EBOOK

Mastering Git with Matillion Data Productivity Cloud



matillion

Summary

Git provides a safety net for developers, allowing them to experiment without fear of irreversibly damaging their or their collaborators' work. With Git, mistakes are not the end of the road—they're merely detours on the path to improvement. In this ebook, "Mastering Git with Matillion Data Productivity Cloud," we'll cover every aspect of Git, from the basics to advanced techniques. Whether you're a novice developer keen to learn version control or a seasoned pro looking to deepen your understanding, this book will turn you into a Git expert in no time. By the end of this book, you'll be able to unlock the full potential of version control and take your pipeline development game to the next level.

Contents

01 Commit, Push & Pull

02 Branching

03 Common Branching Strategies

04 Merging

05 Git Merge Strategies

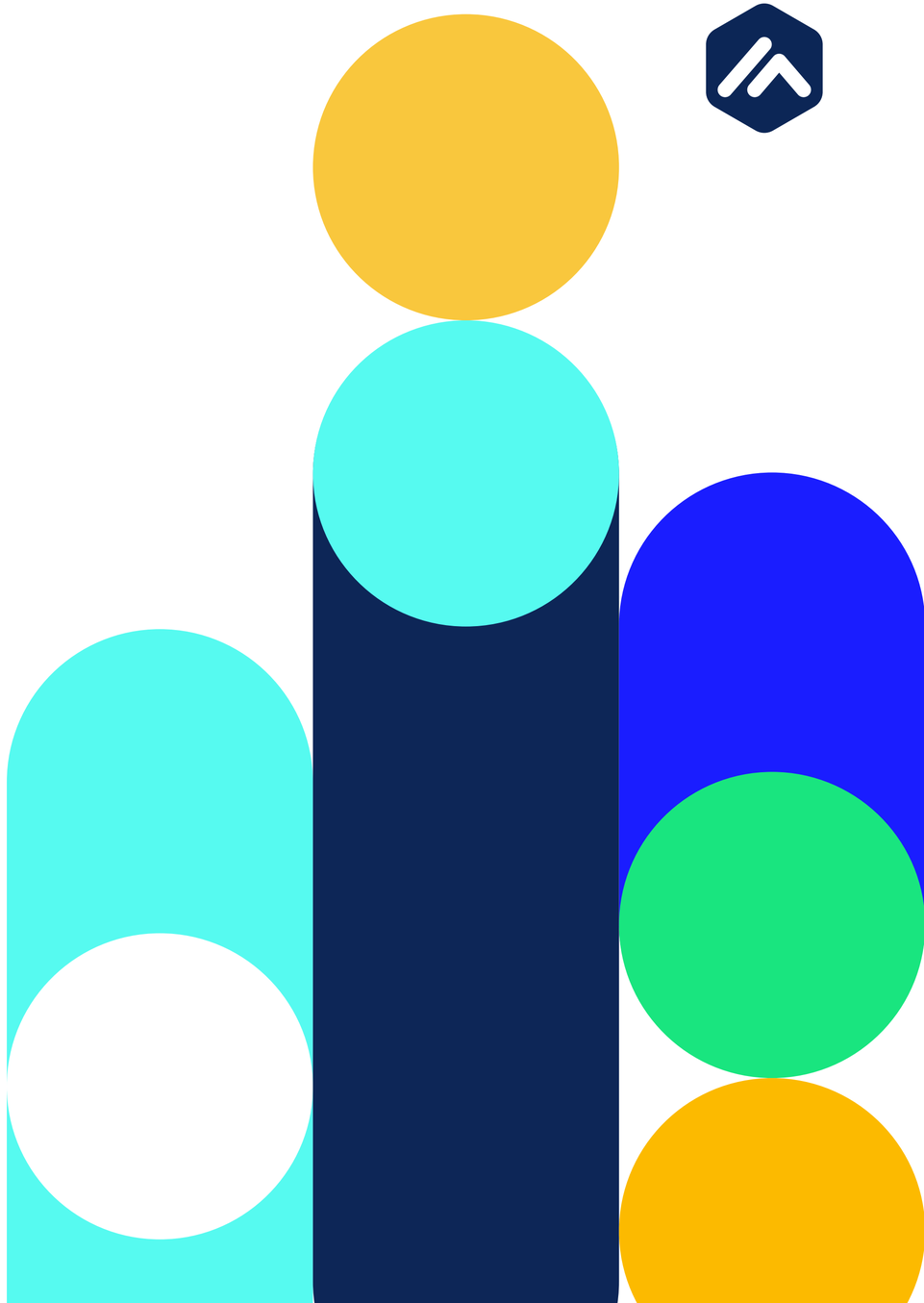
06 Hard Reset

07 Chapter 7: Unpushed Commits

08 Chapter 8: Uncommitted Changes

09 Chapter 9: Revert

10 Conclusion



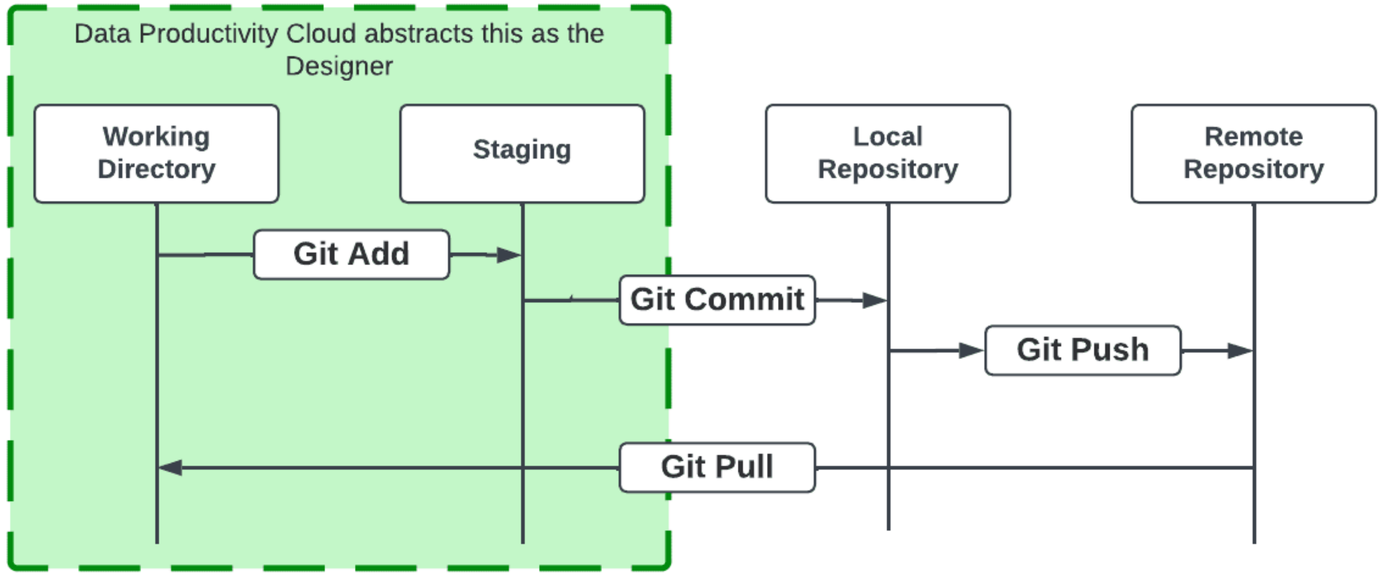


01

Commit, push & pull

Imagine Git as your team of assistants, orchestrating the flow of changes in your codebase, from the initial crafting in the Working Directory to the collaborative integration via add, commit, push, and pull actions. This ensures seamless collaboration and preservation of your project's evolution across all stages of development.

Mastering Git's different areas empowers you to streamline your workflow, collaborate effectively, and manage projects with precision. In this chapter, we'll explore the concepts of **working directory, staging, local,** and **remote** workspaces and unravel the mysteries behind the common git actions: **add, commit, push,** and **pull.**



Working directory

Your **Working Directory** serves as your personal workspace, away from Git, where you craft and refine your code in solitude. In the Data Productivity Cloud, this is the Designer session in your web browser, where your changes are only visible to you.

