AI VOID FIELD GUIDE

Mastering Jujutsu (jj VCS): A Zero-to-Advanced Guide

Embark on a comprehensive journey to master Jujutsu (jj VCS) from the ground up, with practical steps and engaging challenges for modern software engineering workflows.

AI VOID

2026-05-19 03:31 IST

Contents

CHAPTER 01

Welcome to Jujutsu: A New VCS Paradigm

Are you a developer who's ever felt bogged down by the complexities of Git? Do you wish for a version control system that makes iterative development, refactoring, and collaboration feel more natural and less error-prone? You're in the right place!

Welcome to Jujutsu, often abbreviated as `jj`, a modern, Git-compatible version control system designed to streamline your development workflow. In this guide, we'll embark on a journey to master `jj`, starting with its fundamental concepts and practical applications. This first chapter introduces you to the core philosophy of Jujutsu, guides you through its installation, and helps you take your very first steps.

By the end of this chapter, you'll understand `jj`'s unique approach to version control, have it installed on your system, and be ready to create and manage your first commits. We assume you're already familiar with basic version control concepts like commits, branches, and merging, ideally from experience with Git or Mercurial.

Understanding Jujutsu's Core Philosophy

Jujutsu (`jj`) is a new generation of version control system that aims to combine the best aspects of Git and Mercurial while introducing powerful new concepts. It's built to be Git-compatible, meaning you can use `jj` on existing Git repositories and interact seamlessly with Git remotes like GitHub or GitLab. However, `jj` isn't just a Git wrapper; it fundamentally rethinks how you interact with your code history.

Why Jujutsu? Solving Common VCS Pain Points

`jj` was created to address several common frustrations developers face with traditional VCS, especially Git. Understanding these pain points helps clarify why `jj`'s design choices are so impactful.

- **Complex Staging Area:** Git's staging area (index) can be powerful but often adds an extra mental burden, especially for beginners or when making quick changes. It requires an explicit `git add` step before committing.

- **Immutable History Mindset:** Git's strong emphasis on immutable history, while good for integrity, can make refactoring and cleaning up your local commits cumbersome. Operations like `rebase -i` are powerful but require careful manual intervention and understanding of detached HEAD states.
- **Branch Management Overload:** Traditional branches in Git can become unwieldy, leading to "branch spaghetti" and confusion about which branch is tracking what. This often hinders linear, iterative development.
- **Lack of Robust Undo:** While Git has `reflog`, it's not a general-purpose, easy-to-use undo mechanism for most operations. Recovery often involves digging through the reflog manually.

`jj` tackles these issues head-on, offering a fresh perspective that prioritizes simplicity, power, and safety.

The Jujutsu Paradigm Shift: Core Concepts

To truly appreciate `jj`, it's crucial to understand its core differentiating concepts. These aren't just new features; they represent a fundamental shift in how you think about version control, moving from a rigid, "snapshot-based" model to a more fluid, "change-based" one.

The Working Copy as a Commit

In Git, your working directory, the staging area, and your commits are distinct entities. You modify files, `git add` them to the staging area, and then `git commit` to create a new snapshot. This multi-step process can sometimes feel disconnected from the actual work.

`jj` simplifies this by treating your entire **working copy as a commit itself**. What does this mean in practice?

- There's no separate staging area. Any change in your working directory is implicitly part of the "working copy commit."
- When you run `jj commit`, `jj` creates a new commit that contains all changes from your current working directory. Then, it automatically moves the "working copy commit" to be a child of this new, just-created commit.

This model often feels more intuitive, as your working directory is always in a committable state, embodying the current "in-progress" commit. You don't need to explicitly stage changes; just save your files, and `jj` sees them as part of your active work.

Mutable History by Design

One of `jj`'s most powerful features is its embrace of **mutable history**. Unlike Git, where changing past commits is seen as an advanced operation (and potentially dangerous if not handled carefully, especially when interacting with shared history), `jj` makes it a central, safe, and easy part of your local workflow.

Why is this important? It empowers you to refine your work and maintain a clean, logical history before sharing it with others. This means you can effortlessly:

- **Edit previous commits:** Fix a typo, add a missing file, or refine a commit message.
- **Reorder commits:** Arrange your changes in a more logical sequence.
- **Split a single commit into multiple:** Break down a large commit into smaller, more reviewable chunks.
- **Combine multiple commits into one:** Squash several related commits into a single, cohesive change.

`jj` encourages you to iterate and refine your local history before sharing it, leading to cleaner, more logical commit graphs. This flexibility is a game-changer for iterative development and code review preparation.

The Operation Log: Your Safety Net

With great power comes great responsibility, but `jj` provides a crucial safety net: the **operation log**. Every single action you perform with `jj` – every commit, rebase, undo, or update – is recorded in this log. This is a fundamental difference from Git's `reflog`, which primarily tracks where your branch pointers have been. The `jj` operation log tracks every command executed.

This log gives you unparalleled confidence to experiment freely with `jj`'s mutable history features, knowing you can always revert to a previous state of your repository.

- **Undo any operation:** Made a mistake? Just `jj undo`. This command effectively reverses the last `jj` operation, restoring your repository to its state before that operation.
- **Redo an undone operation:** Changed your mind after an undo? `jj redo` will reapply the undone operation.
- **Explore your command history:** The `jj op log` command allows you to see what you did and when, providing a transparent audit trail of your actions.

📌 Key Idea: The operation log is Jujutsu's powerful safety net, allowing you to undo and redo any operation, making mutable history fearless. This significantly reduces the cognitive load of complex refactoring.

Installing Jujutsu (as of 2026-05-19)

Let's get `jj` up and running on your system! Jujutsu is actively developed, and its latest stable version often includes significant improvements. As of **May 19, 2026**, we'll aim for a recent stable release. For consistency and stability, let's target **Jujutsu version 0.20.0** or later, which reflects current stable releases. Always check the official GitHub releases page for the absolute latest version and release notes.

You can install `jj` in several ways depending on your operating system.

macOS Installation

The recommended way to install `jj` on macOS is using Homebrew, a popular package manager.

```
brew install jujutsu
```

Linux Installation

For Linux, you can often find pre-built binaries or install via `cargo`. Using `cargo` is generally recommended for the latest versions or if pre-built packages aren't readily available for your distribution.

Using `cargo` (Recommended for latest versions)

First, ensure you have Rust and Cargo installed. If not, follow the official installation instructions at rust-lang.org.

Then, install `jj` using `cargo`:

```
cargo install jj
```

Pre-built Binaries (Alternative)

You can also download pre-built binaries from the official Jujutsu GitHub releases page. Look for the `.tar.gz` archive for your architecture (e.g., `jj-v0.20.0-x86_64-unknown-linux-gnu.tar.gz`), extract it, and place the `jj` executable in a directory that's included in your system's `PATH`.

Windows Installation

For Windows, pre-built binaries are typically the easiest route.

Pre-built Binaries (Recommended)

Download the appropriate `.zip` file (e.g., `jj-v0.20.0-x86_64-pc-windows-msvc.zip`) from the official Jujutsu GitHub releases page. Extract the `jj.exe` executable and place it in a directory that's included in your system's `PATH` environment variable.

Verify Your `jj` Installation

After installation, open a new terminal or command prompt and run the following command to check if `jj` is correctly installed and accessible:

```
jj --version
```

You should see output similar to this (the exact version number might differ if you installed a newer release):

```
jj 0.20.0
```

If you see the version number, congratulations! `jj` is ready to go. If not, double-check your installation steps, especially ensuring the `jj` executable is in your system's `PATH`.

Getting Hands-On with Jujutsu: Your First Repository and Commit

Now that `jj` is installed, let's dive in and experience its unique workflow by creating a new project and making our first commits. This section will walk you through the essential commands for daily use.

