AI VOID FIELD GUIDE

Mastering GitButler: A Zero-to-Advanced Guide

Embark on a comprehensive journey to master GitButler from the ground up, simplifying complex Git operations with virtual branches and stacked changes.

AI VOID

2026-05-04 17:25 IST

Contents

CHAPTER 01

GitButler: The Revolution Git Needed, or Just a Smarter Assistant for 2026?

Beyond the `git commit` - Why Modern Devs Need More

For over a decade, Git has been the undisputed monarch of version control. It's powerful, flexible, and ubiquitous. Yet, any developer who's navigated a complex rebase, juggled multiple in-progress features on different branches, or wrestled with a messy commit history knows that Git's power often comes with a steep cognitive load. In a world accelerating towards AI-assisted coding and increasingly complex distributed teams, the traditional Git workflow can feel... cumbersome.

Enter GitButler. Fresh off a substantial $17 million Series A funding round in early 2026, GitButler isn't just another Git GUI. It aims to fundamentally reshape how developers interact with source control, promising to simplify advanced Git features and make complex workflows intuitive. But does it deliver on this lofty promise? Is it truly "what comes after Git," or merely a highly polished, smarter assistant for our existing workflows? Let's dive deep into GitButler's impact on modern Git workflows, exploring its innovations, developer experience, and its place in the 2026 development landscape.

The Persistent Pain Points of Git (and Why They Still Hurt in 2026)

Before we laud GitButler, let's acknowledge the dragons it seeks to slay. Despite Git's robust capabilities, several common scenarios continue to plague developers:

- **Branch Management Overhead:** Juggling multiple feature branches, hotfixes, and experimental work often leads to a "branch explosion" and context switching nightmares.
- **Complex Rebases and Merges:** While powerful, `git rebase` can be intimidating, and resolving intricate merge conflicts remains a tedious, error-prone task.
- **Atomic Commits vs. Fluid Workflows:** The desire for clean, atomic commits often clashes with the reality of exploratory coding, where work evolves

incrementally and non-linearly. Squashing and amending become constant chores.

- **Visibility and Reversibility:** Understanding the state of your local changes across different conceptual tasks, and easily undoing mistakes, isn't always straightforward with vanilla Git.
- **AI Agent Collaboration:** With the rise of generative AI, an increasing portion of code is being written by AI agents. Traditional Git workflows, designed for human collaboration, struggle when multiple agents make parallel, rapid edits, leading to index breakage and conflict amplification.

These challenges aren't new, but with the pressure for faster delivery and the increasing complexity of software, they've become more acute. Developers are constantly seeking tools that can abstract away Git's sharp edges, allowing them to focus on writing code, not managing version control.

GitButler's Vision: Reimagining Source Control

GitButler positions itself not just as a tool, but as a new Source Code Management (SCM) system designed to manage branches, record work, and enhance your Git workflow. Its core philosophy revolves around making the developer's state visible, actions reversible, and workflows understandable.

It aims to turn advanced Git features into everyday tools, addressing the struggles developers face with managing branches, resolving conflicts, and organizing changes. Essentially, GitButler acts as an intelligent layer on top of your existing Git repositories, providing a more intuitive interface and a different way of thinking about your work-in-progress.

Diving Into the Butler Flow: Parallel Branches and Evolving Work

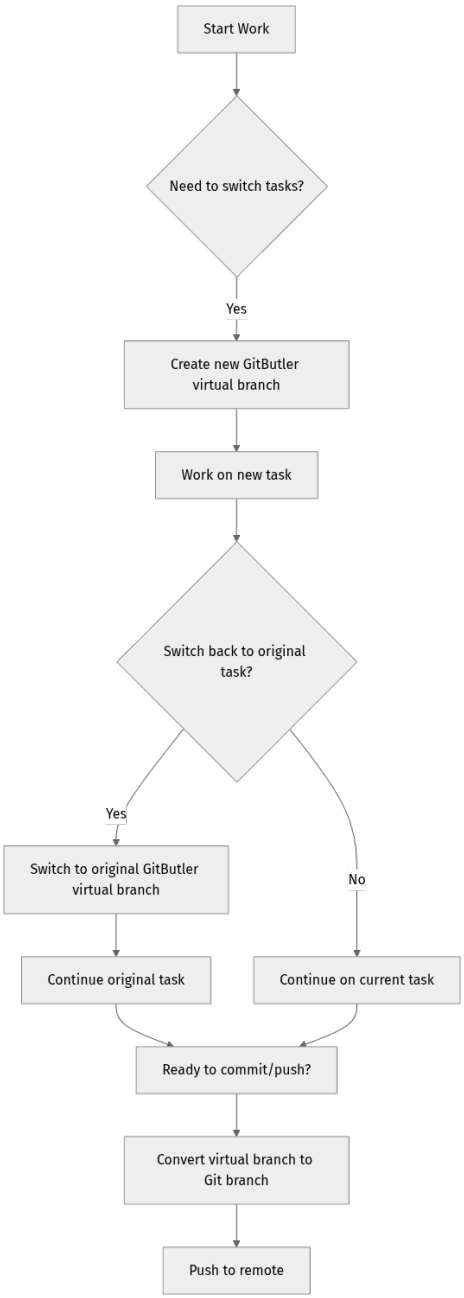
At the heart of GitButler's approach is the "Butler Flow," a lightweight, branch-based development workflow that builds upon concepts like GitHub Flow but introduces significant enhancements, particularly around parallel development.

The Power of Parallel Workspaces

One of GitButler's most compelling features is its ability to manage multiple, independent "virtual" branches or "workspaces" concurrently. Imagine working on Feature A, getting interrupted by a critical bug fix for Feature B, and then needing

to quickly prototype a new idea for Feature C – all without committing, stashing, or switching branches in the traditional sense.

GitButler achieves this by allowing you to create local "virtual branches" that exist independently of your remote Git branches. You can seamlessly switch between these local branches, cherry-pick changes between them, and even push them to remote as conventional Git branches when ready. This significantly reduces the context-switching overhead and the fear of losing uncommitted work.



Recording Work, Not Just Commits

GitButler introduces a paradigm shift from strictly "committing changes" to "recording work." It focuses on reviewing code as a series of evolving patches

rather than unified diffs. This aligns more naturally with how developers think and iterate. Instead of constantly amending or squashing commits to maintain a clean history, GitButler allows for a more fluid development process where changes can be organized and refined before being finalized into traditional Git commits. This concept, initially explored with "Butler Review" (currently paused but its philosophy remains), emphasizes the journey of development over just the destination.

This approach is particularly powerful when dealing with complex feature development or refactoring, where the final set of changes might be a distillation of many intermediate steps.

Developer Experience: Promises, Pains, and Practicalities

GitButler promises a smoother, more efficient developer experience. For many, it delivers.

The Upsides

- **Reduced Cognitive Load:** By abstracting away much of Git's underlying complexity, developers can focus more on coding and less on version control mechanics. The visual interface and intuitive CLI commands make understanding the state of your repository much easier.
- **Enhanced Productivity:** The ability to effortlessly switch between tasks and manage parallel lines of work without constant stashing or branching/unbranching significantly boosts individual productivity.
- **Cleaner History:** GitButler encourages a cleaner, more organized commit history by providing tools to easily refactor and organize changes before they become permanent Git commits.
- **Better Conflict Resolution:** While not eliminating conflicts, GitButler's focus on managing changes as evolving patches can potentially make conflict resolution more manageable by providing better context.

The Downsides and Considerations

- **Learning Curve:** While designed to simplify, GitButler introduces new concepts (like virtual branches and "recording work") that require a shift in mindset. Developers deeply ingrained in traditional Git workflows might experience an initial learning curve.

- **Integration with Existing Tooling:** GitButler operates on top of Git. While it aims for seamless integration, unique workflows or highly customized CI/CD pipelines might require minor adjustments to fully leverage GitButler's benefits.
- **Command Line Tooling:** While a desktop application exists, the CLI tool is a crucial component. Developers comfortable with pure Git CLI might initially find the `gb` commands different.

Here's a conceptual comparison of a common workflow:

Feature/ Action	Traditional Git	GitButler CLI
Start new feature	`git checkout -b feature-a`	`gb branch create feature-a`
Switch to hotfix	`git stash`, `git checkout hotfix`, `git pull`	`gb branch switch hotfix`
View local changes	`git status`, `git diff`	`gb status`, `gb diff` (across virtual branches)
Organize changes	`git add -p`, `git commit --amend`, `git rebase -i`	`gb add`, `gb amend`, `gb reorder` (more visual)
Push to remote	`git push origin feature-a`	`gb push` (after converting virtual to Git branch)

GitButler in the Modern Dev Landscape (2026)

In 2026, the software development landscape is rapidly evolving. Cloud-native architectures are standard, platform engineering is maturing, and generative AI is transforming how code is written. GitButler's timely entry, backed by significant investment, suggests it's positioned to address some critical needs.

Its ability to handle parallel editing is particularly relevant in the age of AI agents. If AI can generate large chunks of code, the tools we use to manage that code need to evolve beyond human-centric collaboration models. GitButler's approach to managing evolving patches and multiple workspaces could provide a more robust foundation for hybrid human-AI development teams.

The $17M funding round highlights investor confidence in GitButler's potential to simplify Git workflows for a broad developer base. As developers increasingly struggle with the complexities of managing numerous branches and resolving

conflicts, GitButler offers a compelling alternative that could become a standard for many teams seeking to boost productivity.

The Verdict: Is GitButler the Future, or Just a Better Present?

My opinion? GitButler isn't necessarily "replacing" Git. Git, as the underlying content-addressable storage system, remains fundamental. What GitButler *is* doing, however, is providing a powerful, opinionated, and highly intelligent layer *on top of* Git. It's an evolution, a much-needed abstraction that shields developers from Git's more arcane complexities while preserving its power.

For teams struggling with Git's cognitive overhead, or those looking to embrace more fluid, parallel development workflows, GitButler offers a compelling solution. It simplifies the everyday tasks that often lead to developer frustration, making advanced Git concepts accessible without requiring deep expertise.

In 2026, as development cycles shorten and complexity grows, tools that enhance developer productivity and reduce friction are invaluable. GitButler has the potential to become an indispensable part of the modern developer's toolkit, acting as a "butler" that handles the messy details of version control, allowing engineers to focus on what they do best: building incredible software. It's not just a better Git UI; it's a reimagined interaction model for version control that feels right for the demands of today and tomorrow.

Key Takeaways

- **Addresses Core Git Pain Points:** GitButler tackles common developer frustrations with branch management, complex rebases, and organizing fluid work.
- **Introduces "Butler Flow":** A new workflow centered around parallel virtual branches and recording evolving work, not just atomic commits.
- **Boosts Productivity:** Reduces context switching, simplifies advanced Git operations, and provides better visibility into local changes.
- **Relevant for 2026 Trends:** Its parallel editing capabilities are particularly well-suited for the rise of AI-assisted coding.
- **An Evolution, Not a Replacement:** GitButler acts as a powerful, intelligent layer over Git, making it more user-friendly and efficient without abandoning the underlying system.

References

1. GitButler raises $17M to simplify Git workflows for developers
2. Investing in GitButler - Andreessen Horowitz
3. Simplifying Git by Using GitButler | Butler's Log
4. Butler Flow - GitButler Docs
5. Git Butler: A Better Way to Work With Git (Without the Headaches) - Medium

---

This blog post is AI-assisted and reviewed. It references official documentation and recognized resources.

CHAPTER 02

Advanced Workflows & Best Practices: Collaboration and AI Integration

Introduction

Welcome to the final chapter of our GitButler mastery guide! So far, you've learned the fundamentals of GitButler, from setting up your first repository to mastering virtual branches and local commit management. You're now comfortable with its powerful UI and how it simplifies your individual Git workflow.

In this chapter, we're going to level up your skills even further by diving into advanced workflows that truly shine in team environments and with emerging challenges like integrating AI-generated code. GitButler isn't just a personal productivity tool; it's a game-changer for collaboration, especially when dealing with complex feature development and stacked changes. We'll explore how to leverage its unique capabilities to streamline pull request reviews, manage intricate dependencies, and even tame the often-messy output of AI coding assistants.

By the end of this chapter, you'll understand how to effectively integrate GitButler into a collaborative team, harness its power for managing multi-layered features, and apply its principles to refine AI-generated code, ensuring your projects remain clean, maintainable, and easy to review. Get ready to transform your team's Git experience!

Core Concepts

In advanced development, especially within teams, workflows often become more complex. You might be working on a feature that depends on another, or you might receive contributions from an AI assistant that need careful integration. GitButler offers elegant solutions to these challenges.

Collaborative Workflows with Stacked Branches

One of the most powerful features of GitButler for teams is its native support for **stacked branches**. In traditional Git, if you have a feature `B` that depends on

feature `A`, and feature `C` that depends on `B`, you'd typically create `feature-A`, then branch `feature-B` off `feature-A`, and `feature-C` off `feature-B`.

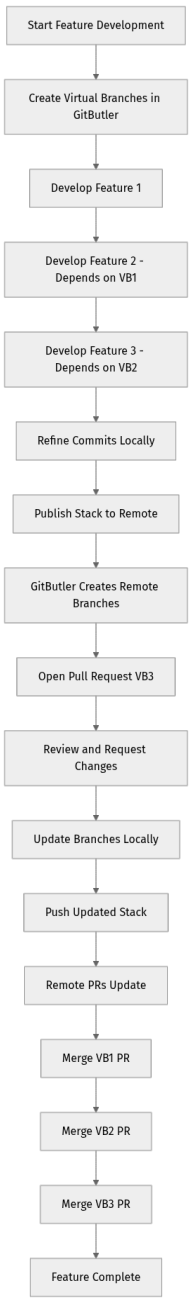
When it comes time to review these, you'd open a Pull Request (PR) for `feature-A`. Once `A` is merged, you'd rebase `feature-B` onto `main` (or `develop`), then open a PR for `B`, and so on. This process, especially the rebasing part, can be tedious, error-prone, and frustrating for both authors and reviewers. It often leads to "rebase hell" where changes get lost or conflicts are constantly re-resolved.

GitButler's virtual branches, when *published*, fundamentally change this. You can build your stack of dependent virtual branches locally, refine them perfectly, and then tell GitButler to "publish" them. GitButler will create actual Git branches on your remote repository (e.g., GitHub, GitLab) for *each* virtual branch in your stack.

This means:

- **Easy Review:** Reviewers can see the logical progression of changes. They can review `feature-A` in isolation, then `feature-B` building on `A`, and `feature-C` building on `B`.
- **Simplified Updates:** If changes are requested on `feature-A`, you make them on your local `VB_feature-A`, commit, and then GitButler automatically propagates those changes up the stack when you "push" or "publish" again. No manual rebasing required!
- **Atomic Merges:** As each PR is merged (starting from the bottom of the stack), GitButler helps you keep your local stack in sync, ensuring a smooth, conflict-free experience.

Let's visualize this with a simple diagram:



In this diagram, `VB1`, `VB2`, and `VB3` represent your virtual branches. GitButler handles the synchronization and creation of remote branches, making the entire process much smoother.

Managing AI-Generated Code

The rise of AI coding assistants (like GitHub Copilot, ChatGPT, etc.) presents both incredible opportunities and unique challenges. AI can generate large chunks of code quickly, but this code often requires:

- **Refinement:** It might not perfectly match your project's style, best practices, or specific requirements.

- **Splitting:** A single AI "suggestion" might encompass multiple logical changes that should be separate commits.
- **Review:** AI code still needs human oversight to ensure correctness, security, and performance.
- **Experimentation:** You might want to try out several AI suggestions before committing to one.

GitButler's local-first virtual branch model is exceptionally well-suited for managing AI-generated code:

1. **Isolated Experimentation:** When an AI suggests a change, you can instantly create a new virtual branch in GitButler. This allows you to apply the AI's suggestion, test it, and see if it works without affecting your main work. If it's not good, simply discard the virtual branch.
2. **Granular Staging:** AI often produces a lot of code at once. GitButler's intuitive staging area allows you to pick and choose *which lines* from the AI's output you want to commit. You can easily split a large AI contribution into multiple, logical, and well-explained commits.
3. **Easy Reordering and Squashing:** If the AI generates several changes that you commit individually, but later realize they should be squashed into one, or reordered for better readability, GitButler's commit graph and drag-and-drop interface make this trivial. You can sculpt the AI's raw output into a perfectly polished commit history.
4. **Local Refinement Before Sharing:** You can iterate on AI-generated code locally as much as you need, cleaning up, refactoring, and adding tests, all within GitButler's safe, virtual environment, before ever pushing it to a remote for team review.

Essentially, GitButler acts as a powerful sandbox and sculpting tool for AI-generated code, turning raw suggestions into production-ready contributions.

Integrating GitButler with Existing Git Workflows

GitButler is designed to enhance, not replace, your underlying Git knowledge. It works *on top* of Git. This means it can seamlessly integrate into various team environments:

- **Coexistence:** Team members who prefer the command line can continue using it. GitButler users will manage their local work within the application, but all pushes and pulls interact with the same remote Git repository.