Staging and add

The **Staging** area is your first assistant, helping you organize and prepare your changes before you commit to them. When you modify your files in the Working Directory, Git allows you to selectively **add** (and remove) these changes to the staging area. Anything added to the staging area is included in the upcoming commit to your local repository.

Note: The Data Productivity Cloud handles this for you. **All changes made in the Working Directory are automatically added to the Staging area, so there is no need to use the 'add' functionality in Matillion's Data Productivity Cloud.**

Local repository and commit

When you're happy with a set of changes, Git's **Local Repository** area steps in as your next assistant, storing your changes for later reference—like a personal archive. **Committing** changes in Git is akin to saving your progress in a labeled folder (with a commit message) **that only you can access.**

Each commit encapsulates a snapshot of your project at a specific point in time, preserving the evolution of your codebase. The **Local Repository** enables you to track changes, revert to previous states, and collaborate with confidence.

Remote repository and push

But what if you're collaborating with a team and want them to see your changes? Enter the **remote repository**, your final assistant. **Pushing** changes to the remote repository is akin to moving your commits from the labeled folders in your local area to a central server, ensuring that every team member has access to the changes you're happy with.

Pull

Finally, we have the **pull** action. **Pulling** changes from the remote repository is akin to synchronizing your local copy of the project with the collective efforts of your teammates. This exchange of changes enhances collaboration and ensures cohesion within your development team.

Of course, there may be occasions when your teammates have changed things that clash with your changes. For more information on this, explore Chapter 4, which covers **Merging**.