Step 1: Initialize Your First Jujutsu Repository

We'll start by creating a new directory for our project and initializing a `jj` repository within it.

1. **Create a Project Directory:** Open your terminal or command prompt and create a new directory for your project.

```
mkdir my_jj_project
cd my_jj_project
```

1. **Initialize the Jujutsu Repository:** Inside your new project directory, run the `jj init` command.

```
jj init
```

```
You'll see output confirming the initialization:
```

```
Initialized empty Jujutsu repository in my_jj_project/.jj
```

```
This command creates a `.jj` directory inside your project, similar to Git's
`.git` directory, where `jj` stores its internal data.
```

1. **Check the Status:** Let's see the current state of our newly initialized repository using `jj status`.

```
jj status
```

```
You should see something like this:
```

```
Current working copy: 000000000000
There is no Diff in the working copy
```

```
`000000000000` is `jj`'s special identifier for the "null" or "empty" commit.
It represents the state before any actual changes have been committed. At this
point, your working copy is empty and has no changes.
```

Step 2: Making Your First Change and Commit

It's time to create some content and make our first `jj` commit! Remember, with `jj`, there's no staging area. Your working directory is your current working copy commit.

1. **Create a File:** Let's create a simple `README.md` file using `echo`.

```
echo "# My First Jujutsu Project" > README.md
echo "This is a test project to learn Jujutsu." >> README.md
```

1. **Check Status Again:** Now that we've made changes, let's see what `jj status` reports.

```
jj status
```

```
Output:
```

```
Current working copy: 000000000000
Working copy has changes:
  A README.md
```

```
Notice how `jj` immediately recognizes the new file and indicates that the
"working copy has changes." There's no `jj add` command because your working
copy *is* a commit; any changes are automatically part of that implicit
working copy commit.
```

1. **View the Diff:** You can see the changes in your working copy using `jj diff`. This is analogous to `git diff`.

```
jj diff
```

```
Output:
```

```
diff --git a/README.md b/README.md
new file mode 100644
--- /dev/null
+++ b/README.md
@@ -0,0 +1,2 @@
+# My First Jujutsu Project
+This is a test project to learn Jujutsu.
```

```
This `diff` output is very similar to what you'd see in Git, showing the new
lines added.
```

1. **Commit Your Changes:** Now, let's create our first actual commit. In `jj`, the `commit` command takes all changes in your working directory and creates a new commit that is a child of your current working copy's parent.

```
    jj commit -m "Initial project setup"
```

```
Output:
```

```
    Created 6d5e1f706e22 (empty) (no description set)
    Replaced 000000000000 with 6d5e1f706e22 (empty) (no description set)
    Working copy now at 6d5e1f706e22 (empty) (no description set)
```

```
What just happened?
- `jj` created a new commit with a unique ID (e.g., `6d5e1f706e22`). This is
your actual commit, containing the `README.md` file.
- The "working copy commit" then *moved* to become this new commit. This is
the "working copy as a commit" in action.
- The `(empty)` and `(no description set)` parts are internal details `jj`
shows briefly because it creates the commit *then* applies the message and
content. Don't worry about them.
```

1. **Check Status After Commit:** Let's confirm the state of our repository after the commit.

```
    jj status
```

```
Output:
```

```
    Current working copy: 6d5e1f706e22 Initial project setup
    There is no Diff in the working copy
```

```
Now, `jj status` shows your working copy is associated with your new commit
`6d5e1f706e22`, and there are no further changes in your working directory.
Perfect!
```

Step 3: Exploring Your History

Let's see the history we've created and how `jj` presents it.

1. **View the Commit Log:** The `jj log` command displays your repository's history, similar to `git log` but with `jj`'s unique perspective.

```
    jj log
```

```
Output (commit IDs will vary):
```

```
    @ 6d5e1f706e22 Initial project setup
    o 000000000000 (empty) (no description set)
```

```
Here's what you're seeing:
- `@ 6d5e1f706e22 Initial project setup`: This is your `Initial project setup`
commit. The `@` symbol is crucial — it indicates that this is the commit your
working copy is currently on.
- `o 000000000000 (empty) (no description set)`: This is the "null" commit,
the conceptual parent of your very first commit.

Notice the linear, clean history. `jj` aims for this by default, simplifying
the visual representation of your changes.
```

1. **Make Another Change and Commit:** Let's add another line to our `README.md` and commit it, observing how the history evolves.

```
    echo "This is the second line of the README." >> README.md
    jj commit -m "Add second line to README"
```

```
Now, run `jj log` again:
```

```
    jj log
```

```
Output (again, IDs will vary):
```

```
    @ 7a1b2c3d4e5f Add second line to README
    o 6d5e1f706e22 Initial project setup
```

```
    o 000000000000 (empty) (no description set)
```

```
You can clearly see your two commits stacked on top of each other, with the
working copy (`@`) pointing to the latest one. This "stacking" of commits is a
core concept in `jj` that we'll explore much more deeply in future chapters.
```

1. **Show a Specific Commit:** You can view the contents of any specific commit using `jj show <commit_id>`. Let's look at the `Initial project setup` commit. You can use a short prefix of the commit ID.

```
    jj show 6d5e1f706e22
```

```
This command will display the files and their content as they were in that
specific commit, allowing you to inspect past states of your project.
```

Mini-Challenge: Build a Small Feature Iteratively

Let's put your new knowledge to the test and practice the `jj` workflow.

Challenge:

1. Create a new file called `features.txt` with the content "Feature A: Initial idea".
2. Commit this change with a message like "feat: Add initial feature A idea".
3. Modify `features.txt` to add "Feature A: Refined details".
4. Commit this modification with a message like "feat: Refine feature A details".
5. Use `jj log` to observe your commit history, paying attention to the order and the working copy pointer.
6. Use `jj show` on your first feature commit (the one adding "Initial idea") to see its content without the refined details.

Hint: Remember, there's no `jj add`. Just make your changes in the working directory and then `jj commit`.

What to observe/learn:

- How `jj` automatically tracks all changes in your working copy, eliminating the staging area.

- The linear progression and stacking of commits in `jj log` as you build a feature iteratively.
- How `jj show` lets you easily inspect the exact content of any past commit.

Common Pitfalls & Troubleshooting for New `jj` Users

As you transition from Git to `jj`, you might encounter a few initial hiccups due to the fundamental differences in their models. Recognizing these common pitfalls can help you quickly adapt.

1. **Expecting a Staging Area (The `jj add` Reflex):** The most common mistake for Git users is instinctively looking for an `add` command.
    - **Pitfall:** Trying to run `jj add <file>` or similar.
    - **Why it happens:** In Git, `git add` is essential to stage changes.
    - **Solution:** Remember, `jj` doesn't have a separate staging area. All changes in your working directory are implicitly part of the current working copy commit. Just modify files and then `jj commit`. If you only want to commit some changes from your working directory, `jj` has powerful commands like `jj split` or `jj amend -i` (interactive) which we'll cover in detail later.
2. **Misinterpreting `jj commit`'s Behavior:** The `commit` command in `jj` behaves differently than in Git.
    - **Pitfall:** Thinking `jj commit` just moves a branch pointer.
    - **Why it happens:** In Git, `git commit` creates a commit and then moves the current branch pointer.
    - **Solution:** In `jj`, `jj commit` creates a new commit that is a child of your current working copy commit, and then it moves your working copy to point to this new commit. Think of `jj commit` as "capture my current working state as a new commit, and then continue working on top of it." This subtle difference is key to `jj`'s mutable history model.

3. **Ignoring `jj log`'s Visual Cues:** Especially when coming from Git, it's easy to just skim `jj log` and miss its unique indicators.
    - **Pitfall:** Not understanding the `@` symbol or the linear presentation.
    - **Why it happens:** Git's `log` often shows complex branch graphs.
    - **Solution:** Make `jj log` your friend. Use it frequently to understand where your working copy (`@`) is positioned and how your commits are stacked. The linear, cleaner history is a feature, not a lack of information.