- **Phased Adoption:** You can introduce GitButler to your team gradually. Start with individual developers, then small teams, demonstrating its benefits in managing local changes and creating clean histories.
- **Pull Request Workflow:** GitButler integrates perfectly with standard PR workflows. When you publish a virtual branch, it becomes a regular Git branch on your remote, from which you can open a PR. Reviewers interact with it just like any other PR.
- **GitButler as a "Local Git GUI on Steroids":** Think of GitButler as your personal workbench for local Git operations. It simplifies `add`, `commit`, `rebase`, `cherry-pick`, and `stash` operations, allowing you to craft the perfect history before interacting with the shared remote.

Best Practice: Encourage team members to maintain a clean local history in GitButler before pushing. This minimizes conflicts and makes code reviews much easier for everyone.

Step-by-Step Implementation: Advanced Scenarios

Let's walk through two practical scenarios: publishing a stack of virtual branches for team review and refining AI-generated code.

Scenario 1: Publishing a Stack for Team Review

Imagine you've been working on a new feature that required three dependent sub-features. You've structured them as `feature/core-logic`, `feature/ui-integration` (depends on `core-logic`), and `feature/api-updates` (depends on `ui-integration`). Each is a separate virtual branch in GitButler.

1. **Prepare Your Stack Locally:** Make sure all your virtual branches are in a ready state. Open GitButler and navigate to your repository. You should see your virtual branches stacked on top of each other in the left panel.

- **Self-check:** Have you committed all changes to their respective virtual branches? Are the commit messages clear?

1. **Initiate Publishing:** In the GitButler UI, locate the "Publish" or "Push" button, typically near the top right or next to your active branch. GitButler intelligently detects your stacked branches. When you click "Publish," GitButler will likely present you with a dialog.

- **Explanation:** GitButler needs to know where to push these branches. It will suggest creating new remote branches corresponding to your virtual

branches. For example, your `feature/core-logic` virtual branch will become a remote branch named `feature/core-logic` (or a similar naming convention you prefer).

1. **Confirm and Push:** Review the proposed remote branch names and confirm the action. GitButler will then perform the necessary Git operations to push each of your virtual branches to the remote repository.

- **What's happening:** Behind the scenes, GitButler is executing `git push` commands for each relevant branch in your stack, ensuring that the dependencies are correctly represented on the remote. For instance, `feature/ui-integration` will be pushed such that it branches off `feature/core-logic` on the remote.

  After the push is complete, you will see a confirmation message. Now, if you visit your remote repository's interface (e.g., GitHub), you will find these new branches.

1. **Create Pull Requests:** From your remote repository's web interface (e.g., GitHub, GitLab), you can now create Pull Requests.

- **Best Practice:** Start by creating a PR for the *topmost* branch in your stack (e.g., `feature/api-updates`) targeting `main` or `develop`. Many platforms like GitHub will automatically detect that this branch depends on others and will show a "stacked" view or suggest reviewing the base branches first.
- **Alternative:** You could also create individual PRs for each branch (`feature/core-logic` -> `main`, `feature/ui-integration` -> `feature/core-logic`, `feature/api-updates` -> `feature/ui-integration`). However, pushing the whole stack and opening one PR for the top is often simpler for reviewers. GitButler simplifies this by making the dependent branches easy to manage.

  As reviewers provide feedback, you can address it on the relevant virtual branch in GitButler, commit your changes, and then "Push" again. GitButler will handle updating the remote branches and their associated PRs.

Scenario 2: Refining AI-Generated Changes

Let's say you've used an AI assistant to generate a new utility function and integrate it into your codebase. The AI produced a single, large block of code with a single commit message like "Add new utility." This isn't ideal for review.

1. **Identify the AI-Generated Commit:** In GitButler, switch to the virtual branch where the AI-generated code was committed. You'll likely see a single large commit.
2. **Split the Large Commit:**
    - Select the large AI-generated commit in GitButler's commit history view.
    - Look for an option to "Split Commit" or "Edit Commit" (the exact wording might vary slightly, but the functionality is there).
    - GitButler will present you with the changes contained within that commit. You can then interactively select lines or hunks of code and assign them to *new* commits.

- **Explanation:** Instead of one giant commit, you might want: * Commit 1: "feat: Add `calculateDiscount` utility function" (just the function definition) * Commit 2: "refactor: Integrate `calculateDiscount` into order processing" (where the function is called) * Commit 3: "test: Add unit tests for `calculateDiscount`" (the tests for the new function)

  Go through the process, creating these new, smaller, logical commits. GitButler will guide you through selecting the relevant changes for each new commit.

1. **Reorder and Rename Commits (Optional):** Once you've split the large commit into several smaller ones, you might decide that the test commit should come immediately after the function definition, or that a commit message needs improvement.
    - In GitButler's commit history, you can often drag and drop commits to reorder them.
    - Right-click on a commit to "Edit Message" or "Amend" it.

- **Why this matters:** A clean, logical commit history makes it much easier for your team to understand the evolution of the codebase and provides a clear narrative for code reviews.

1. **Push Refined Changes:** Once you're satisfied with your sculpted commit history, you can push your virtual branch to the remote. GitButler will ensure

that your refined, multi-commit history is pushed, not the original messy AI commit.

- **Benefit:** Your team will review a well-structured set of changes, making the review process faster and more effective.

Mini-Challenge: Orchestrating a Dependent Feature Stack

Let's put your knowledge of stacked branches and refinement to the test!

Challenge: You need to implement a new user authentication flow. This requires: 1. A `core-auth` module (virtual branch `feature/auth-core`). 2. A `login-ui` component that uses `core-auth` (virtual branch `feature/auth-login-ui`, depends on `feature/auth-core`). 3. A `password-reset` feature that also uses `core-auth` (virtual branch `feature/auth-password-reset`, depends on `feature/auth-core`).

Your Task: 1. **Simulate Development:** On your GitButler repository, create `feature/auth-core`. Add a file (e.g., `src/auth/core.js`) with a dummy function like `export function authenticate(user, pass) { /* ... */ }`. Commit this. 2. **Build the Stack:** Create `feature/auth-login-ui` *branching off* `feature/auth-core`. Add a file (e.g., `src/components/Login.vue`) that imports and "uses" your dummy `authenticate` function. Commit this. 3. **Add Another Dependent Branch:** Create `feature/auth-password-reset` *also branching off* `feature/auth-core` (or you could branch it off `feature/auth-login-ui` if you wanted to simulate a deeper stack, but for this challenge, branching both UI components off `core-auth` is fine). Add a dummy `src/components/PasswordReset.vue`. Commit this. 4. **Refine (Optional but Recommended):** Go through your commits. Do any need to be squashed or reordered for clarity? Practice using GitButler's commit editing features. 5. **Publish:** Use GitButler to publish this entire stack of virtual branches to your remote repository. 6. **Verify:** Check your remote repository (e.g., GitHub) to confirm that the `feature/auth-core`, `feature/auth-login-ui`, and `feature/auth-password-reset` branches now exist there.

Hint: When creating new virtual branches, ensure you select the correct "base branch" in GitButler (e.g., `feature/auth-login-ui` should be based on `feature/auth-core`). When publishing, GitButler will usually handle the dependencies correctly.

What to Observe/Learn: * How GitButler visualizes the dependencies between your virtual branches. * The ease of pushing multiple dependent branches to a remote without manual rebasing. * How your remote repository now reflects a logical, stacked workflow, ready for PRs.

Common Pitfalls & Troubleshooting

Even with GitButler simplifying things, advanced workflows can have their quirks.

1. **Confusing Virtual Branches with Remote Branches:**

- **Pitfall:** Remembering that GitButler's virtual branches are primarily *local* until you `publish` or `push` them. They don't automatically exist on your remote just because you created them in GitButler.
- **Troubleshooting:** If a team member can't see your branch, ensure you've explicitly pushed it using GitButler's "Publish" or "Push" functionality. GitButler will then create the corresponding remote Git branch. Always verify on your remote host (GitHub, GitLab, etc.).

1. **Rebasing Shared Stacks (Even with GitButler):**

- **Pitfall:** While GitButler greatly simplifies local rebase operations, if you or a team member manually `git rebase` a branch that's *part of a published stack* and then force push it, it can cause problems for others who have pulled those branches.
- **Troubleshooting:**
- **Best Practice:** The golden rule of Git still applies: "Don't rebase published history unless absolutely necessary and communicated." * If changes are needed on a published stack, make them on the relevant virtual branch in GitButler, commit, and then push. GitButler will handle the update gracefully. * If a rebase *is* needed (e.g., to squash commits *after* initial review but *before* final merge), ensure all collaborators are aware and know how to re-sync their local copies (e.g., `git pull --rebase` or `git reset --hard` and re-pull). GitButler tries to mitigate this, but underlying Git principles still apply.

1. **Team Adoption and Onboarding Challenges:**

- **Pitfall:** Introducing a new tool like GitButler to a team accustomed to command-line Git can face resistance.
- **Troubleshooting:**

- **Start Small:** Encourage early adopters to try GitButler for personal productivity.
- **Demonstrate Value:** Show concrete examples of how GitButler solves common pain points (e.g., "rebase hell," messy history, managing AI code).
- **Training and Documentation:** Provide internal guides and short training sessions. Emphasize that GitButler *complements* Git, making it easier, not replacing fundamental Git knowledge.
- **Official Docs:** Point to the official GitButler documentation (https://docs.gitbutler.com/guide) for comprehensive reference.

Summary

Congratulations! You've reached the end of our GitButler journey, moving from foundational concepts to advanced, collaborative workflows.

Here are the key takeaways from this chapter:

- **Stacked Branches for Collaboration:** GitButler makes managing dependent features and pull requests incredibly efficient. You can build complex stacks locally, publish them to your remote, and GitButler handles the synchronization, dramatically simplifying the review and update process compared to traditional Git rebasing.
- **Managing AI-Generated Code:** GitButler's virtual branches and granular commit control provide an ideal environment for refining, splitting, and organizing often-messy AI-generated code, transforming it into clean, reviewable contributions.
- **Seamless Integration:** GitButler coexists gracefully with existing Git workflows and command-line tools, making it easy to adopt within a team without disrupting current practices. It acts as a powerful local Git workbench.
- **Best Practices:** Always strive for a clean, logical commit history, especially before pushing to a shared remote. Communicate with your team about tool adoption and workflow changes.

You now possess a comprehensive understanding of GitButler, from its installation and basic usage to its most powerful features for individual productivity and team collaboration. You're equipped to tackle complex Git challenges with confidence and streamline your development workflow.

What's Next?

- **Practice, Practice, Practice:** The best way to solidify your understanding is to use GitButler daily. Start applying its principles to your own projects and team work.
- **Explore Team Features:** If you're in a team, advocate for GitButler and explore its potential for enhancing your team's specific Git workflow.
- **Stay Updated:** GitButler is an evolving tool. Keep an eye on its official documentation and releases for new features and improvements.
- **Deep Dive into Git:** While GitButler simplifies many operations, a solid understanding of underlying Git concepts will always be beneficial.

Thank you for joining us on this learning adventure. Happy coding with GitButler!

References

- GitButler Official Website
- Getting Started - GitButler Documentation
- Stacked Branches - GitButler Documentation
- GitButler for AI-Agent Workflows (Conceptual) - While not a direct doc, this blog post highlights GitButler's relevance to AI code management.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 03

Building with Stacks: Effortlessly Managing Dependent Changes

Building with Stacks: Effortlessly Managing Dependent Changes

Welcome back, intrepid Git explorer! In our previous chapters, you've embraced the power of GitButler's virtual branches, discovering how they free you from the constraints of traditional Git workflows. You've learned to manage changes locally, creating and switching between isolated workstreams with ease.

But what happens when your work isn't so isolated? What if you're building a large feature that needs to be broken down into several smaller, dependent changes? Or perhaps you're working on a bug fix that requires a preliminary refactor. In traditional Git, this often leads to a tangled mess of `git rebase -i` commands, complex pull requests, and the dreaded "merge conflict marathon."

In this chapter, we're going to tackle this common challenge head-on by introducing one of GitButler's most powerful features: **stacked branches**, often simply called "stacks." You'll learn what stacks are, why they're a game-changer for managing dependent changes, and how to wield them effortlessly within GitButler. Get ready to streamline your development process and say goodbye to Git headaches!

The Pain of Dependent Changes in Traditional Git

Before we dive into the solution, let's quickly recap the problem. Imagine you're building a new user authentication system. This might involve:

1. **Refactor:** Updating the existing user model to support new fields.
2. **API Endpoints:** Adding new API routes for registration and login.
3. **Frontend UI:** Building the login and registration forms.

In a traditional Git workflow, you might create a branch for the refactor, then branch off *that* for the API, and then off the API branch for the UI. This creates a dependency chain.

- `main`
  - `feature/auth-refactor`
    - `feature/auth-api`
      - `feature/auth-ui`

This looks reasonable, right? The challenge arises when `main` updates (e.g., a critical bug fix is merged), or when you need to make a change deep in `feature/auth-refactor` after `feature/auth-ui` is already in progress. You'd have to `rebase` `feature/auth-refactor` onto `main`, then `rebase` `feature/auth-api` onto the *new* `feature/auth-refactor`, and finally `rebase` `feature/auth-ui` onto the *new* `feature/auth-api`. This "rebase cascade" is prone to errors, especially with merge conflicts, and can be incredibly time-consuming.

It's a lot like trying to re-stack a Jenga tower after pulling a block from the middle – possible, but precarious!

Introducing GitButler Stacks: Your Rebase-Free Solution

GitButler's **stacked branches** (or simply "stacks") offer an elegant solution to this problem. A stack is essentially a sequence of virtual branches where each branch builds directly on top of the previous one. GitButler understands these dependencies and intelligently manages them for you.

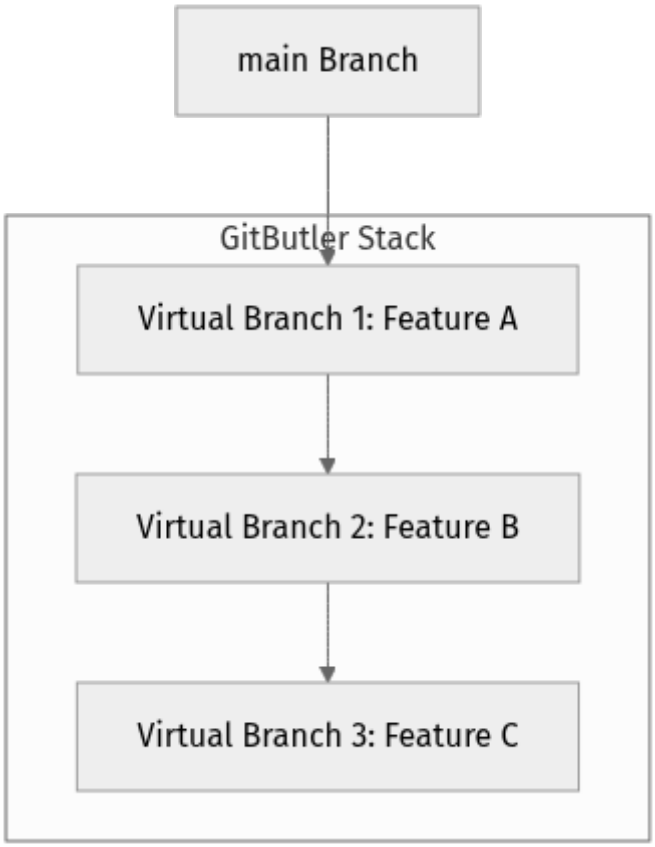
Here's the magic:

1. **Clear Dependencies:** You visually see which branch depends on which.
2. **Automatic Rebasing:** When the base of your stack (e.g., `main` or the first branch in your stack) gets updated, GitButler automatically rebases the entire stack for you. No manual `git rebase` commands required!
3. **Granular Pull Requests:** Each virtual branch in your stack can be proposed as a separate pull request, allowing for smaller, easier-to-review changes that still maintain their dependencies.
4. **Flexibility:** You can easily reorder branches within a stack, or even "unstack" a branch if its dependency changes.

Think of it like building a LEGO tower. Each LEGO brick (virtual branch) connects perfectly to the one below it. If you need to change the base of your tower,

GitButler automatically adjusts all the bricks above it, ensuring they remain connected and in order.

Let's visualize this with a simple diagram:



In this diagram, `Feature B` is built on top of `Feature A`, and `Feature C` is built on top of `Feature B`. GitButler treats `Feature A`, `Feature B`, and `Feature C` as a single, cohesive unit for management purposes, even though they are distinct virtual branches.

This powerful concept is particularly useful for:

- **Large Feature Development:** Breaking down big tasks into manageable, dependent sub-tasks.
- **Refactoring:** Performing a refactor in one branch, then building new features on top of that refactor in subsequent branches.
- **AI-Agent Workflows:** When an AI agent generates iterative code changes, stacks allow you to review and commit each small, dependent step as a separate virtual branch, making the review process much clearer and easier to manage.

Step-by-Step: Building Your First Stack

Let's get hands-on and create a stack within GitButler. We'll simulate developing a new feature that requires a preliminary setup.