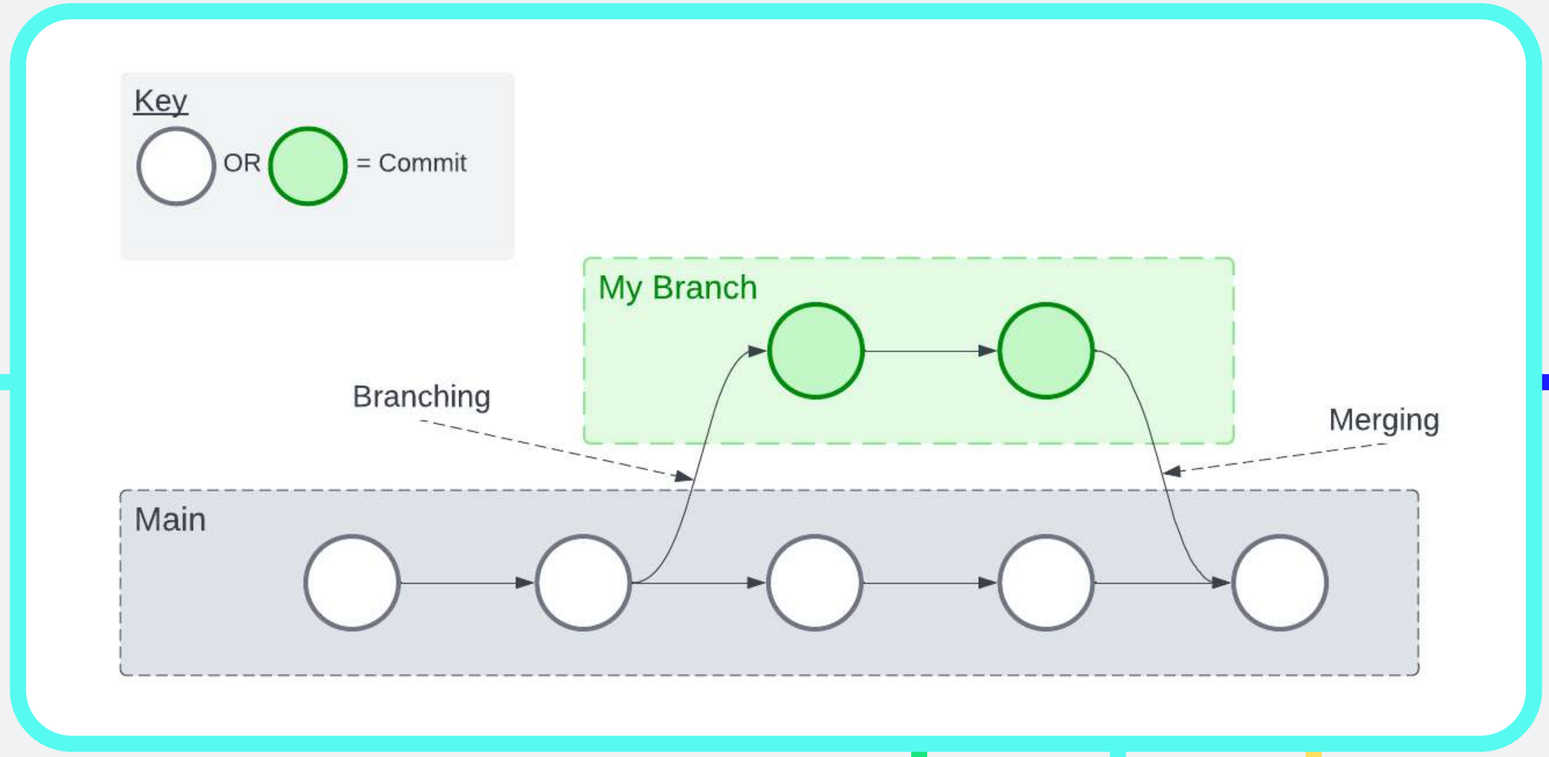


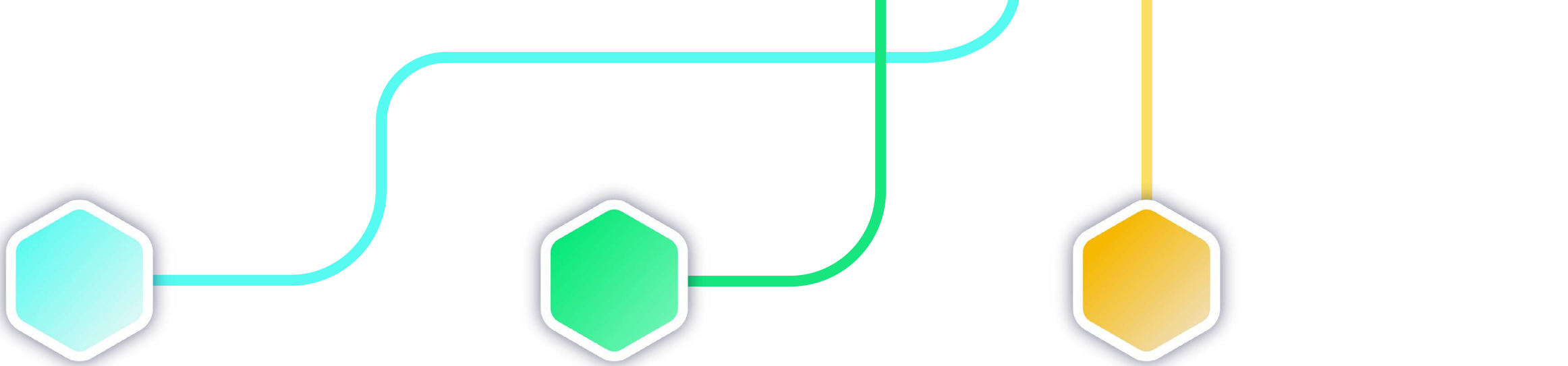
02

Branching

Now that we've covered the working directory, staging, local, and remote workspaces, along with common Git actions like add, commit, push, and pull, it's time to dive into another essential aspect of Git: **branching**.

Branching in Git empowers developers to explore new ideas, experiment with features, and collaborate seamlessly with team members. By embracing the branching model, you can navigate the complexities of software development with confidence, safety, and agility.





Branching, safety, and isolation

Just as a tree branches out into different paths from its trunk (in Git, the main branch is sometimes referred to as “trunk”), **Git allows you to create branches in your project’s development timeline.** Creating a branch, aptly named “**branching**,” in Git provides a separate space for you to innovate and iterate without disrupting the stability of the main project.

And fear not—if you make a mistake or encounter obstacles along the way, Git’s commit history allows you to backtrack and explore alternative routes with ease.

Note: Backtracking through previous versions is currently supported externally in the Data Productivity Cloud using our “Connect your GitHub Repository” feature.

Collaborating

Branching enhances flexibility and collaboration within your team. By creating feature branches, **each developer can work independently** on their assigned tasks without stepping on each other’s toes, promoting productivity and accelerating the pace of innovation.

Merging complements branching in Git’s version control workflow. When it’s time to integrate your changes back into the main codebase, Git’s **merging capabilities allow you to incorporate your branch’s changes into the project** safely and easily.

In Chapter 4, we’ll discuss different approaches to the merging process, ensuring smooth transitions between branches and preserving the integrity of your project’s history.

Commits on branches

Each branch maintains its own isolated history of commits, allowing developers to work on distinct features or fixes without interfering with the main codebase. When commits are made on a branch, they remain confined to that branch until it’s merged back into another branch, such as the main branch.