🧠 Important: `jj`'s philosophy is "commit early, commit often." Because history is mutable and undo is easy with the operation log, you're encouraged to save your work frequently without worrying about creating "messy" commits. You can always clean them up and refine them later before sharing. This encourages smaller, more focused changes.

Summary and What's Next

In this first chapter, you've taken your initial steps into the world of Jujutsu, a powerful and intuitive version control system. You've installed `jj` and experienced its fundamental workflow.

Here's a quick recap of the key takeaways:

- **Jujutsu's Purpose:** A modern, Git-compatible VCS designed to simplify development workflows and overcome common Git frustrations with its unique model.
- **Core Concepts:**
    - The **working copy as a commit**, eliminating the need for a separate staging area.
    - **Mutable history by design**, making it easy and safe to refine your local commits.
    - The **operation log**, providing a robust undo/redo mechanism for every action, ensuring safety.
- **Installation:** You successfully installed `jj` (targeting version 0.20.0 or later as of 2026-05-19) on your system.
- **First Steps:** You initialized a `jj` repository, made changes, committed them, and explored your repository's history using `jj status`, `jj diff`, `jj commit`, and `jj log`.

- **Common Pitfalls:** We discussed the key mindset shifts required when moving from Git, particularly regarding the absence of a staging area and the unique behavior of `jj commit`.

You're now equipped with the foundational knowledge to interact with `jj` and appreciate its core philosophy. In the next chapter, we'll dive deeper into `jj`'s mutable history features, learning how to modify, reorder, and refine your commits with unparalleled ease. Get ready to experience true flexibility in your version control!

References

- Jujutsu GitHub Repository: https://github.com/jj-vcs/jj
- Jujutsu Releases: https://github.com/jj-vcs/jj/releases
- Jujutsu Tutorial (Official Docs): https://github.com/martinvonz/jj/blob/main/docs/tutorial.md
- The Rust Programming Language (for `cargo` installation): https://www.rust-lang.org/tools/install

---

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

CHAPTER 02

Your First Jujutsu Repository: The Working Copy as a Commit

Welcome back, future Jujutsu wizard! In the previous chapter, we successfully installed `jj` and confirmed it was ready for action. Now, it's time to create our very first Jujutsu repository and dive into a concept that fundamentally differentiates `jj` from traditional Version Control Systems (VCS) like Git: the "working copy as a commit" model.

This chapter is a cornerstone of your `jj` journey. Understanding this core principle is crucial because it's the foundation for `jj`'s powerful mutable history, streamlined workflows, and branchless development style. By the end, you'll not only have a functioning `jj` repository but also a deep intuition for how `jj` perceives your code, preparing you for advanced techniques like stacked changes and effortless rebasing. Let's make that paradigm shift together!

Understanding the Working Copy Commit: Jujutsu's Core Difference

If you're accustomed to Git, you're familiar with a workflow where you explicitly `add` changes to a staging area, then `commit` them. Your working directory often contains uncommitted modifications, and once pushed, Git's history is generally considered immutable. Jujutsu challenges this mental model entirely.

What is the "Working Copy as a Commit" (The `@` Commit)?

In Jujutsu, your working directory—the collection of files you're currently editing—is always treated as a live, mutable commit. This special commit is known as the **working copy commit**, often referred to by its symbolic name `@`.

- **No Staging Area Needed:** Forget `git add`. `jj` automatically tracks all changes in your working directory. Any file you modify, add, or delete immediately becomes part of your working copy commit. This simplifies your daily workflow by removing an entire step.
- **Always a Commit:** From the moment you initialize a `jj` repository, a working copy commit exists. As you make changes, this commit continuously evolves, representing the current state of your project.

- **Parent-Child Relationship:** Your working copy commit always has a parent. When you finalize changes into a new, "permanent" commit, your working copy automatically "moves" to become a new, empty commit with your just-created commit as its parent. It's like a dynamic pointer to your current workspace.

Why This Model Matters for Developers

This design isn't just a quirky difference; it's a deliberate choice that simplifies many common development challenges and empowers `jj`'s unique capabilities:

- **Streamlined Workflow:** By eliminating the staging area, `jj` reduces cognitive load. You focus purely on making and describing changes, rather than managing an intermediate state. This means less friction and more flow.
- **Natural Mutable History:** Since your working copy is inherently a commit, modifying history (like amending a previous commit or reordering changes) feels incredibly natural. You're simply editing a live commit, much like you edit a file in your editor. This is a game-changer for iterative development and code reviews.
- **Foundation for Stacked Changes:** This model is the bedrock of `jj`'s branchless workflow. It allows you to easily build a "stack" of dependent commits, where each commit builds logically on the previous one. Your working copy is just the latest, evolving step in that continuous stack. This dramatically simplifies complex feature development and refactoring.
- **Undo Capabilities:** Because every change to your working copy is tracked as part of its evolution, `jj` can offer powerful undo capabilities that are far more granular and reliable than traditional VCS. We'll explore this in the next chapter!

Visualizing the Working Copy Flow

Let's visualize how your working copy commit evolves as you make changes and finalize them into your repository's history.

In this diagram:

- `A` represents your starting point, a fresh directory.
- `C` is the working copy commit (`@`), which `jj` creates upon initialization. It's initially empty.
- When you `Edit Files` (D), your working copy commit (`@`) immediately reflects these changes (E). There's no separate `add` step.
- `F` represents running `jj commit`. This action takes the changes currently in `E` and finalizes them into a new, permanent commit (G).

- Crucially, `H` shows that `jj` then automatically creates a new, empty working copy commit (`@`) on top of `G`. This new `@` is ready for your next set of changes, maintaining the continuous "working copy as a commit" flow.

Step-by-Step: Your First Jujutsu Repository

Let's get hands-on. We'll create a new project, initialize a `jj` repository, make some changes, and see this "working copy as a commit" model in action.

1. Create a Project Directory

First, let's set up a clean workspace for our new `jj` project.

```
mkdir my-first-jj-repo
cd my-first-jj-repo
```

2. Initialize the Jujutsu Repository

Now, let's transform this ordinary directory into a `jj` repository. This command is conceptually similar to `git init`.

```
jj init
```

You should see output indicating success:

```
Initialized empty Jujutsu repository in "my-first-jj-repo" at ...
```

What just happened? `jj` created a hidden `.jj` directory within `my-first-jj-repo`. This directory is where `jj` stores all its internal repository data, including your history, the operation log, and configuration.

3. Inspect the Current State

Even though we haven't created any files yet, `jj` has already established our working copy commit. Let's inspect it using `jj status`.

```
jj status
```

You'll see output similar to this:

```
Working copy: 588f98c8c6d7 (empty) (no description set)
Parent commit: zzzzzzzz (empty) (no description set)
```

Let's break down this output, which is fundamental to understanding `jj`:

- **`Working copy: 588f98c8c6d7 (empty) (no description set)`**: This line describes your active working copy commit.
  - `588f98c8c6d7`: This is a unique, short identifier (a commit ID, similar to a Git SHA) for your working copy commit. Your specific ID will be different.
  - `(empty)`: This indicates that your working directory currently contains no files or changes relative to its parent.
  - `(no description set)`: We haven't given this ephemeral working copy commit a message yet.
- **`Parent commit: zzzzzzzz (empty) (no description set)`**: This describes the parent of your working copy commit. In a newly initialized `jj` repository, the working copy's parent is a special, immutable "root" commit. It's always empty and often represented by all `z`'s. Think of it as the ultimate ancestor of all your changes.