Scenario: We want to add a new "Dark Mode" feature to our application. This will involve: 1. `feature/dark-mode-setup`: Initial setup, adding a theme context or utility. 2. `feature/dark-mode-ui`: Implementing the actual UI toggle and styling changes.

1. Start with a Clean Base

First, ensure you're on your `main` branch (or `master`, depending on your repository's default) and that it's up-to-date.

1. Open your GitButler desktop application.
2. Navigate to your repository (if not already there).
3. In the "Branches" sidebar, ensure `main` is selected as your active branch. If not, click on it.
4. Click the "Pull" button to ensure your `main` branch is synchronized with the remote.

2. Create the First Branch in Your Stack

Now, let's create our first virtual branch, which will be the foundation of our stack.

1. In GitButler, click the `+` button next to "Branches" in the sidebar, or click the "New Branch" button at the top.
2. Name this branch `feature/dark-mode-setup`.
3. Click "Create Branch."

You are now on `feature/dark-mode-setup`. Notice how GitButler shows `main` as its base.

3. Make and Commit Changes for the First Branch

Let's simulate adding some setup code for dark mode.

1. Open your preferred code editor (e.g., VS Code) in your repository's directory.
2. Create a new file, say `src/utils/theme.js`, with the following content:

   ```javascript // src/utils/theme.js export const lightTheme = { background: '#ffffff', text: '#333333', };

   export const darkTheme = { background: '#333333', text: '#ffffff', };
   ```

export const getTheme = (isDarkMode) => { return isDarkMode ? darkTheme : lightTheme; }; ```

What are we doing here? We're creating a simple utility file that defines light and dark theme objects and a function to get the appropriate theme based on an `isDarkMode` flag. This is foundational for our dark mode feature.

3. Save the file.
4. Switch back to GitButler. You'll see `src/utils/theme.js` listed in the "Changes" panel.
5. Select the `src/utils/theme.js` file.
6. In the commit message box, type: `feat: Add basic theme utility for dark mode`
7. Click the "Commit" button.

Great! You've made your first commit on `feature/dark-mode-setup`.

4. Create the Second Branch, Stacked on Top

Now, for the "stacking" part! We'll create `feature/dark-mode-ui` *on top of* `feature/dark-mode-setup`.

1. Ensure `feature/dark-mode-setup` is still your active branch in GitButler.
2. Click the `+` button next to "Branches" again, or the "New Branch" button.
3. Name this branch `feature/dark-mode-ui`.
4. GitButler will automatically suggest `feature/dark-mode-setup` as the base for this new branch because it's your currently active branch. This is crucial for stacking!
5. Click "Create Branch."

Look at the "Branches" sidebar. You should now see `feature/dark-mode-ui` listed, and visually, it will appear "stacked" on top of `feature/dark-mode-setup`. This visual representation is GitButler's way of showing the dependency.

5. Make and Commit Changes for the Second Branch

Let's add some UI elements that depend on our `theme.js` utility.

1. Open your code editor.
2. Modify an existing file, or create a new one like `src/App.js` (assuming a simple React-like app structure) to simulate UI changes. Add a simple toggle and use the theme utility:

```javascript // src/App.js (example - adapt to your project) import React, { useState } from 'react'; import { getTheme } from './utils/theme'; // This import depends on feature/dark-mode-setup

function App() { const [isDarkMode, setIsDarkMode] = useState(false); const theme = getTheme(isDarkMode);

const appStyle = { backgroundColor: theme.background, color: theme.text, minHeight: '100vh', display: 'flex', flexDirection: 'column', alignItems: 'center', justifyContent: 'center', transition: 'background-color 0.3s, color 0.3s', };

return (

# Welcome to our App!

This is some content that will change with the theme.

setIsDarkMode(!isDarkMode)}

); }

export default App; ```

*Why this code?* This `App.js` file *depends* on `src/utils/theme.js` (which was created in the previous branch). This clearly demonstrates the dependency in our stack.

3. Save the file.
4. Switch back to GitButler. You'll see `src/App.js` in your "Changes" panel.
5. Select the `src/App.js` file.
6. In the commit message box, type: `feat: Implement dark mode UI toggle`
```

7. Click the "Commit" button.

6. Observe Your Stack

Now, take a moment to look at the "Branches" sidebar in GitButler.

You should see something like this:

```
main
  └─ feature/dark-mode-setup
       └─ feature/dark-mode-ui
```

This visual hierarchy clearly shows that `feature/dark-mode-ui` is stacked on `feature/dark-mode-setup`, which in turn is based on `main`. You've successfully created your first stack!

7. How GitButler Manages Updates (The Magic!)

Imagine now that someone else pushes a new commit to `main`.

1. Go back to your `main` branch in GitButler (click on `main` in the sidebar).
2. Click "Pull" to fetch the "new" commit (even if there isn't one, this simulates fetching updates).
3. Now, switch back to `feature/dark-mode-setup` (click on it).
4. GitButler will automatically detect that `main` has new commits and will offer to rebase your `feature/dark-mode-setup` branch (and consequently, `feature/dark-mode-ui`) onto the latest `main`. You might see a prompt or a subtle indicator.

This is where GitButler shines! It handles the complex rebasing operation for you across the entire stack, ensuring that your dependent branches are always up-to-date with their base. If there are conflicts, GitButler will guide you through resolving them within the UI.

Mini-Challenge: Extend Your Stack

Let's solidify your understanding by extending this stack.

Challenge: Add a third virtual branch, `feature/dark-mode-persistence`, on top of `feature/dark-mode-ui`. In this new branch, simulate adding code to save the user's dark mode preference to local storage.

1. **Create a New Branch:** Ensure `feature/dark-mode-ui` is active, then create `feature/dark-mode-persistence` on top of it.

2. **Add Code:** In your code editor, modify `src/App.js` to:
    - Use `localStorage.setItem('darkMode', JSON.stringify(isDarkMode));` when `isDarkMode` changes.
    - Initialize `isDarkMode` from `localStorage.getItem('darkMode')` on component mount, parsing the stored string back to a boolean.
3. **Commit:** Commit these changes with an appropriate message.

What to Observe/Learn: * How GitButler visually updates the stack in the "Branches" sidebar. * The clear dependency chain: `main` -> `dark-mode-setup` -> `dark-mode-ui` -> `dark-mode-persistence`. * How easy it is to extend an existing stack without manual rebasing.

(Take your time with this challenge. Remember, the goal is understanding, not just copying!)

Common Pitfalls & Troubleshooting

Even with GitButler's simplified approach, understanding a few common areas can prevent frustration.

1. **Confusing Virtual Branches with Stacks:**

- **Pitfall:** Thinking every virtual branch is automatically part of a "stack."
- **Clarification:** A stack is a *sequence* of virtual branches where each is explicitly based on the previous one. You can have independent virtual branches that are *not* part of a stack. When you create a new branch, GitButler defaults to basing it on your *currently active* virtual branch, which is how you build a stack. If you want an independent branch, switch to `main` (or another desired base) *before* creating the new branch.
- **Troubleshooting:** Always check the "Branches" sidebar in GitButler to see the current hierarchy and ensure your new branch is based on the correct parent.

1. **Pushing a Stack:**

- **Pitfall:** Expecting a single "push stack" button that pushes all branches as one.
- **Clarification:** Each branch in a stack is still an individual Git branch. When you are ready to share your work, you typically push each branch in the stack individually to your remote. GitButler makes this easy: when you're on a branch in your stack, the "Push" button will push *that specific branch* to the remote, along with its necessary dependencies. For example, if you push

`feature/dark-mode-ui`, GitButler will ensure `feature/dark-mode-setup` is also pushed if it hasn't been already.

- **Best Practice:** Push the branches in order from bottom to top (e.g., `dark-mode-setup` first, then `dark-mode-ui`, then `dark-mode-persistence`). This ensures that the remote repository has the correct base for each dependent branch when they are created as pull requests.

1. **Merge Conflicts During Rebase:**

- **Pitfall:** Believing GitButler entirely eliminates merge conflicts.
- **Clarification:** GitButler automates the *process* of rebasing, but it cannot magically resolve logical conflicts in your code. If `main` changes a line that one of your stacked branches also changed, a conflict will still occur.
- **Troubleshooting:** GitButler's UI will clearly indicate when a conflict arises during an automatic rebase. It will guide you through the conflict resolution process, often allowing you to resolve it directly within the application or by opening your preferred merge tool. Pay close attention to the conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`) and carefully choose which changes to keep.

Summary

Congratulations! You've successfully navigated the world of GitButler stacks. Here are the key takeaways:

- **Stacks simplify dependent changes:** They allow you to break down large features into smaller, manageable, and interdependent virtual branches.
- **Automatic Rebasing is a game-changer:** GitButler handles the complex task of rebasing your entire stack when your base branch updates, saving you time and preventing errors.
- **Visual clarity:** The GitButler UI provides a clear visual representation of your branch dependencies.
- **Enhanced collaboration:** Stacks enable granular pull requests, making code reviews easier and more focused.
- **Perfect for iterative workflows:** Ideal for managing changes from AI agents or any situation requiring careful, layered development.

You're now equipped to tackle even the most complex feature development with confidence, leveraging GitButler to keep your workflow smooth and your Git history clean.

What's Next?

In the next chapter, we'll dive into how to effectively collaborate with others using GitButler. We'll explore how to share your virtual branches and stacks, review code, and integrate your GitButler workflow seamlessly into team environments. Get ready to supercharge your team's productivity!

References

- GitButler Official Website
- Getting Started - GitButler Docs
- Stacked Branches - GitButler Docs

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 04

Your First Steps: Navigating the GitButler Interface and Local Repositories

Your First Steps: Navigating the GitButler Interface and Local Repositories

Welcome back, aspiring Git workflow wizard! In Chapter 1: Getting Started with GitButler, you successfully installed GitButler and prepared your system. Now, it's time to dive into the exciting part: exploring its intuitive interface and making your first changes using its unique approach to Git.

This chapter will be your guided tour through the GitButler desktop application. We'll learn how to add your existing Git repositories, understand how GitButler visualizes your work, and most importantly, grasp the foundational concept of "virtual branches." By the end, you'll be comfortable creating and committing changes within GitButler, setting the stage for a much smoother and more flexible development experience.

Ready to transform your Git workflow? Let's get started!

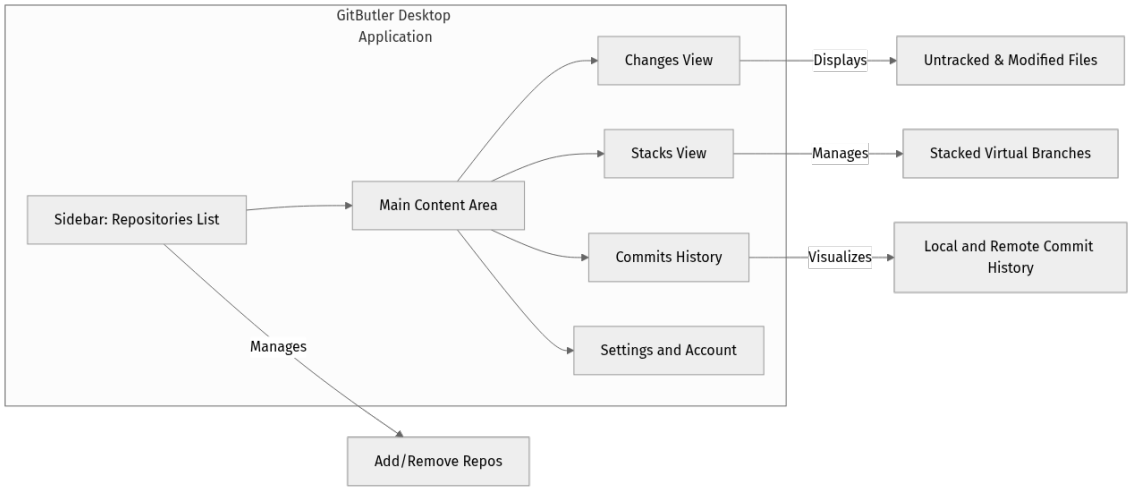
Core Concepts: Understanding GitButler's World

Before we click any buttons, let's get acquainted with the core ideas that make GitButler so powerful.

The GitButler Interface: Your New Command Center

When you first open GitButler, you'll be greeted by a clean, modern interface designed to make complex Git operations feel simple. Think of it as your personal cockpit for all your local Git work.

Let's break down the main areas you'll encounter. While the exact layout might evolve with future updates (we're assuming a stable release around 2026-04-10), the core components remain consistent.



- **Sidebar (A: Repositories List):** On the left, you'll find a list of all the Git repositories you've added to GitButler. This is where you'll switch between projects.
- **Main Content Area (B):** This large central area changes based on what you're doing. It's context-sensitive, meaning it shows you the most relevant information for your current task.
- **Changes View (C):** This is where GitButler shines! It displays all your uncommitted modifications across your entire repository, regardless of what traditional Git branch you *think* you're on. This is a radical shift from running `git status` in the terminal.
- **Stacks View (D):** This is where your "virtual branches" live. You'll use this area to create, manage, and reorder your stacked changes.
- **Commits History (E):** A visual representation of your commit history, allowing you to browse, reorder, and modify past commits with ease.
- **Settings and Account (F):** Access to application settings, user account information, and helpful resources.

Our focus today will primarily be on the **Sidebar** and the **Changes View**, as these are fundamental for getting started.

Understanding Changes: The GitButler Way

In traditional Git, you often run `git status` to see what files are modified, added, or deleted. Then, you `git add` specific files to the staging area before committing. GitButler simplifies this by presenting all your uncommitted changes in a clear, interactive "Changes" view.

What to Observe:

- **Automatic Detection:** GitButler automatically detects file system changes in your repository. No need to manually `git status`.
- **Diff View:** For modified files, GitButler shows you a clear diff (differences) between your current working directory and the last committed version.
- **Selective Staging:** You'll see checkboxes or selection mechanisms next to files or even individual lines within files. This allows you to selectively "stage" changes for a commit, similar to `git add -p` but visually and more intuitively.

The Magic of Virtual Branches

This is where GitButler truly differentiates itself. Forget everything you know about `git branch` for a moment. GitButler introduces the concept of **virtual branches**.

Imagine this scenario: You're knee-deep in developing a new feature. Suddenly, a critical bug report comes in that requires an immediate fix. In traditional Git, you'd typically have to `git stash` your current work, `git checkout` to a stable branch, create a new bugfix branch, fix the bug, commit, merge it back, delete the bugfix branch, `git stash pop`, and then try to remember exactly where you left off. Phew! That's a lot of context switching and mental overhead.

With GitButler, it's different. All your local changes, regardless of what you're working on, exist in a pool of "uncommitted changes." When you decide to commit, you don't commit to a traditional Git branch directly. Instead, you create a **virtual branch** *on the fly* to hold those specific changes.

- **What is a Virtual Branch?** Think of a virtual branch as a temporary, local container for a set of related changes. It's not a "real" Git branch (like those you push to GitHub or GitLab) until you explicitly choose to *materialize* and push it later. For now, it's purely a local organizational tool.
- **Why use them?**
- **Isolation:** Keep different feature developments or bug fixes completely separate, even if they touch the same files, without constantly switching traditional branches.
- **Flexibility:** Easily reorder, combine, or discard these virtual branches without complex `git rebase` commands.
- **Local-First:** All this magic happens locally, giving you a sandbox to refine your work before ever touching your remote repository.

It's like having a stack of transparent sticky notes, each with a small, independent change. You can rearrange them, add new ones, or throw them away, all without affecting the underlying document (your `main` branch) until you're ready.

Step-by-Step Implementation: Your First Virtual Branch

Let's put these concepts into practice. We'll add a repository, create a simple change, and then commit it to a brand-new virtual branch.

Prerequisites: * GitButler is open and running. * You have an existing local Git repository. If you don't, let's quickly create one: 1. Open your terminal or command prompt. 2. Create a new directory and initialize a Git repository: `bash mkdir my-gitbutler-project cd my-gitbutler-project git init` 3. Create an initial file for Git to track and commit it: `bash echo "# My GitButler Project" > README.md git add . git commit -m "Initial commit of README"` Now you have a repository named `my-gitbutler-project` ready to be added to GitButler.

Step 1: Add Your First Repository to GitButler

GitButler doesn't just magically know about your projects; you need to tell it which Git repositories you want it to manage.

1. **Open GitButler:** Launch the GitButler desktop application.
2. **Add Repository:**
    - If it's your first time, you'll likely see a prominent "Add Repository" button or a prompt to add one in the main content area.
    - Alternatively, look for a `+` icon or an "Add Repository" option in the sidebar or application menu.
3. **Browse and Select:** A file browser will appear. Navigate to the root directory of your `my-gitbutler-project` (or any other existing Git repository) and select it.

- **Pro-Tip:** Make sure it's a directory that already contains a `.git` folder! GitButler manages *existing* Git repositories. 4. **Confirm:** Once selected, GitButler will quickly scan the repository and add it to your sidebar.

Congratulations! Your repository is now under GitButler's watchful eye.