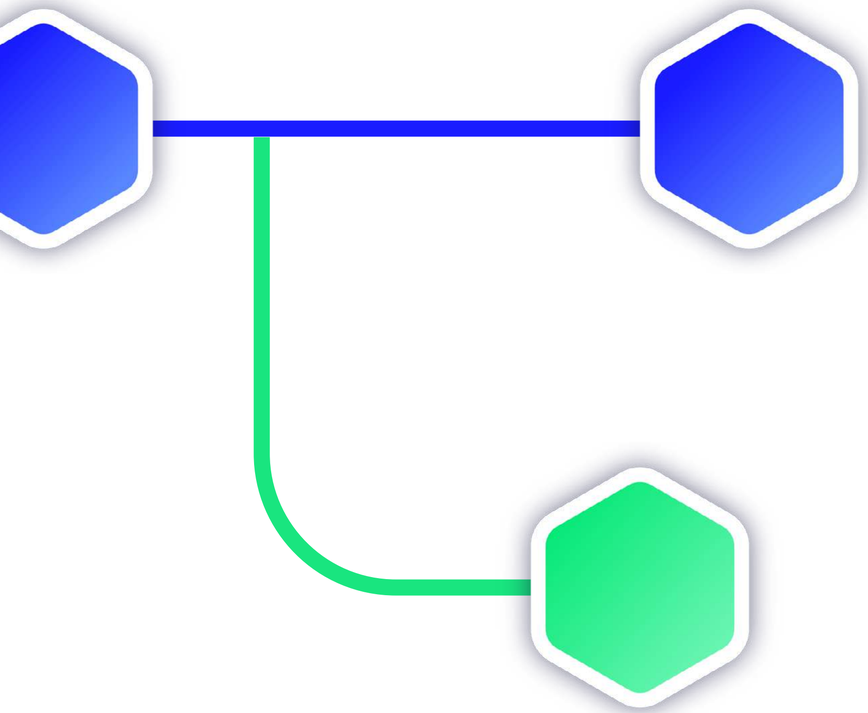
This ensures that **changes are contained within their respective branches until they’re ready to be integrated into the larger project.** The branching and merging workflow facilitates a structured approach to development, enabling teams to collaborate efficiently and manage changes with precision.



03

Common branching strategies

Whether working on a small-scale project or a large-scale enterprise application, selecting a suitable branching strategy is essential for maintaining code quality and project stability. This chapter covers best practices for managing branches effectively in different project scenarios.



Git-Flow: a structured approach

Git-Flow is a branching model designed to streamline the development workflow. This strategy revolves around two main branches: **main** (or master) and **develop** (or dev). The main branch serves as the stable production-ready codebase, while the **develop** branch acts as the staging area for new features.

Engineers create **feature branches** off the develop branch, where they implement and test new functionality. Once features are complete and engineers are happy with their implementation, the feature branch is merged back into the develop branch for integration testing.

Periodically, releases are branched off from the develop branch for final testing and deployment then merged back into the main branch once this process is complete. Git-Flow provides a structured framework for collaboration, ensuring a smooth and organised development process.

This is particularly useful for teams working asynchronously (e.g., teams spread across time zones or a large, distributed team that can't always discuss changes together).

Trunk based development: embracing simplicity

Trunk Based Development takes a minimalist approach to branching, using a single, shared trunk or **main branch**. All development, including new features, bug fixes, and experiments, is done directly on the trunk branch. This streamlined approach encourages continuous integration and frequent deployments, reducing the overhead associated with managing multiple branches.

While Trunk Based Development may lack the formal structure of Git-Flow, its simplicity promotes agility and rapid iteration, making it ideal for projects with fast-paced development cycles and small teams working synchronously.

Other popular strategies

In addition to Git-Flow and Trunk Based Development, several other branching strategies are commonly used in Git workflows, including:

Feature Branching: Where each feature or task is developed on its own branch before being merged into the main codebase, providing a flexible and scalable approach for teams working on diverse sets of features.

Release Branching: Similar to Git-Flow, it involves creating a separate branch for each release to facilitate final testing and bug fixes.

Selecting your strategy

Selecting the right branching strategy for your team is crucial for effectively managing codebases and facilitating collaboration within development teams. Whether you prefer the structured approach of Git-Flow, the simplicity of Trunk Based Development, or a customised hybrid model, Git offers a versatile toolkit for navigating the complexities of version control.

By understanding and implementing common branching strategies, engineers can optimise workflows, accelerate development cycles, and deliver high-quality software with confidence.



04

Merging

Merging is a fundamental process in Git that facilitates collaboration and teamwork. By understanding the different merging strategies and knowing how to handle common challenges, you can ensure smooth integration and a cohesive codebase.

In this chapter, we'll examine what merging is, why it's essential for distributed teams, the different strategies used in merging, and some common pitfalls and solutions when things don't go as planned.

What is merging in Git?

Merging in Git is the process of integrating changes from one branch into another. This could involve merging a feature branch into the main codebase, a release branch back into development, or even resolving changes from a collaborator's branch.

Merging is the backbone of collaboration in Git, allowing teams to work on separate tasks and combine their work into a cohesive whole.

Collaborating in distributed teams

For distributed teams, where developers might be working across different time zones or geographic locations, merging is indispensable. It allows team members to work independently in their own branches, developing features, fixing bugs, or experimenting with new ideas.

When it's time to bring these changes together, merging makes it possible to integrate work from multiple sources into a single branch, ensuring that everyone's contributions are included without overwriting or losing valuable code.

Merging issues and how to fix them

Merging can sometimes lead to conflicts, where changes in different branches are incompatible. This can happen when two developers edit the same part of a file or when changes from one branch impact the work in another. Here are some common problems and ways to resolve them.

Merge conflicts

If Git detects conflicting changes, it will prompt you to resolve them manually. Tools like Git's built-in conflict resolution or third-party merge tools can help. Once resolved, you can commit the changes and complete the merge.

In the Data Productivity Cloud, we currently offer conflict resolution on a file-by-file basis, allowing the user to select whether they want to keep the changes from their branch or the branch they are merging into.





05

Git
merge strategies

In this chapter, we dive into a more nuanced topic: Git merge strategies. While the default merge strategy typically works well, understanding different strategies can be invaluable in specific scenarios.

We'll explore the reasons behind selecting a merge strategy, the popular options available, and when you might want to deviate from the default.