To see a more historical view, we can use `jj log -r @`. The `-r @` tells `jj log` to show the working copy commit and its ancestors.

```
jj log -r @
```

Output:

```
@  588f98c8c6d7 (empty) (no description set)
o  zzzzzzzz (empty) (no description set)
```

This log output clearly shows `@` (our working copy commit) on top, with its parent being the root commit.

4. Create Your First File

Now, let's create a simple text file. Remember, there's no `jj add`!

```
echo "Hello, Jujutsu! This is my first file." > hello.txt
```

5. Check the Status Again

Let's immediately check `jj status` to see how `jj` perceives this change without any explicit "add" command.

```
jj status
```

Output:

```
Working copy: 588f98c8c6d7 (no description set)
Added hello.txt
Parent commit: zzzzzzzz (empty) (no description set)
```

Notice the difference! The `(empty)` label is gone from your working copy description, and `Added hello.txt` clearly indicates that `jj` has recognized and included your new file within the working copy commit. Your working copy commit is no longer empty; it now contains `hello.txt`.

6. "Commit" Your Changes

To finalize the changes currently present in your working copy commit into a permanent part of your history, you use the `jj commit` command.

```
jj commit -m "Initial commit: Add hello.txt with a greeting"
```

Output:

```
Working copy: 1c3a7b9319e7 (empty) (no description set)
Committed as 3e9d4a2b1f0e "Initial commit: Add hello.txt with a greeting"
```

This output reveals the core mechanism of `jj commit`:

- **`Committed as 3e9d4a2b1f0e "Initial commit: Add hello.txt with a greeting"`**: Jujutsu took the current state of your previous working copy commit (the one that included `hello.txt`) and created a new, immutable commit with the ID `3e9d4a2b1f0e` (your ID will differ) and the message you provided. This is now a stable part of your repository's history.
- **`Working copy: 1c3a7b9319e7 (empty) (no description set)`**: After creating the new commit, `jj` automatically "moved" your working copy. It is now a brand new, empty commit that sits on top of the commit you just finalized. This new `@` is ready for your next set of changes, maintaining the continuous flow.

Let's confirm this new state with `jj log`. This time, we'll use `-r @-` to show the parent of the working copy, giving us a view of our actual history.

```
jj log -r @-
```

Output:

```
@  1c3a7b9319e7 (empty) (no description set)
o  3e9d4a2b1f0e Initial commit: Add hello.txt with a greeting
o  zzzzzzzz (empty) (no description set)
```

Fantastic! You now have a permanent commit (`3e9d4a2b1f0e`) in your history, representing the state where `hello.txt` was added. And your working copy (`@`) is patiently waiting, empty and ready for your next creative burst, sitting directly on top of that new commit.

Mini-Challenge: Evolving and Amending Your First Commit

Now that you've created a commit, let's put `jj`'s mutable history to the test by modifying that commit. This is where `jj` truly shines compared to Git's default immutable mindset.

Challenge:

1. Add a new line of text to `hello.txt`, perhaps asking a question.
2. Instead of creating a new commit on top, **amend** your previous commit (`3e9d4a2b1f0e`) to include this new line and update its commit message to reflect the change.

Hints:

- Remember, your current working copy (`@`) is always a commit.
- To modify the parent of your working copy (which is your "Initial commit"), you'll need a specific flag for the `jj commit` command. Think about how other VCS tools handle "amending" a previous commit.

```
# Your code here
```

💡 **STUCK? HERE'S A GUIDED APPROACH:**

First, let's add the new line to `hello.txt`:

```
echo "How are you enjoying Jujutsu so far?" >> hello.txt
```

Next, let's use `jj status` to confirm `jj` sees the modification:

```
jj status
```

You'll see `Modified hello.txt` listed under your working copy. Now, to amend the parent of your working copy (our "Initial commit"), we use `jj commit --amend`. This command takes the current changes in your working copy and applies them to its parent, effectively rewriting that parent commit.

```
jj commit --amend -m "Initial commit: Add greeting and a question for
Jujutsu"
```

Finally, let's check `jj log -r @-` again. You'll notice that the commit ID for your "Initial commit" has changed, and its message is updated!

```
jj log -r @-
```

(The output will show a new commit ID for the amended commit, and the new message. The old commit ID is gone from the immediate history.)

What to Observe/Learn:

You should have observed that `jj commit --amend` directly modified the content and message of the parent commit. This is a powerful demonstration of `jj`'s mutable history in action. Instead of creating a new commit on top and leaving the old one in history (which is what `git commit --amend` technically does, though the old commit becomes unreachable from branches), `jj` actually replaces the commit in your linear history. This results in a cleaner, more focused history that is much easier to manage, especially during iterative development and code review cycles. This direct mutation is a core differentiator from Git.

Common Pitfalls & Troubleshooting

Transitioning to `jj` from another VCS can sometimes lead to habits that don't quite fit `jj`'s model. Here are a few common pitfalls:

1. **Searching for `jj add`:** The most common instinct for Git users is to look for a staging command. Remember, `jj` automatically tracks all changes in your working directory. You don't need to `add` anything. Just make your changes, and then `jj commit` (or `jj describe` for the working copy itself) when you're ready to finalize them.
2. **Confusing the Working Copy with a "Finished" Commit:** The working copy commit (`@`) is a special, ephemeral, and constantly changing commit. It's your live workspace. When you run `jj commit`, you're essentially taking a snapshot of that working copy's state, finalizing it into a "real" commit, and then creating a new, empty working copy on top. Always remember `@` is your dynamic scratchpad.
3. **Misinterpreting `jj commit`'s Behavior:** In Git, `git commit` creates a new commit based on the staging area. In `jj`, `jj commit` creates a new commit based on the current state of your working copy. If your working copy is empty, `jj commit` will create an empty commit. If it has changes, those changes form the new commit. This distinction is subtle but important. If you want to describe the current working copy without creating a new commit, you'd use `jj describe`.

Summary

In this crucial chapter, you've taken your first significant steps with Jujutsu, grasping its most fundamental paradigm:

- You learned that in `jj`, your **working directory is always a commit** (`@`), which dynamically reflects your current file state. This eliminates the need for a separate staging area.
- You initialized your very first `jj` repository using `jj init`.
- You used `jj status` and `jj log` to inspect your working copy and its ancestral history.
- You created files and observed how `jj` automatically tracks these changes within your working copy.
- You finalized your working copy's changes into a permanent commit using `jj commit -m "message"`.

- You experienced `jj`'s powerful mutable history firsthand by using `jj commit --amend` to modify an existing commit, demonstrating how `jj` keeps your history clean and linear.

This "working-copy-as-a-commit" model is the beating heart of `jj`'s philosophy, enabling all the advanced mutable history, stacked changes, and branchless workflows we'll explore. In the next chapter, we'll dive into `jj`'s incredibly robust operation log and discover how you can undo almost any action, no matter how complex, giving you unparalleled safety and flexibility.

References

- Jujutsu GitHub Repository
- Jujutsu Official Tutorial (on main branch)
- Jujutsu Releases
- Jujutsu Concepts Documentation

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

CHAPTER 03

Unlocking Mutable History: Amending, Splitting, and Squashing Changes

Imagine your version control system not just as a rigid recorder of events, but as a flexible canvas where you can sculpt your work into a perfect narrative. Traditional systems often treat history as immutable once committed, making it a chore to refine your work after the fact. But what if you could easily fix mistakes, reorganize your thoughts, and present a pristine sequence of changes for review?

This is where Jujutsu (`jj`) truly shines. In this chapter, we'll dive deep into `jj`'s mutable history model. You'll learn how to refine your commit history with ease, transforming messy development into clean, logical steps. This ability is crucial for effective code reviews, simplifying debugging, and maintaining a healthy, understandable project history.