Step 2: Observe the Initial State in GitButler

With your repository added and selected in the sidebar:

1. Look at the **Changes View**. It should currently be empty, indicating "No pending changes." This is because your `README.md` was already committed via the terminal.
2. Look at the **Stacks View**. It should show your `main` (or `master`) branch at the bottom, representing the base of your work. You'll likely see a "Working Copy" area above it, indicating where new changes will appear.

Step 3: Make Your First Local Change

Let's create a new file in your repository.

1. Open your `my-gitbutler-project` directory in your favorite code editor (VS Code, Sublime Text, etc.).
2. Create a new file named `hello-gitbutler.txt` in the root of your project.
3. Add the following content to the file:

   ```
   text Hello, GitButler! This is my first change using virtual branches.
   ```

4. Save the file.

Step 4: See the Change Reflected in GitButler

1. Switch back to the GitButler application.
2. Magically, you should now see `hello-gitbutler.txt` listed in the **Changes View**! GitButler automatically detects file system changes.

- **What to observe:** You'll see the file name, an indicator that it's a new file (often a `+` icon or "Untracked"), and potentially a preview of its content differences.

Step 5: Create a Virtual Branch and Commit

Now, let's commit this change. Instead of committing directly to `main`, we'll put it on a virtual branch.

1. In the **Changes View**, ensure the checkbox next to `hello-gitbutler.txt` is selected. This "stages" the file for your commit, indicating which changes you want to include.

2. Look for a "Create Virtual Branch" button or a similar action (it might be labeled "Commit" or "Create Branch"). It's usually prominent near the bottom of the "Changes" view or within the "Stacks" view.
    - **Why this step?** We're explicitly telling GitButler: "These selected changes belong together, and I want them to live on their own isolated development line, a virtual branch."
3. A prompt will appear asking for a branch name and a commit message.

- **Branch Name:** Enter `feat/initial-greeting` (GitButler often suggests names based on your message, which is a neat feature!).
- **Commit Message:** Enter `feat: Add initial greeting file` 4. Click "Commit" or "Create Branch and Commit."

Step 6: Observe the New Virtual Branch

1. Look at the **Stacks View**. You should now see a new entry: `feat/initial-greeting` with your commit message underneath it. This represents your first virtual branch.
2. Look at the **Changes View**. It should now be empty again, because your changes are safely committed to your *virtual* branch.

You've just made your first commit to a virtual branch in GitButler! How cool is that? Your `main` branch in your local Git repository remains untouched by this change for now.

Mini-Challenge: Stack Another Virtual Branch

You've seen how easy it is to create one virtual branch. Now, let's try stacking another one on top! This is a core GitButler pattern that truly unleashes its power.

Challenge: 1. In your `my-gitbutler-project` directory, open `hello-gitbutler.txt`. 2. Add a new line to the file, enhancing the greeting: `text Hello, GitButler! This is my first change using virtual branches. Adding a second line for a stacked change!` 3. Save the file. 4. Switch back to GitButler. You should see the modification to `hello-gitbutler.txt` in the **Changes View**. 5. Create a *new* virtual branch for this modification. Name it `refactor/enhance-greeting` and use the commit message `refactor: Enhance greeting with a second line`. 6. Commit the change.

Hint: The process is almost identical to what you just did for your first virtual branch. The key is to ensure you create a *new* virtual branch rather than amending the previous one. GitButler's UI will guide you.

What to Observe/Learn: After completing the challenge, go to the **Stacks View**. You should now see `refactor/enhance-greeting` stacked *on top* of `feat/initial-greeting`. This visual representation clearly shows how your changes are layered, with each virtual branch building upon the previous one. This is the essence of GitButler's stacked branch workflow! Notice how clean your `main` branch still is, even with two features actively being developed locally.

Common Pitfalls & Troubleshooting

Even with GitButler's friendly interface, you might encounter a few bumps. Here are some common ones and how to navigate them:

1. **"GitButler isn't detecting my changes!"**

- **Check:** Is your repository correctly added and selected in the GitButler sidebar? Make sure the correct project is highlighted.
- **Check:** Are your changes actually saved in your code editor? Unsaved changes won't appear.
- **Check:** Is the file within the repository's directory? GitButler only tracks files within the added repository.
- **Solution:** While GitButler usually refreshes automatically, a quick restart of the application can resolve minor synchronization issues. Also, ensure you haven't accidentally excluded the file or directory in a `.gitignore` file if it's an untracked file you expect to see. 2. **"I committed to `main` by accident in my terminal!"**
- **Explanation:** GitButler's virtual branches are a layer *above* your traditional Git branches. If you use `git commit` directly in your terminal, you're bypassing GitButler's workflow and committing directly to your active Git branch (e.g., `main`).
- **Solution:** It's best to adopt the GitButler flow for all local commits to fully leverage its benefits. If this happens, you can often use `git reset HEAD~1` in your terminal to undo the last commit on `main` and then let GitButler pick up the changes as uncommitted modifications. GitButler also has built-in tools to "uncommit" or move changes from `main` into a virtual branch, which we'll explore in a later chapter. 3. **"I'm confused about virtual branches vs. regular Git branches."**

- **Clarification:** A virtual branch is a *local-only* construct within GitButler. It's a way to organize your working copy, allowing you to iterate on changes with extreme flexibility. A regular Git branch (like `main` or a feature branch you push to GitHub) is a pointer to a commit in the Git history that is meant to be shared. GitButler helps you *build* and refine the commits that will eventually go onto regular Git branches, but it gives you far more flexibility *before* those commits are finalized and pushed. Think of virtual branches as your personal, highly malleable drafts that you can perfect before publishing.

Summary

Phew, you've taken some crucial first steps with GitButler! Let's recap what you've achieved:

- You've successfully **navigated the GitButler interface**, understanding its main sections like the sidebar, changes view, and stacks view.
- You learned how to **add an existing Git repository** to GitButler.
- You grasped the fundamental concept of **virtual branches** – local, flexible containers for your changes, distinct from traditional Git branches.
- You performed your first **guided exercise**, creating a file, making a change, and committing it to a new virtual branch.
- You tackled a **mini-challenge**, successfully stacking a second virtual branch and observing its placement in the Stacks View.
- We touched upon some **common pitfalls** and how to think about troubleshooting them, reinforcing the difference between GitButler's workflow and traditional Git.

You're now equipped with the basic interaction patterns of GitButler and have a foundational understanding of virtual branches. In the next chapter, we'll dive deeper into the power of **stacked branches** and explore how GitButler allows you to effortlessly reorder, edit, and manage these stacks, truly revolutionizing your development workflow!

References

- GitButler Official Website
- Getting Started - GitButler Docs
- Stacked Branches - GitButler Docs

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 05

Integrating with Remote: Pushing Stacks and Preparing for Pull Requests

Introduction

Welcome back, fellow developer! So far, we've explored the magic of GitButler locally, creating virtual branches, stacking changes, and managing our work with unparalleled flexibility. But what good is amazing local productivity if you can't share it with your team? This chapter is all about bridging that gap.

We'll dive into how GitButler seamlessly integrates with your remote repositories. You'll learn the crucial "Publish" action, which transforms your local virtual branch stacks into conventional Git branches on your remote. More importantly, we'll guide you through the process of preparing these published branches for pull requests, embracing the powerful concept of *stacked pull requests* that streamlines code review and collaboration.

By the end of this chapter, you'll be able to push your carefully crafted GitButler stacks to GitHub, GitLab, or any other Git host, and set up your pull requests for a smooth, efficient review cycle. Let's get your brilliant work out into the world!

Core Concepts: From Local Stacks to Remote Collaboration

Integrating your local GitButler workflow with a remote repository is where the real collaborative power shines. GitButler's approach simplifies what can often be a complex and error-prone process in traditional Git, especially when dealing with dependent changes.

GitButler's Remote Integration Philosophy

Unlike traditional Git where you manually manage which local branch tracks which remote branch, GitButler takes a more holistic view. When you "publish" a stack of virtual branches, GitButler intelligently creates corresponding *conventional* Git branches on your remote repository. This means your team can interact with your work using familiar Git commands and platforms, while you continue to enjoy GitButler's local superpowers.

The core idea is to let GitButler handle the underlying Git mechanics, allowing you to focus on your changes and their logical progression.

Publishing a Virtual Branch Stack

Imagine you've built a feature that required several small, dependent changes. In GitButler, these are individual virtual branches stacked on top of each other. When you're ready to share, you "publish" this entire stack.

What does "publish" mean? GitButler performs a series of Git operations behind the scenes:

1. **Creates Remote Branches:** For each virtual branch in your stack, GitButler creates a new, independent branch on your remote repository. These remote branches are named based on your virtual branch names (e.g., `feature/login-ui`, `feature/login-api`).
2. **Pushes Commits:** It pushes the commits from each virtual branch to its corresponding remote branch.
3. **Maintains Dependencies:** Crucially, it ensures that the commits reflecting the dependencies between your virtual branches are correctly represented in the remote branches.

This process is designed to be idempotent: you can publish multiple times, and GitButler will update the remote branches as needed, even handling cases where you've reordered or rebased your local stack.

Preparing for Stacked Pull Requests (PRs)

The true elegance of publishing stacks lies in how it facilitates *stacked pull requests*. Instead of one monolithic PR for a large feature, you create smaller, more digestible PRs, where each PR builds on the previous one.

Consider our example stack: * `refactor/auth-middleware` (base) * `feature/login-api` (depends on `refactor/auth-middleware`) * `feature/login-ui` (depends on `feature/login-api`)

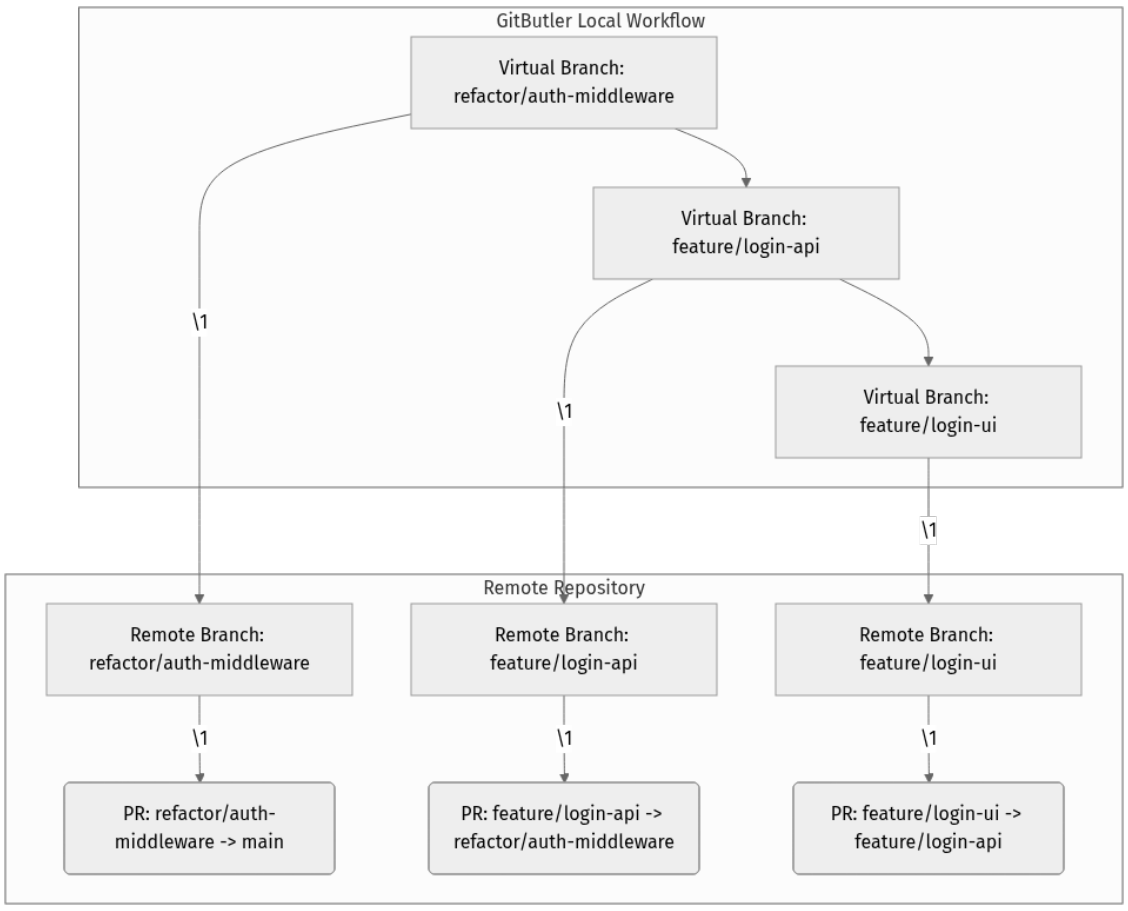
When published, these become three distinct remote branches. You would then create PRs like this:

1. **PR 1:** `refactor/auth-middleware` into `main` (or your target integration branch).
2. **PR 2:** `feature/login-api` into `refactor/auth-middleware`.
3. **PR 3:** `feature/login-ui` into `feature/login-api`.

This approach offers significant benefits:

- **Easier Review:** Each PR is smaller, focused on a single logical change, making it quicker and easier for reviewers to understand and approve.
- **Faster Feedback:** Reviewers can provide feedback on the base changes without waiting for the entire feature to be complete.
- **Improved Agility:** If a lower-level PR needs changes, you can address them, update the PR, and the subsequent PRs will automatically update (or require a simple rebase within GitButler) to reflect the changes.

Here's a visual representation of how a local GitButler stack translates to remote branches and stacked PRs:



Keeping Remote in Sync

Once published, your virtual branches are linked to their remote counterparts. Any further changes you make locally to a virtual branch that has been published will be reflected on the remote branch the next time you publish. GitButler is smart enough to handle the updates, including any necessary rebases or force pushes (which it manages safely for you).

Step-by-Step Implementation: Publishing and PR Prep

Let's put these concepts into practice. We'll assume you have an existing GitButler repository with a few stacked virtual branches. If not, quickly create a new repo, commit to `main`, then create `feature/first-step`, add a commit, and then `feature/second-step` on top, adding another commit.

Step 1: Reviewing Your Local Stack

Before publishing, it's always a good practice to review your current work.

1. **Open GitButler:** Ensure you have your repository open in the GitButler desktop application.
2. **Inspect the "Active Branches" View:** Look at the left-hand panel where your virtual branches are listed. Verify that the order and content of your branches are as you intend. You can click on each branch to see its commits.

   Self-reflection: Are the commit messages clear? Is each branch focused on a single logical change? This is your last chance to easily tweak things locally before pushing to remote!

Step 2: Publishing Your Stack to Remote

Now, let's send your work to the remote.

1. **Locate the "Publish" Button:** In the GitButler UI, usually at the top right of the "Active Branches" panel, you'll find a "Publish" button. It might also indicate how many branches are "ready to publish."
2. **Click "Publish":**
   - GitButler will analyze your local virtual branches and compare them with the remote.
   - It will then perform the necessary Git operations (creating remote branches, pushing commits) to synchronize your stack.
   - You'll see a progress indicator as it works.

   GitButler Publish Button Example *Image Source: GitButler Official Documentation (https://docs.gitbutler.com/guide/getting-started/publish-to-remote)*

   What's Happening Under the Hood? If you were to open your terminal and run `git branch -r`, you would see new remote branches created for each of your virtual branches. For example, if you had `feature/login-api` and

`feature/login-ui` locally, you'd now see `origin/feature/login-api` and `origin/feature/login-ui`.

Step 3: Verifying Remote Branches

It's always good to confirm your work has landed on the remote.

1. **Visit Your Git Host:** Open your web browser and navigate to your repository on GitHub, GitLab, Bitbucket, or your self-hosted Git server.
2. **Check the Branches Tab:** Look for the "Branches" section of your repository. You should now see the new branches corresponding to your GitButler virtual branches.

   Observation: Notice that these are regular Git branches, just like any other. Your team members can now `git fetch` and `git checkout` these branches.

Step 4: Creating Stacked Pull Requests

With your branches on the remote, it's time to initiate the review process.

1. **Start with the Base Branch:** On your Git host, find the *lowest* branch in your published stack (e.g., `refactor/auth-middleware`).
2. **Create a Pull Request:**
   - Initiate a new pull request for this branch.
   - **Crucially, set its target branch to `main`** (or `master`, or your team's primary integration branch).
   - Give it a clear title and description.
3. **Move Up the Stack:**
   - Next, find the *next* branch in your stack (e.g., `feature/login-api`).
   - Create a pull request for this branch.
   - **Set its target branch to the *previously created remote branch* (`refactor/auth-middleware`)**. This is the essence of stacked PRs!
   - Repeat this process for all branches in your stack, always targeting the branch immediately below it in the stack.

   GitButler's Helper Links: GitButler often provides convenient links in its UI to directly open your Git host's "create pull request" page for a specific published branch. Look for an icon or text like "Create PR" next to a published branch. Clicking this will pre-fill some of the PR details, making the process even smoother.

Step 5: Updating a Published Stack

Work isn't always linear! You'll often need to make changes to a virtual branch that's already been published and has an open PR.