Why select a merge strategy?

When you merge branches in Git, the merge strategy determines how changes from different branches are combined. Most of the time, the default merge strategy, `ort` (formerly `recursive`), is more than sufficient for handling merges in Git. It effectively combines changes and resolves conflicts for the majority of cases you'll encounter.

However, there are specific scenarios where you might encounter issues or need more control over the merge process. In these situations, considering an alternative merge strategy can provide a tailored solution to address unique challenges or requirements in your development workflow.

Understanding these strategies and their use cases ensures you're equipped to handle any merging complexities that arise. Selecting a merge strategy can help with the following:

- **Conflict Resolution:** Different strategies provide various ways to handle conflicts, making it easier to resolve complex merges.
- **History Management:** Some strategies influence how the commit history is structured, which can be important for maintaining a clear and understandable project history.
- **Specific Use Cases:** Certain strategies are designed for particular scenarios, such as merging multiple branches simultaneously or incorporating changes from external repositories.

Handling merge conflicts

Merge conflicts are an inevitable part of collaborative development, occurring when changes in different branches overlap or contradict each other. Understanding how merge conflicts arise and how different merge strategies handle them is essential for maintaining a smooth workflow and ensuring the integrity of your codebase.

How merge conflicts arise

Merge conflicts typically occur in the following situations:

- **Simultaneous Changes:** Two or more developers make changes to the same line of code or the same file in different branches.
- **Conflicting Modifications:** Changes in one branch contradict changes in another branch, leading to uncertainty about which version should be retained.
- **Reordering:** Changes in the order of lines or files can also lead to conflicts, especially in closely related code.

When a conflict arises, Git may be unable to merge the changes automatically. This can result in requiring manual intervention, where the user is prompted to perform an interactive merge, to resolve the discrepancies. Different merge strategies can result in a different set of conflicts, with some strategies able to resolve scenarios more effectively than others.

Popular merge strategies

ort (“Ostensibly Recursive’s Twin”—Formerly “Recursive”)

The `ort` (or `recursive`) strategy is the default in Git and is used in the Data Productivity Cloud’s merge command. It uses a 3-way merge algorithm, merging changes by finding a

common base commit and creating a new commit that combines the new commits from both branches.

It is the default merge strategy as it usually results in the fewest merge conflicts without causing mismerges. It can also detect and handle file renames. Options like `ours` and `theirs` within this strategy can help resolve conflicts by favouring changes from one branch over the other:

ours: This option resolves conflicts by favouring changes from the current branch.

theirs: This option resolves conflicts by favouring changes from the branch being merged in.

ignore-all-space: Ignores whitespace differences when comparing lines, reducing conflicts caused by formatting changes.

ignore-space-change: This option treats sequences of one or more whitespace characters as equivalent, minimising conflicts due to minor spacing variations.

resolve

The `resolve` strategy is simpler and faster than `ort`, but it only works with two branches and is less effective at handling complex conflicts. It’s useful for straightforward merges where changes don’t overlap much. It does not handle renames. This is best for large but straightforward merges.

octopus

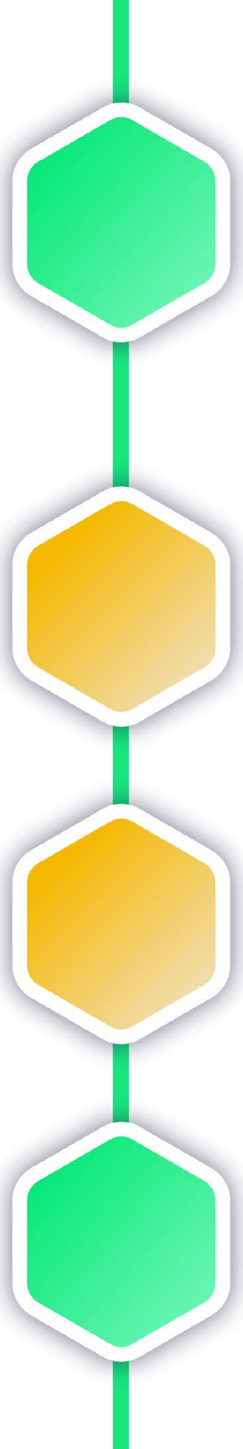
The `octopus` strategy is used when merging more than two branches. It’s designed for simple cases where conflicts are unlikely. It handles conflicts by merging non-conflicting changes and leaving conflicting changes for manual resolution. It’s best used for integrating multiple topic branches with minimal conflicts and is the default when merging more than one branch.

ours

The `ours` strategy is a bit of a misnomer. It records the merge but discards the changes from the branches being merged in, keeping only the changes from the current branch. This can be useful for merges that are about superseding old development histories of side branches while acknowledging their existence. It differs distinctly from the `ours` option for the `recursive` merge strategy.

subtree

The `subtree` strategy is used when you want to merge in a subproject and retain its history as a subdirectory of your project. This is useful for maintaining projects that include other projects as dependencies. It handles conflicts by treating the subproject as a distinct entity, preserving its changes within its designated subdirectory.



When to deviate from the default strategy

While the `ort` strategy works well in most cases, there are scenarios where other strategies might be more appropriate:

- **Complex Merges:** When dealing with a complex, multiple-branch merge, the `octopus` strategy can simplify the process.
- **Maintaining Separate Histories:** Use the `subtree` strategy when you need to maintain a clear history of a subproject within your main project.
- **Specialised Conflict Resolution:** The `ours` or `theirs` options within the `ort` strategy can be useful when you need to resolve conflicts by choosing one branch's changes over the other's.

Merging within the Data Productivity Cloud

Merges are performed using the recursive/ort strategy within the Data Productivity Cloud, resulting in few conflicts. Where conflicts do occur, the user will be advised of the files that have conflicts and may choose the file from either ours or theirs, accepting all changes from that file.