Before we begin, ensure you've set up `jj` as covered in Chapter 1 and understand the "working-copy-as-a-commit" concept from Chapter 2. We'll build on those fundamentals to unlock `jj`'s true power.

The Jujutsu Approach to History

Most developers are familiar with version control systems like Git, where commits are often perceived as set in stone once pushed. While Git offers tools like `rebase` to rewrite history, they can feel cumbersome and are often considered "advanced" or even dangerous by newcomers. Jujutsu approaches history differently, embracing mutability as a core, safe, and intuitive principle.

What is Mutable History?

In `jj`, every change you make is considered a potential commit that can be easily modified, reordered, or combined. This doesn't mean history is arbitrary or unstable; rather, it means `jj` provides powerful, built-in tools to refine your local development history before sharing it with others.

📌 **Key Idea:** In `jj`, your local history is a living document, constantly being shaped and improved, not a rigid, unchangeable record.

This concept extends directly from `jj`'s "working-copy-as-a-commit" model. Your current working directory is a commit (represented by `@`). Any changes you make in your working directory are implicitly modifying this special "working-copy commit." When you run `jj commit`, you're essentially creating a new commit that contains the changes from your current working directory, and the previous working directory commit becomes its parent. Crucially, you can always go back and change any of these commits, past or present, with ease and safety.

Why Mutable History is a Game-Changer for Engineers

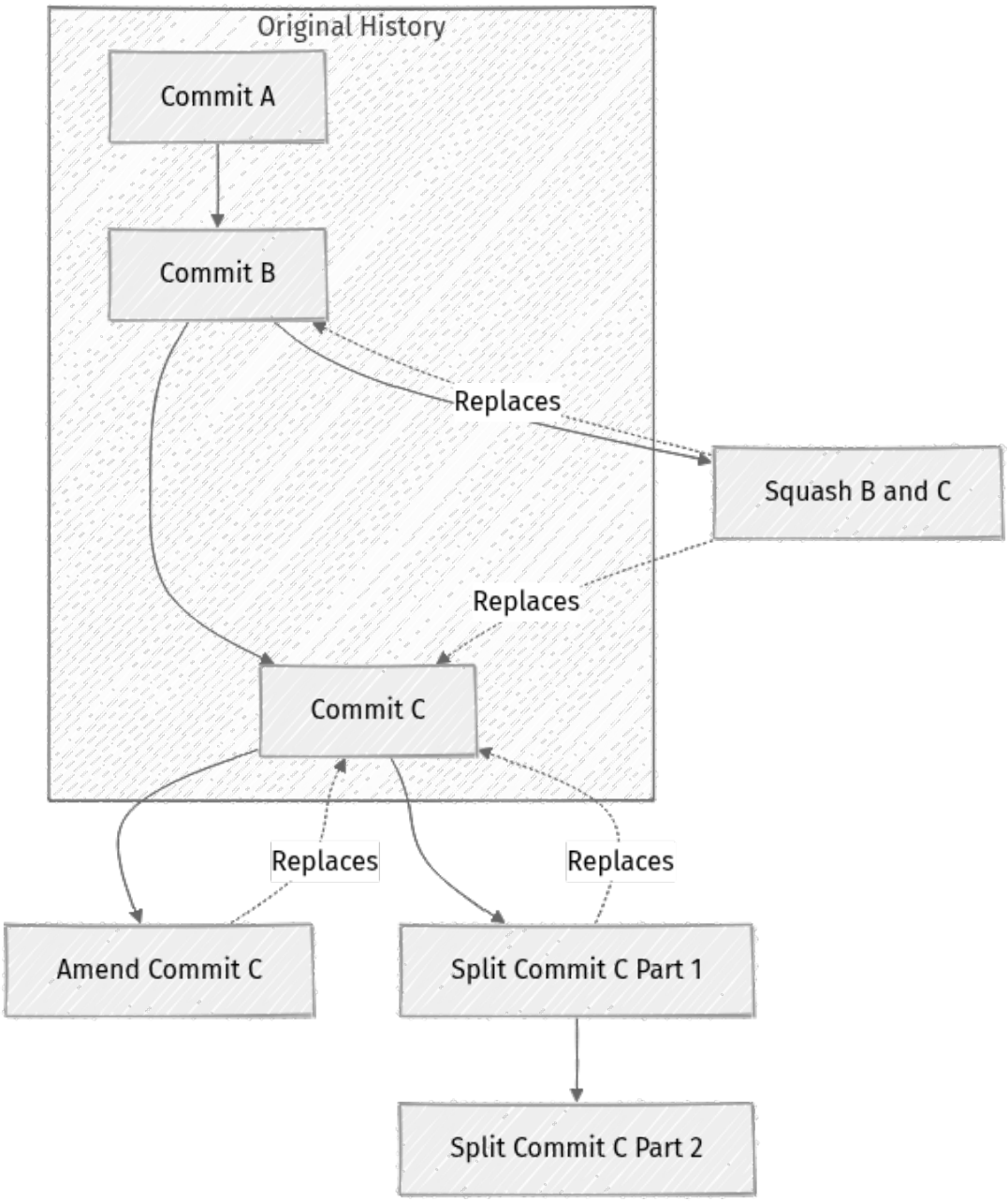
The ability to easily manipulate history provides significant advantages in modern software engineering workflows:

- **Cleaner Code Reviews:** Presenting atomic, logically separated commits makes code reviews faster and more effective. Reviewers can focus on one change at a time, reducing cognitive load and speeding up approval cycles.
- **Simplified Debugging:** A linear, well-structured history with clear, focused commit messages makes tools like `jj bisect` (or `git bisect` when interoperating) incredibly powerful for finding the exact change that introduced a bug. You can quickly pinpoint the culprit.
- **Seamless Refactoring:** You can refactor code across multiple commits, then easily combine or reorder them to tell a coherent story, without losing the intermediate steps you took during development. This encourages cleaner codebases.
- **Reduced Merge Conflicts:** By keeping your local history clean and frequently rebased onto the main branch, you minimize the surface area for complex merge conflicts when integrating your work. This saves valuable time and reduces frustration.

⚡ **Real-world insight:** Many development teams enforce strict code review guidelines that require atomic, well-described commits. Jujutsu's mutable history makes it effortless to meet these standards before pushing your changes for review, improving team collaboration and code quality.

Visualizing History Transformations

Let's visualize how `jj` commands intuitively transform your commit history. Each operation replaces or modifies existing commits, creating a new, cleaner sequence.



This diagram illustrates how basic operations like amending, splitting, and squashing effectively replace or modify parts of your commit graph, making it cleaner and more logical. Each dashed arrow indicates a replacement or transformation.

Step-by-Step Implementation: Hands-On History Sculpting

Let's get hands-on and see `jj`'s mutable history in action. We'll start with a simple repository and progressively refine its history.

First, let's create a new `jj` repository for our experiments:

```
mkdir jj_mutable_history_demo
cd jj_mutable_history_demo
jj init
```

Now, let's add some initial content and make a few commits.

1. Making Initial Commits

We'll start with two simple commits to establish some history.

Step 1.1: Create `feature_a.txt`

```
echo "This is the first line of Feature A." > feature_a.txt
jj branch create my-feature # Create a branch to track our progress easily
jj commit -m "feat: initial setup for feature A"
```

You should see output similar to this, indicating a new commit was created:

```
Working copy now at: 3723793e82b7 (empty) feat: initial setup for feature A
Added 0 files, modified 1 files, removed 0 files
```

Step 1.2: Add more to `feature_a.txt` and a new file

```
echo "Adding a second line for Feature A." >> feature_a.txt
echo "This is a utility script." > utils.py
jj commit -m "feat: add more to feature A and a utility script"
```

Now, let's inspect our history using `jj log`. This command shows the commit graph, with `@` indicating your current working-copy commit.

```
jj log
```

You'll see something like this (commit IDs will differ, but the structure is similar):

```
o  5041934c5b my-feature (empty) feat: add more to feature A and a utility
script
|  @  3723793e82 (empty) feat: initial setup for feature A
| /
o  0000000000 (empty) (no description)
```

The commit `5041934c5b` is the one you just created, and it's the tip of your `my-feature` branch. Your working copy (`@`) is currently on a commit that has `5041934c5b` as its parent.