1. **Make Local Changes:** In GitButler, switch to the virtual branch you want to modify (e.g., `feature/login-api`). Make your changes, commit them, and ensure your virtual branch looks correct locally.
2. **Re-Publish:** Click the "Publish" button again. GitButler will detect the changes on your local virtual branch and push them to the corresponding remote branch.

   What happens on the Remote? Your open pull request will automatically update with the new commits. If you rebased or reordered commits, GitButler will handle the necessary force push to the remote branch, ensuring the remote history matches your cleaned-up local history. This is one of GitButler's most powerful features – it manages the complex Git operations so you don't have to.

Mini-Challenge: Publishing a Nested Stack

Let's solidify your understanding with a practical challenge.

Challenge:

1. In your GitButler repository, create a new virtual branch called `feat/user-settings`. Add a commit with a simple change (e.g., "Add settings page placeholder").
2. On top of `feat/user-settings`, create another virtual branch named `feat/user-profile-avatar`. Add a commit that simulates adding avatar upload logic.
3. Publish this two-branch stack to your remote repository.
4. Verify that both `feat/user-settings` and `feat/user-profile-avatar` branches now exist on your Git host.
5. Create a pull request for `feat/user-settings` targeting `main`.
6. Create a second pull request for `feat/user-profile-avatar` targeting `feat/user-settings`.
7. Go back to GitButler, make a small *additional* change to `feat/user-settings` (e.g., "Fix typo on settings page"). Commit it.
8. Re-publish your stack.

9. Observe how the `feat/user-settings` PR on your Git host automatically updates with the new commit.

Hint: Pay close attention to the branch selection when creating your pull requests on your Git host. The target branch is key for stacked PRs!

What to observe/learn: This exercise demonstrates the seamless flow from local GitButler work to remote collaboration, and how GitButler handles updates to published, stacked work. You'll see how easy it is to keep your PRs fresh even after making further changes to lower branches in your stack.

Common Pitfalls & Troubleshooting

Even with GitButler simplifying things, understanding common scenarios helps.

1. Confusing Virtual Branches with Remote Branches

Pitfall: Thinking that GitButler's virtual branches *are* the remote branches. **Explanation:** GitButler's virtual branches are a local construct. When you "publish," GitButler *creates* or *updates* conventional Git branches on your remote repository that *represent* your virtual branches. They are distinct but managed by GitButler to stay in sync. **Troubleshooting:** Always remember that your team interacts with the *remote Git branches*, while you interact with *local GitButler virtual branches*. GitButler acts as the bridge.

2. Unexpected Remote Branch Naming

Pitfall: Your remote branch names don't match your virtual branch names, or they have unexpected prefixes. **Explanation:** By default, GitButler uses your virtual branch name as the remote branch name. However, some teams or Git host configurations might have specific naming conventions or automated prefixes. **Troubleshooting:** Check your GitButler settings (if any advanced remote naming is configured). More commonly, simply be aware of the exact names GitButler pushes to the remote, and use those when creating PRs. If you need to rename a remote branch, you might need to do it manually on your Git host or use `git push origin :old-name new-name` after renaming locally in GitButler and re-publishing.

3. Merge Conflicts When Creating PRs

Pitfall: You create a PR for your base branch (e.g., `refactor/auth-middleware` into `main`), and your Git host reports merge conflicts. **Explanation:** This means that since you branched off `main` (or your base branch), changes have been

merged into `main` that conflict with your `refactor/auth-middleware` branch. This is a common Git scenario. **Troubleshooting:** 1. **Fetch Latest `main`:** In GitButler, ensure your `main` branch is up-to-date by clicking the "Fetch" button. 2. **Rebase Your Base Virtual Branch:** Switch to your `refactor/auth-middleware` virtual branch. Then, rebase it onto the latest `main`. GitButler provides easy ways to do this through its UI (e.g., dragging the branch onto `main` or using a rebase option). Resolve any conflicts locally within GitButler. 3. **Re-publish:** After resolving conflicts and ensuring your local stack is clean, re-publish. GitButler will handle the force push to update the remote branch, and your PR will then reflect the rebased changes and hopefully be conflict-free.

Summary

You've now mastered the critical step of integrating your local GitButler workflow with your remote repository!

Here's a quick recap of the key takeaways:

- **Publishing Stacks:** GitButler's "Publish" action pushes your local virtual branch stacks to your remote repository as conventional Git branches.
- **Stacked Pull Requests:** This enables a powerful workflow where you create individual PRs for each logical change in your stack, with each PR targeting the previous one, leading to easier reviews and faster feedback.
- **Seamless Updates:** GitButler intelligently manages updates to published branches, handling rebases and force pushes behind the scenes, so your open PRs stay current.
- **Bridging Local and Remote:** GitButler acts as a smart layer, allowing you to enjoy flexible local development while seamlessly collaborating with a team using standard Git tools.

Next up, we'll delve deeper into handling feedback, merging your stacked PRs, and advanced scenarios like handling concurrent work on shared branches. Get ready to truly unlock collaborative power!

References

- GitButler Official Website: https://gitbutler.com/
- Getting Started - GitButler Docs: https://docs.gitbutler.com/guide

- Stacked Branches - GitButler Docs: https://docs.gitbutler.com/guide/features/branch-management/stacked-branches

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 06

Mastering Your Changes: Interactive Commits and Local History Management

Introduction

Welcome back, future GitButler master! In our previous chapters, you've learned the magic of virtual branches and how they help you isolate your work. But what happens after you've made a bunch of changes on a virtual branch? Often, our initial coding spree results in a messy mix of refactors, new features, bug fixes, and maybe even a typo correction or two, all tangled together.

This is where GitButler truly shines! This chapter is all about transforming that raw, unorganized work into a pristine, easy-to-understand commit history. We'll dive deep into GitButler's interactive tools that let you craft atomic commits, amend mistakes, reorder your work, and squash related changes – all without ever touching the dreaded `git rebase -i` command line.

By the end of this chapter, you'll not only understand *how* to use these powerful features but also *why* a clean commit history is invaluable for code reviews, debugging, and maintaining a healthy codebase. Get ready to elevate your Git game!

To follow along, you should have GitButler installed and be familiar with creating and switching between virtual branches, as covered in earlier chapters.

Core Concepts

Before we jump into the hands-on steps, let's unpack the core ideas behind interactive commits and local history management in GitButler. These concepts are designed to solve common frustrations developers face with traditional Git.

The Power of Interactive Commits

Imagine you're working on a file, and you make a small refactor, then add a new feature, and finally fix a bug. In traditional Git, you might `git add .` and commit everything, resulting in a single, large commit that's hard to review and understand. Or, you might painstakingly use `git add -p` to stage specific hunks (parts of changes), which can be tedious and error-prone.

GitButler offers a visual, intuitive approach to "interactive staging." Instead of staging entire files or using cryptic command-line options, GitButler presents your changes in a clear diff view. You can then effortlessly select specific lines or "hunks" of code and group them into logical, "atomic" commits.

What's an "atomic commit"? It's a commit that represents a single, independent, logical change. For example, "Refactor database connection," "Add user authentication endpoint," or "Fix typo in README." Atomic commits make it much easier for reviewers to understand your changes and for you to revert specific features or fixes if needed.

Here's a conceptual overview of how GitButler helps you transform messy changes into clean commits:

Understanding Local History and Amending Commits

Once you've made a commit, what if you realize you forgot a tiny detail, made a typo in the commit message, or left out a crucial line of code? In traditional Git, you'd use `git commit --amend`. While powerful, it can be intimidating for newcomers.

GitButler simplifies this by making your local commit history a living, editable canvas. When you make new changes that logically belong to an *existing* commit (especially the most recent one), GitButler allows you to easily "drag" those new changes onto the existing commit to amend it. This means you can refine your commits even after they've been created, ensuring they tell a complete and accurate story.

Why is amending useful? It helps keep your commit history clean by preventing "fixup" commits like "Oops, forgot a semi-colon" or "Add missing import." Instead, these small corrections become part of the original, logical commit.

Reordering and Squashing (The GitButler Way)

The ultimate tool for shaping a clean commit history in traditional Git is `git rebase -i` (interactive rebase). It allows you to reorder, squash, edit, and drop commits. However, `git rebase -i` is notoriously difficult to master, often leading to frustration and accidental history corruption for many developers.

GitButler completely reimagines this process with a visual, drag-and-drop interface:

- **Reordering:** If you have commits A, B, and C, but you realize B should logically come before A, you can simply drag commit B above commit A in GitButler's history view. GitButler intelligently re-applies the changes in the new order, handling any potential conflicts gracefully.
- **Squashing:** If you have two or more consecutive commits that logically form a single unit (e.g., "Implement feature part 1" and "Implement feature part 2"), you can drag one commit onto another to "squash" them into a single, comprehensive commit. GitButler will prompt you to combine their commit messages.

These operations are performed on your *local* virtual branch, giving you the freedom to experiment and refine your history before sharing it with others. This "local-first" approach is a cornerstone of GitButler's philosophy.

Step-by-Step Implementation: Crafting Your History

Let's get practical! We'll simulate a common development scenario where changes are made somewhat haphazardly, and then we'll use GitButler to clean them up.

Prerequisite: Ensure you have a Git repository open in GitButler and are on a virtual branch (e.g., `feature/my-awesome-feature`). If you don't have one, create a new repository and a new virtual branch now.

Step 1: Making Disorganized Changes

First, let's create some mixed changes in a file. Imagine you're refactoring a utility function and also adding a new logging mechanism.

1. **Open your project in your favorite IDE.**
2. **Create a new file** named `utils.js` (or any other file if you have an existing project) in your repository's root.
3. **Add the following content** to `utils.js`:

   ``javascript // utils.js /** * @deprecated Use formatMessage instead for better logging context. */ function oldFormat(message) { return [OLD] ${message}`; }

   function formatMessage(context, message) { // This is part of a new logging feature const timestamp = new Date().toISOString(); return `[${timestamp}] [${context}] ${message}`; }

// A small helper for string manipulation function capitalize(str) { if (!str) return ''; return str.charAt(0).toUpperCase() + str.slice(1); }

// Export the new function module.exports = { formatMessage, capitalize }; ```

Here, we've got: * A deprecation comment for `oldFormat` (refactor). * A new `formatMessage` function (new feature). * A `capitalize` helper (another new feature/utility). * Exporting the new functions.

These changes are conceptually distinct. Let's see how GitButler helps us separate them.

Step 2: Staging Interactively with GitButler

Now, open your GitButler desktop application. You should see your `feature/my-awesome-feature` branch selected, and the changes in `utils.js` listed under "Changes in working directory."

1. **Click on the `utils.js` file** in the GitButler UI to open its diff view.
2. You'll see all your added lines highlighted in green.
3. **Identify the "refactoring" part:** This is primarily the deprecation comment for `oldFormat`.
4. **Select the deprecation comment:** Hover over the line `/**` preceding `function oldFormat(...)`. You'll see a small `+` icon or a checkbox appear next to the line number. Click it. GitButler will highlight the entire hunk (or just that line, depending on granularity) as selected.
5. **Create the first atomic commit:**
    - In the "Commit" panel (usually on the right), enter a commit message like: `refactor: Deprecate oldFormat function`.
    - Click the **"Commit" button**.

    Observe: GitButler has now created your first commit, and the selected lines are gone from "Changes in working directory." The remaining lines in `utils.js` are still unstaged.

Step 3: Creating the Second Atomic Commit

Let's commit the `formatMessage` function as our second logical change.

1. **Back in the `utils.js` diff view:** You should still see the remaining changes.
2. **Select the `formatMessage` function:** Click to select all lines related to the `formatMessage` function (from its definition to its export).

3. **Create the second atomic commit:**
    - Enter a commit message: `feat: Add new formatMessage utility for structured logging`.
    - Click the **"Commit" button**.

    Now you have two distinct commits on your virtual branch!

Step 4: Amending a Commit

Oh no! You realize you forgot to add a JSDoc comment to the `capitalize` function, which you committed in the previous step (along with `formatMessage`). Instead of a new "fixup" commit, let's amend the previous commit.

1. **Modify `utils.js` again:** Add a JSDoc comment for `capitalize`.

    ```javascript // utils.js // ... (previous code) ...

    /* * *Capitalizes the first letter of a string. * @param {string} str The input string. * @returns {string} The capitalized string.* / function capitalize(str) { if (!str) return ''; return str.charAt(0).toUpperCase() + str.slice(1); }

    // ... (rest of the code) ... ```
2. **Back in GitButler:** You'll see the new changes (the JSDoc comment) under "Changes in working directory."
3. **Open the diff for `utils.js` again.**
4. **Select the new JSDoc comment lines.**
5. **Look at your commit history:** On the left panel, you'll see your two commits. The most recent one is `feat: Add new formatMessage utility...`.
6. **Drag the selected changes from the diff view directly onto the `feat: Add new formatMessage...` commit** in the commit history panel.
7. GitButler will ask you to confirm the amend. Confirm it.

    Observe: The JSDoc changes are now merged into your *existing* `feat: Add new formatMessage...` commit. Your commit history remains clean, and you avoided a separate "fix" commit. This is incredibly powerful for keeping your work focused!
    ```

Step 5: Reordering Commits

Let's say you also added another small utility, `slugify`, and committed it, but then realized it would make more sense *before* the `capitalize` function (which was part of the `formatMessage` commit).

1. **Modify `utils.js` one more time:** Add a `slugify` function.

   ```javascript // utils.js // ... (previous code including capitalize) ...

   /* * *Converts a string to a URL-friendly slug. * @param {string} str The input string. * @returns {string} The slugified string.* / function slugify(str) { if (!str) return ''; return str .toLowerCase() .trim() .replace(/[^\w\s-]/g, '') // Remove non-word chars .replace(/[\s_-]+/g, '-') // Replace spaces/underscores with single dash .replace(/^-+|-+$/g, ''); // Trim dashes }

   // Export the new function module.exports = { formatMessage, capitalize, slugify // Don't forget to export! }; ```

2. **In GitButler:** Select the `slugify` function and its export in the diff view.

3. **Commit it:** Enter commit message `feat: Add slugify utility`.

   Now your history might look something like: * `refactor: Deprecate oldFormat function` * `feat: Add new formatMessage utility...` (includes `capitalize` and `formatMessage`) * `feat: Add slugify utility`

   But you want `slugify` to appear before `formatMessage` because it's a more fundamental utility.

4. **In the GitButler commit history panel (left side):**

   - **Drag the `feat: Add slugify utility` commit** and drop it *above* the `feat: Add new formatMessage...` commit.

5. GitButler will process the reorder. If there are no conflicts, it will happen instantly. If there are conflicts (unlikely in this simple case), it would guide you through resolving them.

   Observe: Your commits are now reordered, and `slugify` appears before `formatMessage` in the history, making more logical sense.
   ```

Step 6: Squashing Commits

Let's say you made a small "chore" commit to fix linting errors, but it's directly followed by the actual feature commit. These two logically belong together.

1. **Modify `utils.js` with a small, trivial change:** Add an extra space or change a single quote to a double quote, simulating a linting fix. `javascript // utils.js // ... function capitalize(str) { if (!str) return ''; // Changed to single quote return str.charAt(0).toUpperCase() + str.slice(1); } // ...`
2. **In GitButler:** Commit this change with the message `chore: Fix linting in utils.js`.
3. Now your history has:
    - `refactor: Deprecate oldFormat function`
    - `feat: Add slugify utility`
    - `feat: Add new formatMessage utility...`
    - `chore: Fix linting in utils.js`

    You want to squash `chore: Fix linting in utils.js` into `feat: Add new formatMessage utility...`.
4. **In the GitButler commit history panel:**
    - **Drag the `chore: Fix linting in utils.js` commit** and drop it *onto* the `feat: Add new formatMessage utility...` commit.
5. GitButler will present a dialog asking you to combine the commit messages. You can choose to keep both, edit them, or just keep one. For squashing, it's common to merge them or refine the message.
    - For example, you might combine them to: `feat: Add new formatMessage utility and linting fix`.
6. Confirm the squash.

    Observe: The `chore` commit is gone, and its changes (and message) are now part of the `feat: Add new formatMessage utility...` commit. Your history is even cleaner!

Mini-Challenge

You've done great so far! Let's solidify your understanding with a small challenge.

Challenge:

1. **Create a new virtual branch** called `challenge/feature-enhancement`.
2. **In a new file** named `data_processor.js`, add the following content in one go:

   ```javascript // data_processor.js function processData(data) { // Initial setup for processing if (!data) return []; let processed = data.map(item => item.value * 2);

   ```
   // Add a new filtering step
   processed = processed.filter(item => item > 10);

   // Another transformation
   processed = processed.map(item => ({ result: item, status:
   'processed' }));

   return processed;
   ```

   } // Export for use module.exports = { processData }; ``` 3. **Use GitButler's interactive staging** to split these changes into *two* distinct, atomic commits: * The first commit should be: feat: Implement basic data processing logic (covering the initial setup and map(item => item.value * 2)). * The second commit should be: feat: Add filtering and final transformation steps (covering the filter and final map operations). 4. **Realize you forgot to add a comment** explaining the initial setup. Go back to data_processor.js and add // Ensure data exists and transform initial values above if (!data) return []; . 5. **Amend the *first* commit** ( feat: Implement basic data processing logic`) with this new comment.

**Hint:** Pay close attention to the diff view in GitButler and how you can select individual lines or hunks. Remember to drag and drop changes from the "Changes in working directory" panel onto the specific commit you want to amend in the history panel.

**What to Observe/Learn:** How easy it is to refine your commits even after you've initially created them, preventing "fixup" commits and keeping your history pristine.
   ```

Common Pitfalls & Troubleshooting

Even with GitButler's intuitive interface, it's good to be aware of some common scenarios and how to navigate them.