Matillion's Git merging strategy

In the Data Productivity Cloud, we use a **custom three-way** merge strategy. It is a method for merging two divergent branches into a single branch, used when two branches have developed separately from a common ancestor and changes from both branches need to be reconciled to produce a unified result.

Three-way merges are robust because they leverage the common ancestor to understand the context of changes, allowing for more informed merging and easier conflict resolution when necessary.

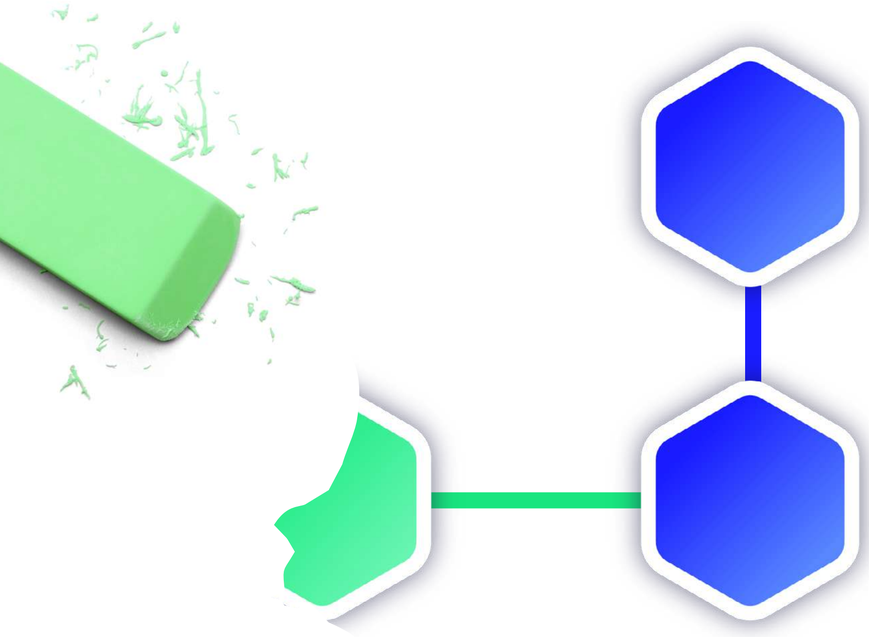


06

Hard reset

A hard reset is a powerful Git feature that allows you to revert to a specific commit, providing a way to correct mistakes or start fresh. However, it comes with risks, and caution should be exercised when using it.

By following best practices and understanding the implications, you can use hard reset to maintain control over your project's development. In this chapter, we'll discuss what a hard reset is, when it can be useful, and the precautions you should take when using it.



What is a hard reset in Git?

A hard reset in Git is an operation that allows you to return your local branch, including commit history, to a specific commit. The Data Productivity Cloud offers the two most common variations: resetting to the latest **local** commit (which simply drops uncommitted changes), and resetting to the latest **remote** commit.

This operation erases all changes made since that commit, effectively reverting your branch to an earlier state. It can be a lifesaver when you need to undo a series of changes quickly or when you want to experiment without worrying about affecting the branch you're on.

When is hard reset useful?

Hard reset can be useful in various situations, particularly when you want to start fresh from a known good point. If you've been working on a feature and decide to remove local changes or revert to a previous, committed state, a hard reset can help you return to this known state without committing unnecessary changes.

An example might be working experimentally in the main branch to find out how to fix a bug. Once you know the solution, you need to revert everything back to how it was in production, ready to push the fix cleanly through development.

It's important to note that a hard reset in the Data Productivity Cloud returns the project to the last committed state, which may differ from the last pushed state.

This distinction is critical because, even after you hard reset, the local version of your branch will differ from the remote version if you have commits that are yet to be pushed or if there are commits upstream that are yet to be pulled.

The dangers of hard reset

While hard reset can be useful, it also comes with risks. Here's what you should be aware of before using it:

- **Loss of Uncommitted Work:** If you have uncommitted changes in your working directory or staging area, a hard reset will erase them.
- **Irreversible Action:** Once a hard reset is performed, you cannot undo it. There is no "undo" button, so make sure you're certain before proceeding.

Best practices for hard reset

To use hard reset safely, consider these best practices:

- **Commit Important Work:** Before performing a hard reset, ensure you have committed critical changes.
- **Use with Caution:** Only use hard reset when you are confident it's the correct action. Double-check the commit you want to reset to and ensure no essential work will be lost. Again, this is not relevant in the Data Productivity Cloud since we only reset to the most recent commit.
- **Communicate with Your Team:** If you're working in a team, communicate your intentions to avoid disrupting others' work. Of course, communicating clearly is always good practice when using version control.



07

Unpushed commits

Unpushed commits are one of the more nuanced aspects of Git. Understanding unpushed commits is helpful for maintaining a smooth and error-free development workflow. By regularly checking your unpushed commits, you can review changes, assess the impact of your push, and correct errors before they reach the remote repository.

This section dives into why you might want to view unpushed commits, how to retrieve this information both in the Data Productivity Cloud and using Git commands, and some useful alternatives to keep you informed about your unpushed commit history.

Why view unpushed commits?

In a collaborative development environment, understanding the state of your local repository compared to the remote repository is crucial. Unpushed commits refer to commits present in your local branch but not yet shared with the remote repository. Viewing unpushed commits is important for many reasons, including:

- **Impact Assessment:** By examining unpushed commits, you can understand the potential impact your push action will have on the remote repository. This is particularly important when working on shared branches, as your changes might affect other team members' work.
- **Error Correction:** Viewing unpushed commits helps spot mistakes or unnecessary commits before they reach the remote repository, giving you the opportunity to amend changes with new commits.

Viewing your unpushed commits acts as a local, high-level review of sorts, allowing you to check that you are about to share the work you expect to be sharing without needing to go back over all the code or pipeline changes you have made.

The messages you write for each commit are important because they provide a high-level description of the changes you've made, and they are what you'll be reading when reviewing your unpushed commits.