2. Amending the Latest Change (`jj amend`)

It's common to realize you forgot a small detail or introduced a typo immediately after committing. `jj amend` is perfect for incorporating these last-minute changes into the current working-copy commit's parent, effectively updating the "latest" logical commit.

Step 2.1: Make a small correction

Let's say we forgot to add a crucial comment to `feature_a.txt`.

```
echo "# Important comment for feature A" >> feature_a.txt
```

Step 2.2: Amend the previous commit

Now, instead of creating a new commit, we'll add this change to the latest commit that `my-feature` points to. This commit is the parent of your current working copy (`@`).

```
jj amend
```

`jj` will apply the changes from your working directory to the parent of your working-copy commit. The original commit's ID will be replaced with a new one that contains the combined changes.

Let's check `jj log` again:

```
jj log
```

Notice that the commit ID for the `my-feature` branch's tip has changed (e.g., from `5041934c5b` to something new). The content of that commit now includes your latest addition, but it's still logically a single commit. This is a core aspect of `jj`'s mutable history – commit IDs change when their content or parentage changes.

3. Splitting a Commit (`jj split`)

Sometimes, you accidentally lump unrelated changes into a single commit. This makes code reviews harder and history less clear. `jj split` allows you to break a commit into two or more distinct, logically separated commits.

Step 3.1: Introduce unrelated changes

Let's modify `feature_a.txt` and `utils.py` in a single go, then commit them.

```
echo "Refined logic for Feature A." >> feature_a.txt
echo "Added a new helper function." > utils_helper.py # Renamed for clarity
jj commit -m "refactor: improve feature A and add helper in utils"
```

Your `jj log` now shows this new combined commit:

```
jj log
```

Step 3.2: Split the combined commit

We want to separate the `feature_a.txt` changes from the `utils_helper.py` changes. We'll split the latest commit, which is currently the parent of our working copy (`@`).

```
jj split @
```

This command opens your configured editor (e.g., `vi`, `nano`, `VS Code`) with the diff of the commit you're splitting.

Here's how to interact with the editor:

- The diff is presented in a special format. Lines starting with `pick` indicate changes that will go into the first resulting commit.
- Lines starting with `(empty)` indicate changes that will go into subsequent commits.
- To move a change to a new commit, you need to change its `pick` tag to `(empty)`.

For our example, the editor might show something like this (actual content depends on your `jj` version and configuration):

```
# You are splitting the commit <commit_id> (e.g., d8e3f4a5b6)
#
# Lines starting with "pick" are included in the first commit.
# Lines starting with "(empty)" are included in a new subsequent commit.
#
# To move a change to a new commit, change its "pick" to "(empty)".
#
# Example:
# pick:
#   - files:
#       - src/main.rs
#     hunks:
#       - 1
# (empty):
#   - files:
#       - src/lib.rs
```

```
#     hunks:
#       - 1
#
# Hunks in commit:
#   feature_a.txt:
#     1: +Refined logic for Feature A.
#   utils_helper.py:
#     1: +Added a new helper function.

pick:
  - files:
      - feature_a.txt
    hunks:
      - 1 # The hunk with "+Refined logic for Feature A."
(empty):
  - files:
      - utils_helper.py
    hunks:
      - 1 # The hunk with "+Added a new helper function."
```

In the editor, we've already set it up so that the `utils_helper.py` changes are in an `(empty)` block, and the `feature_a.txt` changes remain in `pick`. Save and close the editor. `jj` will then prompt you for a commit message for the new commit(s).

First, it will ask for the message for the first commit (containing `feature_a.txt` changes). Let's use: `refactor: improve feature A logic`. Then, it will ask for the message for the second commit (containing `utils_helper.py` changes). Let's use: `feat: add new helper function to utils`.

After saving the commit messages, run `jj log`:

```
jj log
```

You'll see two new commits replacing the original combined one, each with its specific message and changes. The original commit's ID is gone, replaced by two new ones.

4. Squashing Changes Together (`jj squash`)

The opposite of splitting, `jj squash` allows you to combine multiple commits into a single, more comprehensive commit. This is incredibly useful for cleaning up "work-in-progress" commits or grouping related changes that were initially separated.

Step 4.1: Make a few small, related commits

Let's make some minor UI adjustments as separate commits that we intend to combine later.

```
echo "UI element 1 added." > ui_changes.txt
jj commit -m "feat: add first UI element"

echo "UI element 2 added." >> ui_changes.txt
jj commit -m "feat: add second UI element"

echo "UI element 3 added." >> ui_changes.txt
jj commit -m "feat: add third UI element"
```

Now, `jj log` will show these three new commits on top of your history.

Step 4.2: Squash the UI commits

We want to combine these three UI commits into one logical commit. A common pattern is to squash the current commit (`@`) and its immediate parents into a single commit. We'll squash `@` into its parent, `@-`.

```
jj squash @-
```

The `@-` represents the parent of the current working-copy commit. This command takes the changes from `@` and combines them with `@-`. `jj` will then prompt you to edit the commit message.

`jj` will open your editor, allowing you to combine or edit the commit messages of the squashed commits. It will show the messages of both commits you are squashing. Edit it to a single, coherent message, like: `feat: implement all UI elements`.

After saving the message, run `jj log`:

```
jj log
```

You'll now see that the three separate UI commits have been replaced by a single commit with your new message, containing all the changes from the original three. The history looks much cleaner!

5. Reordering Changes (`jj rebase`)

`jj rebase` is a powerful command for changing the parent of a commit, effectively moving it and its descendants to a new location in the history. This is vital for maintaining a linear history and organizing your work, especially when integrating with a main branch or preparing for code review.

Step 5.1: Create a scenario for reordering

Let's imagine you made an initial structural commit, then a bug fix, and then a new feature. However, you decide the bug fix should logically come after the new feature, perhaps because the feature exposes the bug, or the fix is only relevant once the feature is integrated.

```
# First commit: Initial structure
echo "Initial project setup." > project_init.txt
jj commit -m "feat: initial project structure"

# Second commit: A bug fix
echo "Fixed a minor issue." > bug_fix.txt
jj commit -m "fix: address minor bug"

# Third commit: New feature
echo "Implemented a key new feature." > new_feature.txt
jj commit -m "feat: introduce exciting new feature"
```

Your `jj log` will now show the history like this (top to bottom, most recent at the top):

- `feat: introduce exciting new feature` (current `@`)
- `fix: address minor bug` (`@-`, the parent of `@`)
- `feat: initial project structure` (`@--`, the grandparent of `@`)

Step 5.2: Reorder the commits

We want the `fix: address minor bug` commit to appear after `feat: introduce exciting new feature`. To achieve this, we can rebase the `fix: address minor bug` commit (`@-`) onto the `feat: introduce exciting new feature` commit (`@`).

```
# Rebase the 'fix: address minor bug' commit (@-)
# onto the 'feat: introduce exciting new feature' commit (@)
jj rebase -s @- -d @
```

This command means: take the commit at `@-` (our "fix" commit) and rebase it so its new parent is `@` (the "feature" commit).

Now, check `jj log`:

```
jj log
```

You'll see the history has been reordered. The "feature" commit now appears before the "fix" commit. The order will be (top to bottom):

- `fix: address minor bug` (new `@`)

- `feat: introduce exciting new feature` (new `@-`)
- `feat: initial project structure` (new `@--`)

The commit IDs will have changed, reflecting the new parentage.