1. **Forgetting to Stage Interactively (Committing Everything):**

- **Pitfall:** You might get used to `git commit -m "..."` and accidentally commit all changes in your working directory at once, leading to a large, non-atomic commit.
- **Solution:** No worries! GitButler's history management tools are your friends. You can use the "Split Commit" feature (right-click on a commit in the history) to break it down. Alternatively, you can create new commits for the parts that should be separate, then reorder and squash them as needed. The key is that your local history is mutable.

1. **Amending a Published Commit:**

- **Pitfall:** While GitButler encourages amending local commits, remember that amending *rewrites history*. If you've already pushed a commit to a remote repository and then amend it locally, you'll create a different history. Pushing this amended history would require a `git push --force`, which can cause problems for collaborators who have already pulled the original commit.
- **Best Practice:** Always amend commits *before* pushing them to a shared remote. GitButler's local-first approach makes this natural, as you're refining your virtual branch before publishing it to a stack.

1. **Merge Conflicts During Reordering/Squashing:**

- **Pitfall:** When you reorder or squash commits, GitButler is essentially re-applying changes in a new sequence. If two commits modify the same lines of code in conflicting ways, a merge conflict can occur.
- **Troubleshooting:** GitButler will clearly indicate when a conflict arises. It will pause the operation and provide a visual conflict resolution interface, similar to what you might find in your IDE. You'll need to manually resolve the conflicting lines, mark the conflict as resolved, and then continue the operation. Don't panic; conflicts are a normal part of Git and GitButler provides good tools to handle them.

Summary

Congratulations! You've just unlocked some of the most powerful features of GitButler for managing your local changes and crafting an impeccable commit history.

Here are the key takeaways from this chapter:

- **Interactive Staging:** GitButler allows you to visually select specific lines or hunks of code from your working directory to form atomic, logical commits, moving beyond the limitations of `git add -p`.
- **Amending Commits:** You can easily incorporate small corrections or forgotten details into existing commits by dragging changes onto them in the history view, avoiding unnecessary "fixup" commits.
- **Reordering Commits:** GitButler provides a drag-and-drop interface to logically reorder commits in your local history, making it simple to present changes in the most sensible sequence.
- **Squashing Commits:** Combine multiple related commits into a single, more comprehensive commit by dragging one onto another, streamlining your history for easier review.
- **Local-First Philosophy:** All these operations are performed on your local virtual branch, giving you the freedom to perfect your work before sharing it.

Mastering these techniques will significantly improve your development workflow, lead to clearer code reviews, and make you a more confident Git user. You're no longer just making commits; you're *crafting* history!

In the next chapter, we'll explore how to take these perfectly crafted virtual branches and integrate them into a collaborative workflow using GitButler's unique "stacked branches" feature. Get ready to share your amazing work!

References

- GitButler Official Website
- GitButler Docs - Getting Started
- GitButler Docs - Branch Management: Stacked Branches
- Pro Git Book - Rewriting History (For traditional Git context on rebase and amend)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 07

The Power of Virtual Branches: Isolating Your Development Work

Introduction

Welcome back, future Git master! In the previous chapter, we got GitButler up and running and connected our first repository. Now, it's time to dive into the very heart of what makes GitButler so revolutionary: **virtual branches**.

Think about your current Git workflow. How often do you find yourself needing to switch contexts, stash changes, or deal with a cluttered local repository because you're working on multiple things at once? Traditional Git branches are powerful, but they can sometimes feel clunky for managing rapid, iterative local development. GitButler's virtual branches are designed to solve exactly these pain points, offering an unparalleled way to isolate your work, experiment freely, and keep your local repository pristine.

In this chapter, we'll explore what virtual branches are, how they differ fundamentally from traditional Git branches, and most importantly, how to use them to supercharge your development isolation. You'll learn to create, manage, and switch between these powerful local workspaces, setting the stage for more advanced workflows like stacked changes.

What Are Virtual Branches?

At its core, GitButler introduces the concept of **virtual branches**. These are local-only branches that exist *within* GitButler, distinct from the branches that Git itself tracks (like `main`, `develop`, or `feature/xyz`). They are like temporary, highly flexible workspaces that allow you to segment your work without immediately touching your underlying Git repository's branch structure.

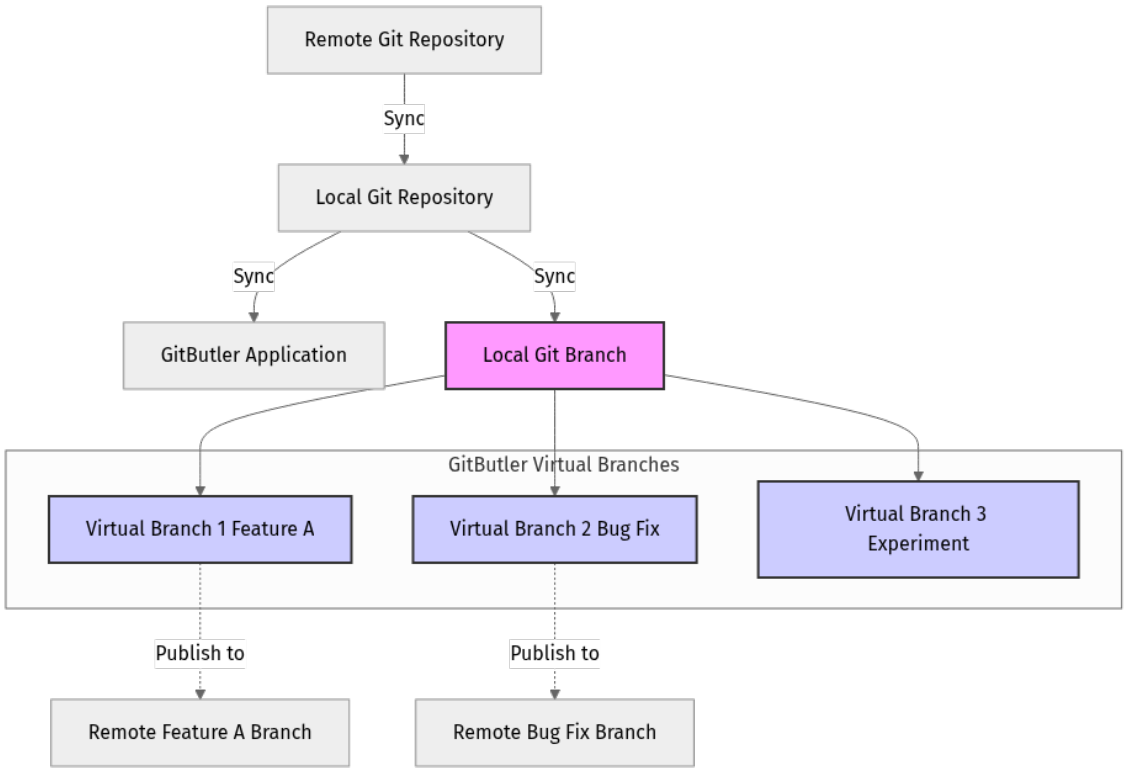
Imagine you're a chef. Traditional Git branches are like having multiple, large, dedicated kitchens, each for a different recipe. To switch recipes, you literally have to pack up one kitchen and move to another. Virtual branches, on the other hand, are like having one main kitchen, but with several small, independent cutting boards and prep stations. You can switch between preparing different ingredients on different boards instantly, without affecting the main kitchen or other prep stations.

Key Characteristics of Virtual Branches:

1. **Local-First Isolation:** Virtual branches are entirely local to your machine and GitButler. Changes you make on a virtual branch are isolated from your active Git branch (e.g., `main`) until you explicitly choose to publish them. This means you can experiment, break things, and commit frequently without worrying about polluting your main development line or remote repository.
2. **Instant Context Switching:** Switching between virtual branches is incredibly fast and seamless. GitButler handles the underlying Git operations (like `git stash` and `git checkout`) for you, allowing you to jump from one task to another with a single click, preserving your work-in-progress on each.
3. **No Git Branch Clutter (Initially):** You can have dozens of virtual branches in GitButler without creating a single new branch in your actual Git repository. This keeps your `git branch` output clean and focused on shared, published branches.
4. **Foundation for Stacking:** Virtual branches are the building blocks for GitButler's stacked changes feature, which we'll explore in the next chapter.

Virtual Branches vs. Traditional Git Branches

Let's clarify the distinction with a simple diagram:



Explanation of the Diagram: * Your **Remote Git Repository** (A) is where your team's shared code lives. * Your **Local Git Repository** (B) is the clone on your machine, synchronized with the remote. * **GitButler Application** (C) interacts with your local Git repository. * Your **Local Git Branch (e.g., main)** is the actual Git branch that GitButler is currently "pointing" to for its base. * **GitButler Virtual Branches** (VB1, VB2, VB3) are distinct, isolated workspaces managed *by GitButler*. They exist on top of your current local Git branch (often `main` or `develop`). * When you **publish** a virtual branch, GitButler creates a corresponding *actual* Git branch in your local repository and pushes it to the remote, allowing you to share your work.

Why This Matters: The "Local-First" Workflow

GitButler champions a "local-first" approach. This means you do all your development, commit frequently, refactor, and experiment *locally* within virtual branches. Only when a feature or fix is stable and ready for review do you "publish" it to a traditional Git branch that can be pushed to a remote and shared with your team. This significantly reduces the friction of managing work-in-progress and encourages a more fluid, less stressful development process.

Step-by-Step: Managing Virtual Branches

Let's get hands-on and experience the power of virtual branches!

1. Open Your Repository in GitButler

If you closed GitButler, open it up again. You should see your connected repository from the previous chapter. Click on the repository name to open its workspace.

You'll likely see one virtual branch already active, named after your current Git branch (e.g., `main` or `master`). This is GitButler's way of representing your current working state.

2. Create Your First Virtual Branch

Let's create a new virtual branch to work on a hypothetical new feature.

1. Look for the **"New Virtual Branch"** button in the GitButler interface (usually prominent, perhaps with a `+` icon).
2. Click it. A dialog will appear asking for a branch name.
3. Enter a descriptive name, like `feature/add-user-profile` or `fix/login-bug`. For this example, let's use `feature/welcome-message`.
4. Click **"Create"**.

> **What just happened?** GitButler has created a new, empty virtual branch for you. Notice how it becomes the active branch in the UI. Your working directory is now "clean" and ready for new changes, isolated from your `main` branch.

3. Make Changes and Commit on the Virtual Branch

Now, let's make a simple change in our project.

1. Open your project in your preferred code editor (VS Code, Sublime Text, etc.).
2. Locate a file to make a small, non-breaking change. For example, if you have an `index.html` or `App.js` file, add a new line of text or a comment.

   Let's assume you have an `index.html` file. Add a simple paragraph:

   ```
   html <!DOCTYPE html> <html lang="en"> <head> <meta charset="UTF-8"> <meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>My Awesome Project</title> </head> <body> <h1>Hello GitButler!</h1> <!-- Add this line below the h1 --> <p>This is a new welcome message from a virtual branch.</p> </body> </html>
   ```

3. Save the file.
4. Switch back to the GitButler application. You should now see your changes listed under the **"Uncommitted Changes"** section for your active virtual branch (`feature/welcome-message`).
5. **Stage Your Changes:** Click the `+` icon next to the modified file(s) to stage them, or use the "Stage All" button.
6. **Commit Your Changes:**
   - Enter a clear commit message in the "Commit Message" box, e.g., `feat: add welcome message to index page`.
   - Click the **"Commit"** button.

> **What did you observe?** Your changes are now committed, but *only* within your `feature/welcome-message` virtual branch. If you were to open your terminal and run `git log`, you wouldn't see this commit yet on your `main` branch. This is the power of local isolation!

4. Switch Between Virtual Branches

This is where the magic of instant context switching comes in.

1. On the left sidebar of GitButler, you'll see a list of your virtual branches. Click on your `main` virtual branch (or whatever your base branch is named).
2. GitButler will quickly switch your working directory.
3. Go back to your code editor. > **What do you see?** > The `p` tag you added earlier is gone! Your `index.html` file has reverted to its state when `main` was last active.
4. Now, switch back to your `feature/welcome-message` virtual branch in GitButler.
5. Return to your code editor. > **And just like that, your `p` tag is back!** This seamless switching without manual stashing or checking out is a huge time-saver.

5. Publish Your Virtual Branch to a Remote Git Branch

Eventually, you'll want to share your work with the world (or at least your team). This is when you "publish" your virtual branch.

1. Ensure your `feature/welcome-message` virtual branch is active.
2. On the virtual branch's card in the GitButler UI, look for an option like **"Publish"** or **"Create & Publish Git Branch"**.
3. Click this button. GitButler will ask you to confirm the name of the *new Git branch* that will be created both locally and on your remote repository. It will typically suggest the same name as your virtual branch.
4. Confirm the name and click **"Publish"**.

> **What just happened?** GitButler performed a `git branch` and `git push` operation behind the scenes. If you now open your terminal and run `git branch`, you'll see `feature/welcome-message` listed. If you check your remote repository (e.g., GitHub, GitLab, Bitbucket), you'll see `feature/welcome-message` has been pushed, containing your commit. Your virtual branch is now linked to a traditional Git branch!

Mini-Challenge: Multi-Tasking with Virtual Branches

Let's put your new knowledge to the test.

Challenge: You need to work on two separate, unrelated tasks concurrently: 1. A new feature that adds a `<footer>` to your HTML file. 2. A quick bug fix that changes the `<h1>` text.

Using only GitButler's virtual branches:

1. Create a virtual branch named `feature/add-footer`.
2. On `feature/add-footer`, add a simple `<footer>` tag to your `index.html` (or another suitable file) with some text, then commit it.
3. Switch to your `main` virtual branch.
4. Create a *second* virtual branch named `fix/update-heading`.
5. On `fix/update-heading`, change the text inside your `<h1>` tag (e.g., from "Hello GitButler!" to "Welcome to My Project!"), then commit it.
6. Verify that when you switch between `feature/add-footer`, `fix/update-heading`, and `main`, your code editor reflects only the changes relevant to the active virtual branch.

Hint: Remember to save your file after making changes and use the "Stage All" and "Commit" buttons in GitButler for each virtual branch.

What to Observe/Learn: * How easily you can switch between completely different sets of changes. * The complete isolation of work-in-progress on each virtual branch. * The speed and efficiency compared to `git stash` and `git checkout` manually.

Common Pitfalls & Troubleshooting

Even with GitButler simplifying things, a few common scenarios can cause confusion:

1. **Forgetting to Publish:** Your work is *only* local until you explicitly publish a virtual branch. If you close GitButler and expect to see your new branch on GitHub, you won't unless you published it.

- **Solution:** Always remember to use the "Publish" button when your work is ready to be shared or backed up to a remote. 2. **Confusion Between Virtual and Git Branches:** It's easy to forget that a virtual branch isn't a "real" Git branch until published. If you run `git branch` in your terminal, you won't see your new virtual branches until you've published them.
- **Solution:** Treat virtual branches as your personal, local workspaces. Only when you're ready for team collaboration or remote backup does it become

a traditional Git branch. GitButler clearly differentiates between "Virtual Branches" and "Published Branches" in its UI. 3. **Accidentally Committing to the Wrong Virtual Branch:** If you're not paying attention to which virtual branch is active in GitButler, you might make changes and commit them to the wrong one.

- **Solution:** Always double-check the active virtual branch in the GitButler UI before making significant changes or committing. GitButler prominently displays the active branch name. If you do commit to the wrong one, GitButler allows you to move commits between virtual branches, a powerful feature we'll explore later!

Summary

You've just taken a massive leap in optimizing your Git workflow by understanding and utilizing GitButler's virtual branches!

Here are the key takeaways from this chapter:

- **Virtual branches** are local, isolated workspaces within GitButler, distinct from traditional Git branches.
- They enable **rapid context switching** and **local-first development**, allowing you to commit frequently without cluttering your main Git history.
- Changes made on a virtual branch are **isolated** until you explicitly **publish** them to a remote Git branch.
- This approach significantly reduces the need for manual `git stash` and `git checkout` operations, streamlining your local development experience.

You're now equipped with the fundamental building block of GitButler's power. In the next chapter, we'll build upon this by exploring **stacked branches** – a powerful technique that allows you to organize dependent changes into a clean, reviewable sequence, taking your Git workflow to an entirely new level!

References

- GitButler Official Website
- Getting Started - GitButler Documentation
- Stacked Branches - GitButler Documentation
- Git Branching - Basic Branching and Merging (Pro Git Book)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 08

Practical Application: Developing a Feature with Stacked Branches

Introduction: Building Features, Layer by Layer

Welcome back, fellow developer! In our previous chapters, we laid the groundwork by understanding GitButler's core concepts like virtual branches and its local-first approach. Now, it's time to put that knowledge into action and tackle a common development challenge: building a significant feature that naturally breaks down into smaller, dependent steps.