08

Uncommitted changes

Keeping track of uncommitted changes is a critical part of the Git workflow. By regularly reviewing your modifications before committing, you can ensure that only the intended changes are included, maintain a clean commit history, and catch errors early.

Using commands like **git diff, git status**, and their various options, you can gain detailed insights into the state of your working directory and make informed decisions about your next steps.

In this chapter, we'll discuss why it's important to keep track of uncommitted changes, how to view these changes using Git commands, and some useful alternatives and options for getting detailed information about your uncommitted code additions, modifications, and deletions.

Why view uncommitted changes?

In a development environment, keeping track of uncommitted changes is essential.

- **Review Before Committing:** Before you commit your changes, it's important to review what you've modified. You can spot mistakes or unintended modifications early, reducing the risk of introducing bugs or regressions into your codebase. This ensures that only the intended changes are included in the commit, helping maintain a clean and accurate commit history.
- **Selective Staging:** Viewing uncommitted changes allows you to decide which changes should be staged for the next commit and which should be discarded or modified further. Selective staging for commit is a feature that is coming soon to the Data Productivity Cloud.

How to retrieve uncommitted changes

Git provides several commands to help you view and manage uncommitted changes. One of the most common commands is `git diff`, which shows the differences between the working directory and the staging area (or a specific commit). To see uncommitted changes, you can use:

git diff

This command will display a line-by-line comparison of your modified files, highlighting additions and deletions. It's an invaluable tool for understanding the current state of your working directory.

Alternatives and options for Git diff

While git diff provides a comprehensive view of uncommitted changes, there are several alternatives and options to customise its output and make it more useful for specific tasks:

- **git diff --color-words:** This option highlights changes at the word level rather than the line level, making it easier to spot small modifications within a line of code.
- **git diff --name-only:** If you only want to see which files have been modified without viewing the detailed changes, use this option. It provides a concise list of changed files.
- **git diff --staged:** To see the differences between the staged changes and the last commit, use this command. It's useful for reviewing what you have added to the staging area before committing.
- **git status:** While not a diff command, git status provides a summary of your working directory and staging area, showing modified, added, and deleted files. It's a quick way to get an overview of uncommitted changes.
- **git diff HEAD:** To compare your working directory changes against the latest commit, use this command. It includes both staged and unstaged changes in the output.



09

Revert

Understanding how to undo changes safely and effectively is crucial for maintaining a clean project history and ensuring your codebase remains stable. In this chapter, we'll take a closer look at the "Git Revert" command, explore the reasons behind undoing changes, look at some of the methods available, and the nuances of reverting both locally and remotely.

Additionally, we'll discuss the differences between **git revert** and **git reset** to give you a comprehensive understanding of these powerful tools.

Why revert changes?

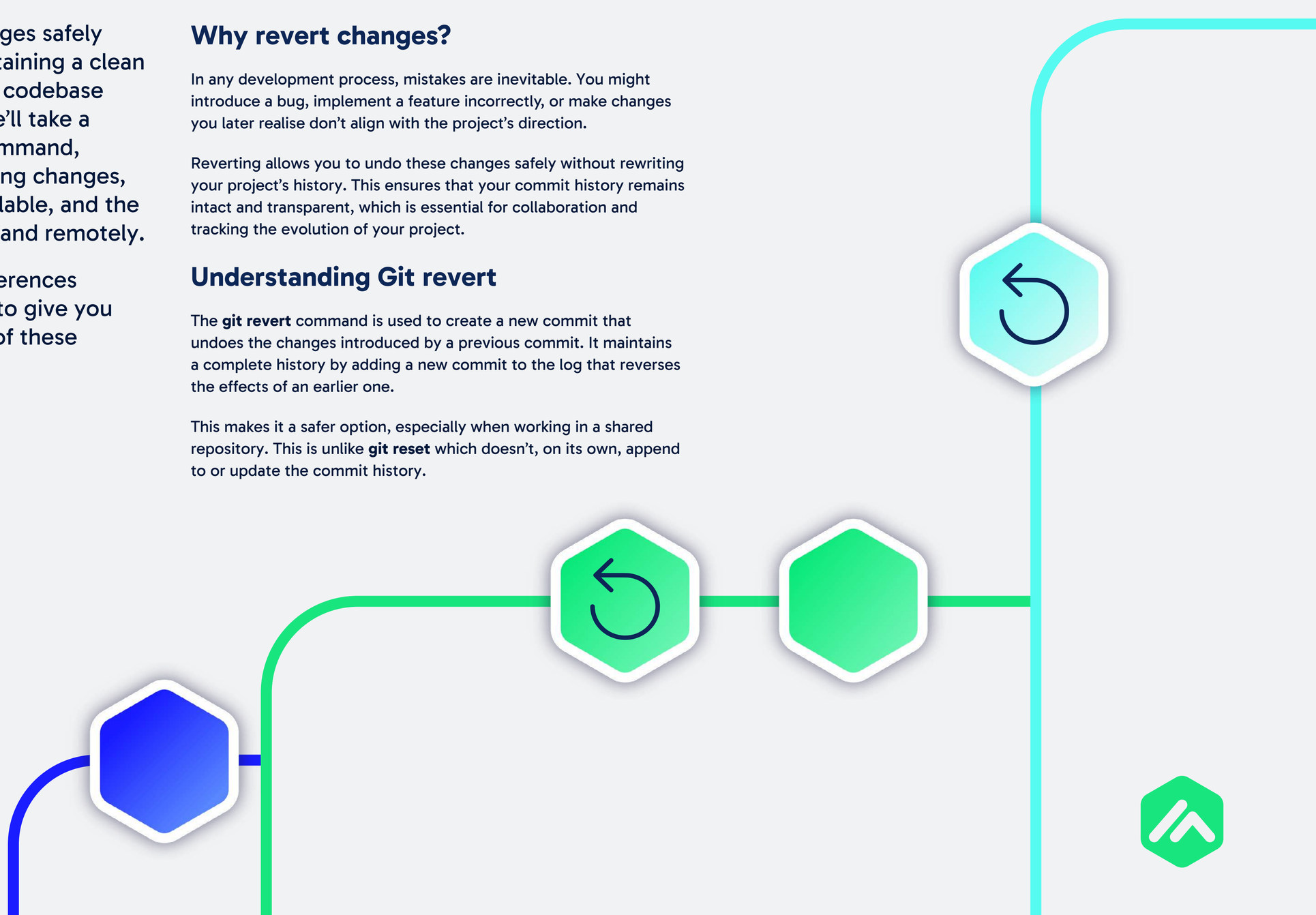
In any development process, mistakes are inevitable. You might introduce a bug, implement a feature incorrectly, or make changes you later realise don't align with the project's direction.

