, you can use the forking functionality directly, and if you're using GitHub, you'll need to import the repository using the full URL of the Bitbucket repository. Once you have a copy of this repository, go ahead and clone it to your local environment. I've got my local copy of the repository open in Visual Studio Code, where we're going to work on a support ticket to remove a link to Google+ that's still in the footer. Before we make the code change, let's create a new feature branch from main. I want to call out something important here. While the workflow is called feature branching, it doesn't mean features in the traditional sense, and a feature branch can be anything, like in this case, a bug fix. With our branch ready, let's make that change. Towards the bottom of the index.html file, we can see the list element and we can just go ahead and delete it. Next, we can add and commit our change, and then I'll push it back to origin. Now let's head back to Bitbucket to see our changes. If you're using GitHub, you'll still be able to merge the changes in, but you won't be able to see the branch diagram. Let's head to the Commits tab now to see our newly added commit. There at the top of the page, you can see our new commit, but it's also still in the same line as the previous commit on the main branch, and that's because there haven't been any changes off of the main branch since the feature was branched off. So let's go ahead and change that by merging the feature branch that was in the repository when we forked it. I'll create a pull request to merge it in. We'll set the Source Branch as change‑sales‑banner, and set the branch to be deleted, and create that pull request. On the pull request, we can see that we're being warned that this branch is a commit behind "main", and we can solve this by checking the branch out locally, pulling main into it, and then pushing it back up to GitHub; or, we can just push the Sync now button, which will do the same thing all within Bitbucket. And we now see that there are three commits, with the latest being main getting merged into this branch. Now in a real situation you would ask somebody else to review your code and then merge your pull request in for you after it was thoroughly tested, but for this demo I'll just merge it myself. Now let's head back to the Commits page and see what our branch diagram looks like now. We see the red line representing the change‑sales‑banner, with a third commit coming from the main branch in green. And finally, we see that that branch is merged back into main. We also see our remove‑google‑plus branch is now represented in blue and hanging out there by itself, so let's fix that. Just like before, we'll create a pull request and delete the source branch when it's closed. And as expected, this branch is behind main, so we can sync the changes. You're also welcome to do this via the command line if you'd like. Fortunately, there aren't any merge conflicts and we can just merge the pull request. So let's take one final look at our Git branch diagram. We can now see that both of our branches have been merged into main, and even though they started at different times, both of our feature branches were easily and safely merged back into main with minimal issues. Next, I'll share some tips on how to advocate and implement feature branching if it's the right fit for your teams.

### Leveraging Feature Branching

I want to cover two things in this clip. First, we'll talk about what types of teams and situations are a good fit for feature branching, and second, we'll cover how you can use that rationale to help you advocate for and finally implement feature branching in your organization. We'll take a look at a fictional company, Carved Rock Fitness, a large outdoor retailer with a significant IT workforce, including several engineering teams. Now, I'm not saying that your company is the same as Carved Rock Fitness; it's not real, after all. However, I hope that you'll be able to find some parallels between the examples that I provide and what's happening in your organization. With that caveat in mind, here's a quick look at the engineering teams at Carved Rock. The first and biggest team is the e‑commerce team, and it's actually several smaller teams with over 100 engineers combined. Not only is this team big, but there's also a mix of senior and early career talent scattered throughout. Another challenge for this team is that their code base is a monolith, a single massive repository that contains all of their functionality. And since the team focuses on e‑commerce, they have to follow SOC2 compliance processes. I won't go into all of the details of SOC2, but the most relevant requirement is that the code must follow a strict development process with reviews, testing, and traceability. The innovations team has a completely different set of circumstances and focuses primarily on creating new customer experiences. The team's five engineers are senior and love pairing and mobbing. This team performs all of their work in their own code base away from the monolith, and they've had the opportunity to build their own CI/CD pipeline, which allows them to ship code more quickly. Now the final team we'll look at is the content management team, and this team is more manageable, it's got about 10 engineers, and the skill level is mostly senior, but there's some junior engineers, too. The code for this team is in its own repository, but there is one distinguishing factor for the content management team: it's globally distributed. The team is split across continents, and there are only a few hours of overlap between the whole team. So, armed with this knowledge, let's take a look at how the key features of feature branching align with their needs. This is a great time for you to think about how you would answer these questions for your team, too. So, feature branching is a very straightforward workflow, which makes it a great fit for both the innovations and content management teams. Unfortunately, it's too simple for the e‑commerce team and would quickly struggle to match the scale of their work. Unlike trunk‑based development, feature branching allows teams to work in multiple features at once, which is a must for the e‑commerce team. Since the innovation team is so small and they like to mob program, there isn't much need for this functionality, but it is a great fit for the content management team, since they can work on different features in different time zones. Unfortunately, the e‑commerce team doesn't have a strong CI/CD pipeline, which reduces the effectiveness of feature branching. Innovations pipeline would make a great fit, though, and the content management team has somewhat of a CI/CD pipeline, but there's still some manual steps, which makes this manageable, but far from ideal. As a part of the simplicity that feature branching offers, there can only be one release plan at a time, specifically, the most recent commit to the main branch. This approach doesn't work for a large team like e‑commerce, but it is an ideal fit for innovations and content management. Feature branching is most successful when the number of features concurrently in development is lower. One of the easiest ways to do this is reduce the number of engineers contributing to it. The e‑commerce team is way too big for feature branching, but it's a good fit for the size of the innovation and content management teams. While mob programming is great in theory, it can be hard on large audit‑restricted teams, like e‑commerce. And similarly, the need to do separate branches and pull requests on a pure pair programming and mobbing team, like innovation, may seem like an impediment. On a team that likes to mix work styles like content management, however, it's a great option. And finally, while feature branching offers some traceability, it is insufficient for the e‑commerce team's needs Similarly, the overhead needed to create traceability can be seen as too high of a burden for the innovations team. The balance of speed and traceability does meet the content management team's needs. In particular, traceability can be a good support system for teams that work asynchronously, as globally distributed teams often do. Based on what we've covered about feature branching so far, let's say that you've decided that it's the right fit for your team. Now, how would you go about convincing others of that? Regardless of your current workflow, the balance of speed and traceability is appealing to large numbers of engineers. And it doesn't matter if your team relies on solo or pairing and mobbing work, or a combination of the two. Feature branching remains a good fit for any and all development approaches. Speed to production is always a priority for teams, and nothing is worse than having a feature that's code‑complete and tested, but waiting for a process to complete before it can go live. Fortunately, feature branching provides an opportunity to safely test and then quickly go to prod. And finally, unlike trunk‑based development, your team can work on multiple features at a time. This allows a larger team size, as engineers can split into smaller groups to work on separate features. Now, before we move on, give some thought on other ways that you might be able to advocate for feature branching in your organization. As much as possible, it's best to transition Git workflows at a point of relative stability for the repository. First, let's make sure that much of the existing work and branches has been completed. Next, have conversations with engineers on how to use the new workflow. These discussions will depend heavily on what workflow you're migrating from. And if you haven't already, it might be helpful to add restrictions to the main branch to prevent accidental direct merges. And finally, remove any unnecessary branches, particularly if you're coming from Git Flow. You can delete all of your long‑living branches, except main, of course.