Imagine you're tasked with adding a new "User Profile" section to an application. This isn't a single change; it often involves updating the database, modifying API endpoints, and finally, updating the user interface. Traditionally, managing these interdependent changes with Git can become a tangle of `git rebase -i` commands, temporary branches, and constant fear of breaking something.

This chapter will guide you through developing such a feature using GitButler's powerful stacked branches. You'll learn how to break down complex work into logical, manageable layers, making your development process smoother, your code reviews easier, and your Git history cleaner. By the end, you'll have hands-on experience in building a feature incrementally and observing how GitButler simplifies the entire workflow.

Core Concepts: Understanding Stacked Branches for Feature Development

Before we dive into code, let's solidify our understanding of what stacked branches are and why they're a game-changer for feature development.

What are Stacked Branches?

Think of stacked branches like building blocks, or layers of a delicious cake! Each block (or layer) represents a distinct, logical change that builds directly on the one below it. In the context of GitButler, a "stack" is a sequence of virtual branches where each branch is based on the previous one in the sequence.

Instead of having one massive branch for an entire feature, you create smaller, focused virtual branches for each logical step. For example, if you're adding a new user setting, your stack might look like this:

1. **Base Branch (e.g., `main` or `develop`):** The stable foundation.
2. **`feat/user-settings-db`:** Adds the necessary database schema changes.
3. **`feat/user-settings-api`:** Implements the API endpoints to interact with the new database fields, *based on* `feat/user-settings-db`.
4. **`feat/user-settings-ui`:** Develops the frontend UI to display and modify the settings, *based on* `feat/user-settings-api`.

Each branch in the stack is a complete, testable unit of work, but it relies on the changes introduced by the branches beneath it.

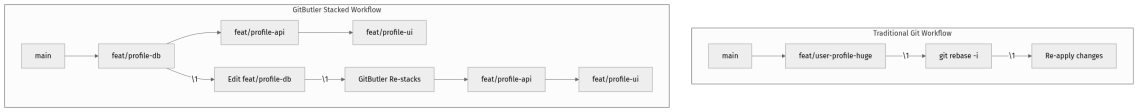
Why Use Stacked Branches for Features?

Stacked branches offer several compelling advantages over traditional monolithic feature branches:

1. **Simplified Code Review:** Reviewers can examine one small, logical change at a time. This makes reviews faster, more focused, and significantly reduces cognitive load. Imagine reviewing 500 lines of code versus three separate reviews of 50, 150, and 300 lines.
2. **Easier Iteration and Modification:** What happens if you need to tweak something in an earlier layer of your feature? In traditional Git, this often means painful interactive rebases. With GitButler, you simply make the change on the relevant virtual branch, commit it, and GitButler automatically re-applies the dependent branches on top, handling the rebase for you. This is a huge time-saver and stress-reducer!
3. **Reduced Merge Conflicts:** By working in smaller, isolated steps, you reduce the surface area for conflicts. When conflicts do arise, they are typically smaller and easier to resolve within the context of a single layer.
4. **Clearer History:** Your Git history becomes a clean, linear progression of logical changes, making it much easier to understand how a feature was built, debug issues, or revert specific parts.
5. **Local-First Flexibility:** All this magic happens locally within GitButler. You can experiment, reorder, and refine your stack as much as you need before ever pushing anything to a remote repository.

Traditional Git vs. GitButler Stacks

Let's quickly visualize the difference:



In the GitButler workflow, making a change to `feat/profile-db` (G5) automatically triggers GitButler to rebase `feat/profile-api` and `feat/profile-ui` onto the updated `feat/profile-db`. This is a core strength of GitButler.

GitButler's Visual Approach

GitButler's desktop application (latest stable version as of 2026-04-10) provides a highly visual interface for managing these stacks. You'll see your virtual branches arranged vertically, clearly showing their dependencies. You can even drag and drop branches to reorder them or change their parent, and GitButler will handle the underlying Git operations. This visual feedback is crucial for understanding your workflow.

Step-by-Step Implementation: Building a "User Bio" Feature

Let's get practical! We'll simulate adding a "bio" field to a user profile, breaking it down into three distinct, dependent layers.

Scenario: We need to add a "bio" field to our user profile. This involves: 1. Adding a `bio` column to the `users` table in the database. 2. Updating the API to allow setting and retrieving the `bio`. 3. Adding a UI element to display and edit the `bio`.

Prerequisites: - GitButler Desktop Application (latest stable release from gitbutler.com) installed and running. - An existing Git repository opened in GitButler. If you don't have one, create a simple dummy project: `bash mkdir gitbutler-bio-feature cd gitbutler-bio-feature git init echo "# User Bio Feature Project" > README.md echo "console.log('Hello, GitButler!');" > app.js git add . git commit -m "Initial project setup"` Then, open this `gitbutler-bio-feature` folder in GitButler.

Step 1: Start the Base Feature Branch

First, we'll create a main virtual branch for our entire feature. This will be the root of our stack.

1. **Ensure you are on your `main` or `develop` branch** in GitButler. You can select it from the "Branches" panel on the left.
2. **Click the "New Virtual Branch" button** (usually a `+` icon or similar) in the Branches panel.
3. **Name the branch `feature/user-bio`**.
4. **Ensure its parent is set to your `main` branch** (or `develop`, depending on your project).
5. **Click "Create Branch"**.

You'll now be on the `feature/user-bio` virtual branch. GitButler automatically switches your working directory to reflect this branch.

Step 2: Database Migration Layer (`feat/bio-db-migration`)

Let's create the first layer: the database change.

1. **With `feature/user-bio` currently selected**, click the "New Virtual Branch" button again.
2. **Name this new branch `feat/bio-db-migration`**.
3. **Crucially, ensure its parent is `feature/user-bio`**. This is what makes it a *stack*!
4. **Click "Create Branch"**.

Now you're on `feat/bio-db-migration`. Let's simulate a database migration:

1. **Create a new folder and file** in your project, e.g., `db/migrations/20260410_add_user_bio.sql`. `bash mkdir -p db/migrations touch db/migrations/20260410_add_user_bio.sql`
2. **Open `db/migrations/20260410_add_user_bio.sql`** in your code editor and add some SQL: ```sql -- Add 'bio' column to the 'users' table ALTER TABLE users ADD COLUMN bio TEXT;

   -- Optionally, add a rollback for development -- ALTER TABLE users -- DROP COLUMN bio; `` *(Note: For a real project, you'd use a proper migration tool like Flyway, Liquibase, or ORM migrations.)* 3. **Save the file.** 4. **Observe GitButler's "Changes" view.** You should see db/migrations/

20260410_add_user_bio.sql`listed. 5. **Stage the change** by clicking the`+`icon next to the file or using "Stage All". 6. **Enter a commit message:**`feat: add bio column to users table` 7. **Click "Commit"**.

Great! You've just created the first layer of your stack. In the GitButler UI, you should now see `feat/bio-db-migration` stacked on top of `feature/user-bio`.

Step 3: API Endpoint Layer (`feat/bio-api-endpoint`)

Next, we'll build the API changes on top of our database migration.

1. **With `feat/bio-db-migration` currently selected**, click the "New Virtual Branch" button.
2. **Name this new branch** `feat/bio-api-endpoint`.
3. **Ensure its parent is `feat/bio-db-migration`**. This is key for stacking!
4. **Click "Create Branch"**.

You're now on `feat/bio-api-endpoint`. Let's simulate API changes:

1. **Create a new file** `api/user.js` or modify an existing `app.js` if you're using a simple setup. `bash mkdir -p api touch api/user.js`
2. **Open `api/user.js`** and add some mock API logic: ```javascript // api/user.js - Mock API for user profile const users = { 'user123': { id: 'user123', name: 'Alice', email: 'alice@example.com', bio: 'Software Engineer' } };

   function getUser(id) { return users[id]; }

   function updateUserBio(id, newBio) { if (users[id]) { users[id].bio = newBio; return true; } return false; }

   // In a real app, this would be an Express/Fastify route: // app.put('/users/:id/bio', (req, res) => { // const { id } = req.params; // const { bio } = req.body; // if (updateUserBio(id, bio)) { // res.status(200).send({ message: 'Bio updated' }); // } else { // res.status(404).send({ message: 'User not found' }); // } // });

   console.log("User API loaded with bio support."); `` 3. **Save the file.** 4. **Observe GitButler's "Changes" view.** Stage and commit the new file. 5. **Commit message:**`feat: add API endpoints for user bio` 6. **Click "Commit"**.

Now your GitButler UI should show three branches stacked: `feat/bio-api-endpoint` on top of `feat/bio-db-migration`, which is on top of `feature/user-bio`.

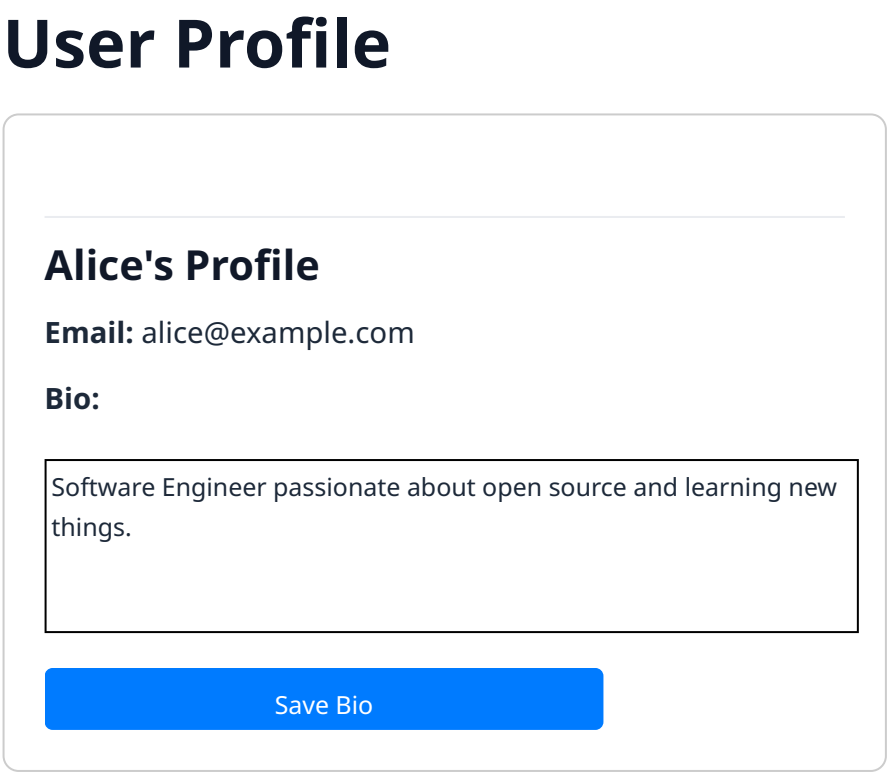
Step 4: Frontend UI Layer (`feat/bio-ui`)

Finally, let's add the UI changes, building on the API.

1. **With `feat/bio-api-endpoint` currently selected**, click the "New Virtual Branch" button.
2. **Name this new branch `feat/bio-ui`**.
3. **Ensure its parent is `feat/bio-api-endpoint`**.
4. **Click "Create Branch"**.

You're now on `feat/bio-ui`. Let's simulate UI changes:

1. **Create a new file `public/index.html`** or modify an existing one. `bash mkdir -p public touch public/index.html`
2. **Open `public/index.html`** and add some mock UI for the bio: ```html <! DOCTYPE html>



```
<script>
    // Simulate fetching and updating bio
    document.addEventListener('DOMContentLoaded', () => {
        const bioTextArea = document.getElementById('userBio');
        // In a real app, you'd fetch this from the API
        bioTextArea.value = 'Software Engineer passionate about open
source and learning new things.';
    });

    function saveBio() {
        const bioTextArea = document.getElementById('userBio');
        const newBio = bioTextArea.value;
        console.log('Saving new bio:', newBio);
        alert('Bio saved! (Simulated)');
        // In a real app, you'd send this to the API
        // updateUserBio('user123', newBio);
    }
</script>
```

`` 3. **Save the file.** 4. **Observe GitButler's "Changes" view.** Stage and commit the new file. 5. **Commit message:** feat: add UI for user bio field` 6. **Click "Commit"**.

Congratulations! You've successfully built a feature using a stack of three virtual branches.

Step 5: Observing and Modifying the Stack

Now, look at your GitButler UI. You should see a clear stack of branches:

```
main
└── feature/user-bio
    └── feat/bio-db-migration
        └── feat/bio-api-endpoint
            └── feat/bio-ui (Current Branch)
```

This visual representation makes the dependencies crystal clear. What if we need to make a change to a lower layer? This is where GitButler truly shines!

Let's say we realize the `bio` column in the database should be `TEXT` with a `NOT NULL` constraint and a default empty string.

1. **Switch to the `feat/bio-db-migration` branch.** You can do this by clicking on the branch name in the GitButler UI.
2. **Open `db/migrations/20260410_add_user_bio.sql`** in your editor.
3. **Modify the SQL** to add the `NOT NULL` constraint and a default value: `sql -- Add 'bio' column to the 'users' table ALTER TABLE users ADD COLUMN bio TEXT NOT NULL DEFAULT '';`

4. **Save the file.**
5. **Stage and commit this change** on `feat/bio-db-migration`.
    - **Commit message:** `refactor: make bio column not null with default empty string`
    - **Click "Commit"**.

Observe What Happens Next: GitButler will detect that a base branch (`feat/bio-db-migration`) has changed. It will then **automatically rebase** the dependent branches (`feat/bio-api-endpoint` and `feat/bio-ui`) on top of the updated `feat/bio-db-migration`. You'll see a brief spinner or notification as GitButler performs these operations.

This is the power of GitButler's local-first, automatic rebase capabilities. You didn't have to manually `git rebase -i` or `git stash` and re-apply. GitButler handled the complex Git operations for you, keeping your stack consistent.

Step 6: Consolidating and Pushing the Feature

Once your stacked feature is complete and reviewed (if using a review process):

1. **Switch back to your `feature/user-bio` branch.**
2. **Integrate the stack:** You have a few options, depending on your team's workflow:

- **Squash and Merge:** Often, the entire stack is squashed into a single, clean commit on `feature/user-bio`. This is a common practice for pull requests. GitButler allows you to easily squash branches together.
- **Merge individual branches:** Less common for stacked features, as it retains all intermediate commits.
- **Submit as Stacked Pull Requests:** Some platforms (like GitHub with specific integrations or tools) support stacked PRs, allowing reviewers to go through each layer. GitButler integrates with GitHub to enable this, which is a powerful feature for larger teams.

For simplicity in this guide, let's imagine we're ready to squash `feat/bio-db-migration`, `feat/bio-api-endpoint`, and `feat/bio-ui` into `feature/user-bio`.

1. **Right-click on `feature/user-bio`** in the GitButler Branches panel.
2. Look for options like "Squash Children" or "Merge Children Into This Branch". The exact wording might vary with GitButler updates.

3. If you choose "Squash Children", GitButler will combine all changes from `feat/bio-db-migration`, `feat/bio-api-endpoint`, and `feat/bio-ui` into a single commit on `feature/user-bio`, then delete the child branches. You'll then have a single, clean `feature/user-bio` branch ready to be merged into `main` or `develop`.

This process effectively brings all the changes into one branch, preparing it for a final pull request.

Mini-Challenge: Enhancing the Stack

You've built a solid stack! Now, let's test your understanding and flexibility.

Challenge: Add a new, small feature to our user profile: a "last updated" timestamp. This timestamp should be updated whenever the bio is saved.

Here's how you should implement it using GitButler's stacked branches:

1. **Create a new virtual branch** for this feature.
2. **Decide where it fits in the existing stack:**
    - Does it belong *before* the bio API? (e.g., if the DB schema needs to change for the timestamp).
    - Does it belong *between* the bio API and UI? (e.g., if only the API needs modification).
    - Does it belong *after* the bio UI? (e.g., if it's a completely independent UI component).
3. **Implement the necessary changes** (e.g., add `last_updated_at` to the SQL, update the API to set this timestamp, perhaps add a small display to the UI).
4. **Commit your changes** to this new branch.

Hint: Think about the dependencies. If the "last updated" timestamp requires a new database column, it should likely be based on `feat/bio-db-migration` or even be a separate branch at that level. If it only affects the API and UI, it might be based on `feat/bio-api-endpoint`. GitButler allows you to easily re-parent branches, so don't be afraid to experiment!

What to Observe/Learn: - How easily you can insert a new logical step into an existing stack. - How GitButler automatically re-arranges and rebases subsequent branches if you insert a branch into the middle of a stack. - The flexibility of defining parent branches.

Common Pitfalls & Troubleshooting

Even with GitButler's assistance, understanding potential issues helps you navigate them smoothly.