Reverting allows you to undo these changes safely without rewriting your project's history. This ensures that your commit history remains intact and transparent, which is essential for collaboration and tracking the evolution of your project.

Understanding Git revert

The **git revert** command is used to create a new commit that undoes the changes introduced by a previous commit. It maintains a complete history by adding a new commit to the log that reverses the effects of an earlier one.

This makes it a safer option, especially when working in a shared repository. This is unlike **git reset** which doesn't, on its own, append to or update the commit history.



How to use Git revert

To revert a specific commit, you can use the following command:

git revert <commit-hash>

This command creates a new commit that undoes the changes made in the specified commit. If the commit you're reverting has already been pushed to a remote repository, you can push the new revert commit as well:

git push origin <branch-name>

Reverting a commit is particularly useful when you need to undo changes in a collaborative environment, as it preserves the integrity and history of the repository.

This works whether the commit you are reverting has already been pushed to remote or not.

Differences between Git revert and Git reset

While both **git revert** and **git reset** can be used to undo changes, they serve different purposes and have distinct effects on the commit history.

- **Git Revert:** Creates a new commit that undoes the changes introduced by a previous commit. It preserves the commit history, making it a safe choice for undoing changes in a shared repository.
- **Git Reset:** Moves the current branch to a specified commit, effectively undoing all of the changes since that commit. It can modify the commit history, which can be useful for removing unwanted commits, which changes the record of that branch of the repository. It is highly recommended to avoid using this approach if the changes have already been pushed to a shared repository—its advised use is for undoing commits that are yet to be pushed to remote.

How to use Git reset

The **git reset** command comes with different options, each serving a different purpose:

- **Soft Reset:** Moves the branch pointer to a specified commit without changing the working directory or the staging area. The changes remain staged:

  git reset --soft <commit-hash>

- **Mixed Reset:** Moves the branch pointer to a specified commit and updates the staging area to match but leaves the working directory unchanged:

  git reset --mixed <commit-hash>

- **Hard Reset:** Moves the branch pointer to a specified commit and updates both the staging area and the working directory to match. This discards all changes after the reset point:

  git reset --hard <commit-hash>

Note: In the Data Productivity Cloud, the hard reset option is the one that is currently available. The reset, in this context, moves the branch pointer to the last commit (i.e., it removes all uncommitted changes from the current working directory).



Conclusion

Mastering Git enables developers to take advantage of version control to collaborate more effectively while safely exploring new ideas. This ebook provided a thorough overview of Git, including basic commands and advanced data management strategies, while covering how Git is handled in the Matillion Data Productivity Cloud.

Chapter One focused on Git workspaces, including working directory, local, staging, and remote. We covered when it's best to use each workspace and introduced basic common Git actions, such as add, commit, push, and pull. These fundamental Git concepts enable developers to be more effective in managing projects and collaborating with others.

Chapter Two dove deeper into branching, which provides developers with a way to collaborate safely, try new features, and work on new ideas without impacting the main codebase. We demonstrated how branching can help each developer work independently and covered how comments can be used to track changes.

Chapter Three covered branching strategies. We outlined common branching methods, and took a closer look at two of the most common options: Git-Flow and Trunk Based Development. By looking at the strengths of each strategy, we provided a framework to help developers choose the best option for them.

Chapter Four shifted the focus to merging. We defined merging in Git and illustrated how this function is used to facilitate collaboration by bringing together the independent work that happens with branching. We also provided developers with a roadmap of common issues and demonstrated how to fix them in the Data Productivity Cloud.

Chapter Five addressed merging strategies. We described common scenarios in which the default merge strategy may cause merge conflicts. By introducing popular merge strategies and their use cases, we provided a roadmap for developers to reference when deciding how they want to consolidate branches.

Chapter Six provided insight on hard resets. We discussed use cases for the hard reset and looked into how this function is handled in the Data Productivity Cloud. This section also covered the risks associated with hard resets and provided best practices for developers to follow to keep their overall risk low.

Chapter Seven introduced one of the nuances of Git: unpushed commits. We looked at how viewing unpushed commits can be used for several purposes, including impact assessment and error correction. We then demonstrated how to view unpushed commits in the Data Productivity Cloud and introduced alternate commands developers can use to find comparable information.

Chapter Eight focused on uncommitted changes. We explained how viewing uncommitted changes in a development environment is essential for reducing bugs, avoiding regressions, and selective staging. By covering several commands that retrieve uncommitted changes in the Data Productivity Cloud, we gave developers a toolbox to streamline their work and easily manage uncommitted changes.

Chapter Nine covered the revert and reset commands. We discussed the usefulness of these commands and demonstrated key differences between the two. By diving deeper into how and when to use revert and reset, we gave developers a framework to help them safely undo changes.

Over the course of these nine chapters, we covered how to use basic and advanced elements of Git to leverage version control. The features and strategies discussed provide developers with an essential overview of the tools available in Git and how to choose the best method of working based on each project.

By understanding the fundamental and nuanced features of Git, data teams can safely and effectively collaborate and experiment without losing previous states. Thank you for reading, and we encourage you to explore the Matillion Data Productivity Cloud further to streamline your data management practices.