🧠 **Important:** `jj rebase` is incredibly flexible. You can rebase single commits, ranges of commits, or entire branches onto any other commit using `revsets`. We'll explore `revsets` in more detail in a later chapter, but for now, remember that `jj rebase -s <source_commit> -d <destination_commit>` is your go-to for moving things around. The `-s` flag specifies the "source" commit(s) to be rebased, and `-d` specifies the "destination" commit to be the new parent.

Mini-Challenge: Refine a Feature Branch

You're working on a new feature. You've made some progress, but your commit history is a bit messy. Let's clean it up!

Challenge:

1. Initialize a new `jj` repository or clear your current one (`rm -rf .jj` and `jj init`).
2. Create a file named `feature_x.js`.
3. **Commit 1:** Add a basic function definition to `feature_x.js`. Commit with message: `"feat: initial function skeleton"`

```
// feature_x.js
function calculateSum(a, b) {
  return a + b;
}
```

1. **Commit 2:** Add implementation details to the function, but also accidentally add a commented-out line that says `// TODO: remove this later`. Commit with message: `"feat: implement core logic"`

```
// feature_x.js
function calculateSum(a, b) {
  // TODO: remove this later
  // Added some complex logic
  let result = a + b;
  if (result > 100) {
    result *= 0.9; // Apply discount
  }
  return result;
```

```
}
```

1. **Commit 3:** Add more implementation details, and also fix a typo in `feature_x.js` from Commit 1 (e.g., change `calculateSum` to `sumNumbers`). Commit with message: `"feat: add more logic and fix typo"`

```
// feature_x.js
// Typo fix: Renamed function
function sumNumbers(a, b) {
  // TODO: remove this later
  // Added some complex logic
  let result = a + b;
  if (result > 100) {
    result *= 0.9; // Apply discount
  }
  // Even more logic
  return result + 5;
}
```

1. **Your Task:**
    - **Amend** the "feat: implement core logic" commit (Commit 2) to remove the `// TODO: remove this later` line.
    - **Split** the "feat: add more logic and fix typo" commit (Commit 3) into two commits: one for "add more logic" and another for "fix typo".
    - **Squash** the "initial function skeleton" and the amended "implement core logic" commits into a single "feat: implement feature X" commit.
    - Ensure your final `jj log` shows a clean, logical history for `feature_x.js` with distinct commits like:
        - `fix: rename calculateSum to sumNumbers`
        - `feat: add more logic to sumNumbers`
        - `feat: implement feature X (initial skeleton + core logic)`

Hint: Use `jj log -r @` to always know your current commit. Remember `jj split @` will split the parent of `@`. `jj squash @-` will squash the current commit into its parent. If you get stuck, `jj undo` is your friend – it can revert any `jj` operation!

What to observe/learn: This challenge will help you internalize how these commands change the history and how `jj` guides you through the process, especially with interactive prompts for splitting and squashing. Pay attention to how commit IDs change and how the `jj log` output reflects your desired, clean history.

Common Pitfalls & Troubleshooting

Working with mutable history is incredibly powerful, but it comes with a few things to keep in mind, especially when migrating from a Git mindset.

- **Commit IDs Change Frequently:**
  - `jj` generates new commit IDs whenever a commit's content or parentage changes. This is fundamental to how `jj` tracks history. Don't be alarmed if IDs you've seen before are replaced; it means you've successfully refined your history. The old (now "hidden") commits are still in the operation log, so they aren't truly lost.
  - **Solution:** Always refer to commits by their short prefixes (e.g., `abcd`) or by `revsets` like `@` (current working-copy commit), `@-` (parent), `my-branch` (branch tip), or `main` (main branch). Avoid relying on full, long IDs for anything other than specific, one-off references.
- **Forgetting `jj log`:**
  - It's easy to get lost in your history, especially when actively manipulating it. The visual graph `jj log` provides is your map.
  - **Solution:** Make `jj log` (or `jj log -r @` for just the current commit and its ancestors) your best friend. Use it frequently to visualize your changes and confirm your operations. The `jj status` command is also useful to see your current working directory changes.

- **Over-editing and Losing Track:**
  - ⚠️ `What can go wrong:` While `jj` makes history manipulation easy, constantly changing things without a clear goal can lead to a confusing state. You might make changes, then realize you don't remember the exact sequence of operations that led you there.
  - **Solution:** Plan your history changes. If you make a mistake or get into a state you don't understand, remember the `jj operation log` (covered in the next chapter!) and `jj undo`. These are your ultimate safety nets for any history-rewriting operation. They allow you to step back through every action you've taken.
- **Misinterpreting `jj amend` vs. Git's `commit --amend`:**
  - In Git, `git commit --amend` typically amends the parent of your current `HEAD`. In `jj`, `jj amend` amends the parent of the current working-copy commit (`@`). This distinction is crucial if you're used to Git's model.
  - **Solution:** Always think of `jj amend` as updating the commit that `jj` considers your "latest" completed work. If you want to amend an ancestor further down the history, you'd typically use `jj rebase -i` (interactive rebase, which we'll cover later) or combine `jj edit` with `jj amend` for more advanced scenarios. For now, `jj amend` is for the immediate previous commit.

Summary

Congratulations! You've taken a significant step into mastering Jujutsu's powerful mutable history. You now understand how to:

- **Amend** your latest commit to include new changes or fixes, seamlessly integrating them without creating new, small commits.
- **Split** a single commit into multiple, more focused commits, improving clarity for code reviews and debugging.
- **Squash** several related commits into one coherent change, cleaning up "work-in-progress" history.
- **Reorder** commits to create a linear, logical flow using `jj rebase`, ensuring your history tells a clear story.

These tools empower you to sculpt your local history into a clean, review-ready state, making you a more efficient and effective developer. You're no longer just recording history; you're actively crafting it.

In the next chapter, we'll explore `jj`'s incredible **operation log** and **undo capabilities**. This feature is `jj`'s ultimate safety net, allowing you to review and revert any action you've taken, making fear of history rewriting a thing of the past.

References

- Jujutsu GitHub Repository
- Jujutsu Official Tutorial (GitHub Docs)
- Jujutsu Command Reference (`jj help`)
- Jujutsu Releases (for latest stable version)

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

CHAPTER 04

The Superpower of Undo: Navigating the Operation Log

Have you ever made a change in your version control system, only to realize a few steps later that you've gone down the wrong path? Perhaps you accidentally squashed commits, rebased incorrectly, or simply wish you could rewind to a previous state without losing your work. In traditional VCS like Git, recovering from such scenarios can range from trivial to terrifying, often involving arcane commands or the dreaded "force push."

`jj` (Jujutsu) offers a profound solution to this common developer anxiety: the **operation log**. This isn't just a simple history of your commits; it's a comprehensive record of every action you take that modifies your repository's state. Think of it as a super-powered undo stack for your entire VCS workflow. With the operation log, you can fearlessly experiment, knowing that you can always rewind to any previous point in your repository's history with a single command. This chapter will dive deep into this safety net, showing you how to explore, undo, and even redo your way to a more confident development experience.

Before we begin, ensure you've set up `jj` and have a basic understanding of its core commands, as covered in previous chapters. We'll be building on that foundation to manipulate repository history with unprecedented ease.

The Operation Log: Your Repository's Memory

In `jj`, every command you run that changes the repository's state – whether it's creating a commit, amending one, rebasing, or even resolving a conflict – is recorded as an "operation." These operations are stored in the **operation log**, an immutable, linear history of how your repository has evolved.

📌 **Key Idea:** While Git's reflog tracks changes to references (like branch pointers), `jj`'s operation log tracks changes to the entire repository state, including all commits, working copy, and branches. This fundamental difference is what makes `jj`'s undo capabilities so powerful and comprehensive.