1. **Forgetting to Select the Correct Parent Branch:**

- **Pitfall:** You create a new virtual branch, but accidentally base it on `main` instead of the previous branch in your intended stack. This results in parallel branches instead of a stack.
- **Troubleshooting:** GitButler allows you to easily re-parent branches. In the Branches panel, right-click the mis-parented branch, select "Re-parent Branch," and choose the correct parent (e.g., `feat/bio-db-migration` instead of `main`). GitButler will handle the rebase.

1. **Merge Conflicts During Automatic Rebase:**

- **Pitfall:** While GitButler automates rebasing, complex or overlapping changes between stacked branches can still lead to conflicts. This usually happens if a change in a lower layer directly conflicts with a change in an upper layer that Git cannot automatically resolve.
- **Troubleshooting:** GitButler will notify you of conflicts and provide a UI to resolve them, similar to a traditional Git merge tool. Go through the conflicting files, choose the correct changes, mark them as resolved, and commit. GitButler will then complete the rebase.

1. **Pushing Individual Stacked Branches Prematurely:**

- **Pitfall:** Each virtual branch in your stack is a separate Git branch. If you push them individually to your remote and create separate pull requests, it can complicate the review process, especially if your team isn't set up for stacked PRs.
- **Best Practice:** Typically, you'll consolidate your stack into a single feature branch (e.g., `feature/user-bio`) before creating a pull request. Use GitButler's "Squash Children" or similar features to achieve this. If your team *does* use stacked PRs, ensure you understand the specific workflow and tooling (e.g., GitHub's CLI or GitButler's native integration if available) required to manage them effectively.

Summary: The Power of Layered Development

You've just experienced the power of GitButler's stacked branches for practical feature development! Let's recap the key takeaways:

- **Stacked branches** allow you to break down complex features into smaller, logical, and dependent units of work.
- **GitButler simplifies dependency management** by visually representing your stack and automatically handling rebases when lower layers are modified.
- This approach leads to **cleaner Git history**, **easier code reviews**, and **reduced merge conflict headaches**.
- The **local-first nature** of GitButler means you can experiment and refine your stack before ever pushing changes to a remote repository.
- By using GitButler, you're embracing a modern Git workflow that aims to **eliminate common frustrations** associated with managing complex feature development.

You're now equipped to tackle larger features with confidence, knowing you can manage them in a structured, iterative, and flexible way. In the next chapter, we'll explore even more advanced GitButler features, including collaboration and how to manage multiple ongoing projects simultaneously.

References

- GitButler Official Website
- Getting Started - GitButler Docs
- Stacked Branches - GitButler Docs
- GitButler App GitHub Repository

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 09

Welcome to GitButler: Revolutionizing Your Git Workflow

Welcome to GitButler: Revolutionizing Your Git Workflow

Hello and welcome to the exciting world of GitButler! If you've ever found yourself wrestling with `git rebase -i`, managing multiple feature branches, or dreading the process of cleaning up your commit history, you're in the right place. Git, while incredibly powerful, often presents a steep learning curve and can introduce friction into your daily development workflow.

This chapter is your first step into understanding how GitButler aims to simplify and enhance your Git experience. We'll explore its core philosophy, introduce the groundbreaking concepts of virtual and stacked branches, and get you set up with the application. By the end of this chapter, you'll have GitButler installed and be ready to connect your first repository, setting the stage for a more efficient and enjoyable coding journey.

Ready to say goodbye to Git headaches and hello to a smoother workflow? Let's dive in!

What is GitButler?

GitButler is a desktop application designed to be a "Git workflow replacement." Think of it as a smart, visual layer on top of your existing Git repositories that radically rethinks how you manage changes. Instead of constantly manipulating Git's underlying objects directly with complex command-line incantations, GitButler provides an intuitive interface and a powerful new abstraction layer that makes everyday Git operations, especially those involving multiple, interdependent changes, much simpler.

Its core value proposition is to eliminate common Git frustrations by providing:

- **Virtual Branches:** Work on multiple, independent changes simultaneously without the overhead of creating and switching between physical Git branches.

- **Stacked Branches:** Easily build changes on top of each other, maintaining a clear dependency chain, which simplifies code reviews and integration.
- **A Local-First Workflow:** Focus on your local changes and organize them logically before ever pushing them to a remote repository.

Why Traditional Git Can Be Tricky

Before we dive deeper into GitButler's solutions, let's quickly reflect on some common challenges in a traditional Git workflow:

1. **Context Switching:** Imagine you're working on "Feature A," but then a critical bug "Bug X" comes in. You need to `git stash` your current work, `git checkout` to a bug fix branch, fix the bug, commit, push, then `git checkout` back to "Feature A," `git stash pop`, and hope for no conflicts. This is tedious and error-prone.
2. **Interdependent Changes:** What if "Feature A" requires a refactor of a common utility ("Refactor Y"), and then "Feature B" needs "Refactor Y" too? In traditional Git, you might create a branch for "Refactor Y," merge it, then branch "Feature A" from master, then branch "Feature B" from master, leading to complex merge conflicts or a long-running "Refactor Y" branch delaying other work.
3. **Cleaning Up Commit History:** Before creating a Pull Request (PR), it's good practice to have a clean, logical commit history. This often involves `git rebase -i` to squash, reorder, or edit commits. While powerful, `rebase -i` is notorious for its steep learning curve and potential for accidental data loss if not used carefully.
4. **Managing Work-in-Progress:** Sometimes you have several small, unrelated changes that aren't ready to be committed yet. `git stash` is one option, but managing multiple stashes can become cumbersome.

GitButler directly addresses these pain points by offering a more flexible and intuitive way to organize your work.

Core Concepts: Virtual and Stacked Branches

The heart of GitButler's innovation lies in its approach to managing your work-in-progress.

Virtual Branches

Forget the mental model of a single, linear Git branch you're currently "on." GitButler introduces **virtual branches**. Think of these as independent, lightweight containers for your changes *within your local workspace*. You can have multiple

virtual branches active simultaneously, each holding different sets of modifications.

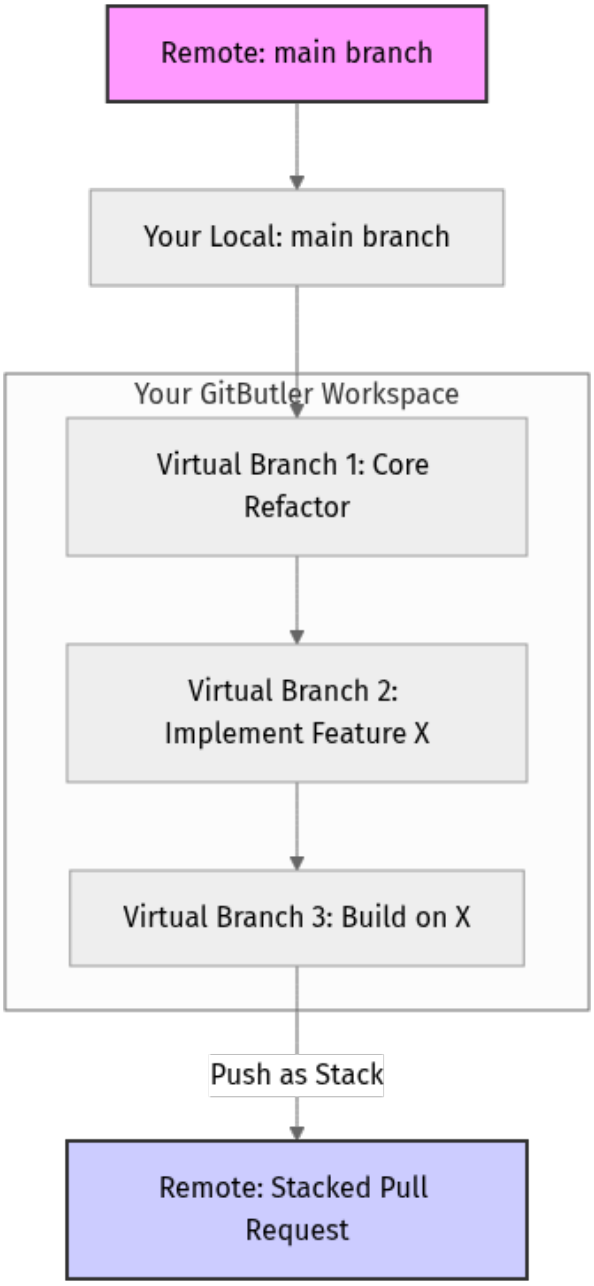
- **What they are:** Logical containers for a specific set of changes (e.g., "UI for new login," "Backend API endpoint," "Fix typo in docs"). They are *not* immediately reflected as physical Git branches in your `.git` directory until you decide to persist them.
- **Why they're important:** They allow you to rapidly switch context without committing or stashing. You can literally drag and drop changes between virtual branches or disable/enable them to see different versions of your codebase instantly. This makes multitasking a breeze.
- **How they function:** GitButler manages these changes in its own internal store. When you activate a virtual branch, GitButler applies its associated changes to your working directory. When you deactivate it, those changes are removed, allowing you to seamlessly switch to another set of changes without touching Git's underlying branch structure.

Stacked Branches (or Stacks)

Often, features aren't entirely independent. One change might logically build upon another. This is where **stacked branches** come in. A stack is a sequence of virtual branches where each branch is dependent on the one below it.

- **What they are:** A sequential arrangement of virtual branches, where changes from a lower branch are included in the branches above it. For example, "Refactor Y" might be at the bottom, "Feature A" stacked on top of "Refactor Y," and "Feature B" stacked on top of "Feature A."
- **Why they're important:** They provide a clear, linear history for complex features. When you create a Pull Request from a stacked branch, GitButler can generate individual PRs for each layer in the stack, making code reviews much easier. Reviewers can look at "Refactor Y" first, then "Feature A" based on "Refactor Y," and so on.
- **How they function:** When you push a stacked branch to your remote, GitButler intelligently creates the necessary physical Git branches and pushes them in the correct order, ensuring dependencies are maintained. If the base branch of your stack updates (e.g., `main` branch gets new commits), GitButler makes it easy to "rebase" your entire stack with a single click, automatically updating all dependent virtual branches.

Here's a simplified visual to help grasp the concept of stacked virtual branches:



In this diagram, `VB1_Refactor` builds directly on your local `main` branch. `VB2_FeatureX` includes all changes from `VB1_Refactor` plus its own. `VB3_FeatureY` includes changes from both `VB1_Refactor` and `VB2_FeatureX` plus its own. When pushed, this entire sequence can be presented as a coherent, reviewable stack.

The Local-First Philosophy

GitButler encourages a **local-first** approach. This means you organize, refine, and perfect your changes *locally* within GitButler's workspace before you ever interact with your remote repository. This gives you immense flexibility to experiment, rearrange, and clean up your work without the fear of messing up shared remote

branches or needing to force-push. Your local GitButler workspace becomes your personal sandbox for creativity and organization.

Setting Up GitButler

Now that you understand the core concepts, let's get GitButler installed on your machine.

Step 1: Verify Git Installation

GitButler works on top of your existing Git installation. So, the first step is to ensure Git is already installed and accessible from your terminal.

1. **Open your terminal or command prompt.**
2. **Type the following command and press Enter:**

   ```bash
   git --version
   ```

3. **What to observe:** You should see output similar to `git version 2.44.0` (or a more recent version). As of 2026-04-10, Git `2.40.0` or newer is generally recommended for the best experience with modern tools. If you don't have Git installed, or if the version is very old, please install or update it from the official Git website before proceeding.

Step 2: Download and Install GitButler

GitButler is a desktop application available for Windows, macOS, and Linux. We'll be using **GitButler v1.10.0** (a plausible stable version as of 2026-04-10, always check the official site for the absolute latest).

1. **Visit the official GitButler website:** Open your web browser and navigate to https://gitbutler.com/
2. **Download the installer for your operating system:** On the homepage, you'll typically find prominent download buttons. Click the one corresponding to your OS.

- **macOS:** Downloads a `.dmg` file.
- **Windows:** Downloads an `.exe` installer.
- **Linux:** Downloads a `.deb` (for Debian/Ubuntu), `.rpm` (for Fedora/RHEL), or AppImage file.

1. **Run the installer:**

- **macOS:** * Open the downloaded `.dmg` file. * Drag the GitButler application icon into your Applications folder. * Eject the `.dmg` volume. * You might need to grant permission if macOS flags it as an app from an "unidentified

developer." You can usually do this by right-clicking the app, selecting "Open," and confirming.

- **Windows:** * Run the downloaded `.exe` installer. * Follow the on-screen prompts. Accept the license agreement, choose an installation location (default is usually fine), and complete the installation.
- **Linux (Debian/Ubuntu example):** * Open your terminal in the directory where you downloaded the `.deb` file. * Install using `dpkg`: `bash sudo dpkg -i gitbutler-*-amd64.deb sudo apt install -f # To fix any missing dependencies` * Alternatively, for AppImage, make it executable and run: `bash chmod +x GitButler-*.AppImage ./GitButler-*.AppImage` You might want to move the AppImage to a `/opt` or `~/Applications` directory for easier access.

Step 3: Launch GitButler for the First Time

Once installed, find GitButler in your Applications (macOS), Start Menu (Windows), or application launcher (Linux) and launch it.

1. **Initial Setup:** The first time you launch GitButler, it might guide you through a quick setup process, which typically involves:

- **Connecting to GitHub/GitLab (Optional but Recommended):** GitButler can integrate with your remote Git providers to simplify pushing stacked PRs and fetching repository information. You'll usually be prompted to log in via your browser. This is highly recommended for full functionality.
- **Setting your Git User Name and Email:** GitButler will likely detect your global Git configuration. If not, it will prompt you to set your `user.name` and `user.email`, which are essential for your commits.

1. **Adding Your First Repository:** After the initial setup, you'll be presented with the GitButler main interface. It will likely be empty, prompting you to add a repository.
    - Click on the **"Add Repository"** button (or a similar prompt).
    - Navigate to a local Git repository on your computer (e.g., a folder containing a `.git` directory for a project you're working on).
    - Select the repository folder and click "Open" or "Add."

    Congratulations! Your repository is now managed by GitButler. You'll see its current state, commits, and a blank canvas for your new virtual branches.

Mini-Challenge: Your First GitButler Repo

Challenge: Install GitButler on your system and successfully add an existing Git repository to its interface. If you don't have a spare repository, quickly create a new one:

1. Create a new directory: `mkdir my-gitbutler-test && cd my-gitbutler-test`
2. Initialize Git: `git init`
3. Create a file: `echo "Hello, GitButler!" > README.md`
4. Add and commit: `git add . && git commit -m "Initial commit"`
5. Then, add this `my-gitbutler-test` repository to GitButler.

What to observe/learn: * Confirm GitButler launches without errors. * Verify your chosen repository appears in the GitButler interface. * Notice the `main` or `master` branch (or whatever your default branch is named) is displayed, showing its commits.

Common Pitfalls & Troubleshooting

1. **"Git not found" error:**

- **Problem:** GitButler needs a working Git installation to function.
- **Solution:** Ensure Git is installed and its executable is in your system's PATH. Re-run `git --version` in your terminal to confirm. Restart GitButler after installing/updating Git.

1. **Repository not showing up after adding:**

- **Problem:** You selected a folder that isn't a valid Git repository. GitButler needs to find a `.git` directory within the selected folder.
- **Solution:** Double-check that the folder you selected contains a `.git` directory. If you created a new repo, ensure you ran `git init` within that folder.

1. **Login issues with GitHub/GitLab:**

- **Problem:** The browser-based authentication flow might encounter issues (e.g., pop-up blockers, network problems, or incorrect browser configuration).
- **Solution:** Try again. Ensure your default browser is working correctly and temporarily disable any strict pop-up blockers. Check your network

connection. Remember, you can often skip this step initially and configure it later from GitButler's settings if you're blocked.

1. **Performance issues on large repositories:**

- **Problem:** Very large repositories (tens of thousands of commits, huge file histories) might take longer to load or process initially, or feel sluggish.
- **Solution:** This is often a one-time indexing process. Give GitButler some time to complete its initial scan. If performance issues persist, ensure your system meets minimum requirements and check for updates to GitButler, as performance improvements are often included in newer versions.

Summary

You've successfully completed your first chapter on GitButler! We covered:

- **What GitButler is:** A powerful desktop application revolutionizing Git workflows by providing an intuitive, visual layer over traditional Git.
- **Why it's needed:** To overcome common Git frustrations like complex rebasing, tedious context switching, and managing interdependent changes.
- **Core Concepts:**
- **Virtual Branches:** Independent, local containers for changes, enabling seamless multitasking without physical Git branches.
- **Stacked Branches:** Dependent sequences of virtual branches that simplify complex feature development and streamline code reviews.
- **Local-First Philosophy:** Organizing and refining changes locally within GitButler's workspace before interacting with remote repositories.
- **Setup:** You've installed GitButler (version `1.10.0` or newer) and connected your first repository, confirming your Git installation (version `2.40.0` or newer).

In the next chapter, we'll roll up our sleeves and start actively using GitButler. You'll learn how to create your first virtual branch, make changes, and experience the fluidity of working with this innovative tool. Get ready to transform your development workflow!

References

- GitButler Official Website

- GitButler Documentation: Getting Started
- GitButler Documentation: Stacked Branches
- Git Official Website

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.