Why the Operation Log Matters

The operation log provides several critical benefits for engineers, transforming the way you interact with your project's history:

- **Fearless Experimentation:** Try out complex rebases, merges, or history rewrites without worry. If something goes wrong, you can simply undo it. This boosts productivity by removing the fear of breaking things and encourages more creative problem-solving.
- **Robust Recovery:** Easily revert to any past state of your repository, effectively undoing multiple `jj` commands in one go. This is invaluable for recovering from mistakes, exploring alternative solutions, or even going back to a stable point for debugging.
- **Audit Trail:** Understand exactly how your repository reached its current state, step by step. Each operation is logged with its command, timestamp, and author, making it invaluable for debugging or reviewing complex history manipulations, especially in team environments.
- **Collaboration Safety:** If a team member accidentally pushes an undesirable change, you can often use the operation log to revert your local state to before that operation, then interact with the remote again, avoiding complex `git revert` or `git reset --hard` scenarios.

Step-by-Step: Exploring and Manipulating Your Operations with the Log

Let's generate some operations and then learn how to navigate them.

First, make sure you're in a `jj` repository. If you don't have one, create a new one:

```
jj init my_project_with_ops
cd my_project_with_ops
```

Now, let's make a series of changes to generate a meaningful operation log.

Generating Operations

1. **Create an initial commit with content:**

   We'll start by adding a `greeting.txt` file and committing it. The `jj new -m "..."` command creates a new commit and moves your working copy to it. This first command creates an operation.

```
echo "Hello, Jujutsu!" > greeting.txt
jj new -m "Initial commit with greeting"
```

1. **Create a `main` branch:**

   Next, we'll create a named branch reference. This action is also recorded as an operation.

```
jj branch create main
```

1. **Make another change and commit:**

   Let's add more content to `greeting.txt` and create a new commit. This commit will be stacked on top of the previous one, generating another operation.

```
echo "Welcome to the operation log tutorial." >> greeting.txt
jj new -m "Added welcome message"
```

1. **Amend the last commit (both content and message):**

   The `jj amend` command modifies the current commit in place, updating both its content (from the working directory) and its message. This is a common operation that rewrites history and creates a new operation in the log.

```
echo "Adding a new line for clarity." >> greeting.txt
jj amend --message "Updated greeting with a new line and clearer message"
```

Viewing the Operation Log

Now that we've performed a few actions, let's inspect the operation log using `jj op log`.

```
jj op log
```

You'll see output similar to this (IDs and timestamps will vary):

```
@  3a2f8b7e7a7e (2026-05-19 10:30:00) My Name <me@example.com>
   amend --message "Updated greeting with a new line and clearer message"
o  c1d3e2f1a0b9 (2026-05-19 10:29:00) My Name <me@example.com>
```

```
    new -m "Added welcome message"
o   9b8a7c6d5e4f (2026-05-19 10:28:00) My Name <me@example.com>
    branch create main
o   6f5e4d3c2b1a (2026-05-19 10:27:00) My Name <me@example.com>
    new -m "Initial commit with greeting"
o   5a4b3c2d1e0f (2026-05-19 10:26:00) My Name <me@example.com>
    init
```

Let's break down the output:

- `@` **(Current Operation):** The `@` symbol indicates your current operation. This is the state your repository is currently in.
- **Operation ID:** A unique hexadecimal ID for each operation (e.g., `3a2f8b7e7a7e`).
- **Timestamp and Author:** When and by whom the operation was performed.
- **Description:** A concise summary of the `jj` command that generated the operation.
- **Parents:** (Not explicitly shown in this default view, but operations can have parent operations, especially after `undo` or `redo`.)

Each line represents a distinct action you took. Notice how `jj new`, `jj branch create`, and `jj amend` are clearly listed with their arguments.

Inspecting a Specific Operation

You can dive deeper into any operation using `jj op show <op-id>`. Let's look at the operation where we amended the commit. Find the `op-id` for the `amend` operation from your `jj op log` output (it's typically the one marked with `@` or the one immediately before it if you've done something else).

```
jj op show 3a2f8b7e7a7e # Replace with your actual amend op-id
```

The output will be verbose, showing:

- Details about the operation itself (timestamp, user, command, parent operations).
- Crucially, a diff showing how the set of commits changed as a result of this operation. This includes which commits were added, removed, or modified.

This detailed view is incredibly powerful for understanding the full impact of any `jj` command.

Unleashing Undo and Redo

Now for the fun part: undoing and redoing operations.

`jj undo`: Rewind to the Previous State

The `jj undo` command is your primary tool for reversing operations. It effectively moves your repository back to the state before the last operation, making the previous operation the new current one.

Let's try it:

```
jj undo
```

After running this, `jj` will tell you what it undid. If you run `jj op log` again, you'll see that the `@` (current operation) has moved up one step, and the `amend` operation is no longer the current one. Your working copy will also reflect the state before the `amend`.

The file `greeting.txt` will now contain:

```
Hello, Jujutsu!
Welcome to the operation log tutorial.
```

The "Adding a new line for clarity." text is gone from the commit and the file. Amazing, right?

`jj redo`: Re-apply Undone Operations

What if you `undo` something and then realize you did want that change after all? No problem! `jj redo` will re-apply the last undone operation.

```
jj redo
```

Now, your `amend` operation is back in effect, and `greeting.txt` will again include "Adding a new line for clarity."

`jj restore --op <op-id>`: Pinpoint Recovery

`jj undo` and `jj redo` are great for stepping back and forth one operation at a time. But what if you want to jump back several operations, or restore to a specific point in the past without affecting more recent, unrelated operations? That's where `jj restore --op <op-id>` comes in.

This command takes your repository back to the state immediately after the specified operation ID was performed. It's like saying, "Make this operation the new current point in my history." Importantly, `jj restore --op` achieves this by creating a new operation in the log. This new operation's purpose is to declare that the repository state is now what it was after the specified `op-id`. This reinforces the immutable nature of the operation log itself; you're not deleting history, you're adding a new entry that points to an older state.

Let's try a more complex scenario:

1. **Add a new file and commit:**

```
    echo "This is a new feature." > feature.txt
    jj new -m "Added feature file"
```

1. **Add another file and commit:**

```
    echo "Configuration settings." > config.ini
    jj new -m "Added config file"
```

```
Now your `jj op log` will show these two new operations on top.
```

```
    jj op log
```

```
You'll see something like:
```

```
    @  new_op_id (time) My Name <me@example.com>
       new -m "Added config file"
    o  prev_new_op_id (time) My Name <me@example.com>
       new -m "Added feature file"
    o  3a2f8b7e7a7e (time) My Name <me@example.com>
       amend --message "Updated greeting with a new line and clearer message"
    ...
```

```
Let's say you want to go back to the state *after* the `amend` operation, but
*before* the `feature.txt` and `config.ini` commits. Find the `op-id` of your
`amend` operation (e.g., `3a2f8b7e7a7e`).
```

```
jj restore --op 3a2f8b7e7a7e # Use your amend op-id
```

```
`jj` will report that it restored to that operation. Check your working
directory: `feature.txt` and `config.ini` are gone! Your repository has been
completely reset to the state right after you amended `greeting.txt`.

This is incredibly powerful for discarding a series of experimental changes
without manually reverting or resetting.
```

🧠 Important: `jj restore --op` vs. `jj undo`

Understanding the subtle but crucial difference between these two commands is key to effectively navigating your `jj` history:

- **`jj undo`:** Reverses the last operation. It's a single step back, moving the `@` pointer to the immediate parent operation. It's typically used for quickly correcting the very last thing you did.
- **`jj restore --op <op-id>`:** Reverts the repository to the state after a specific operation, effectively discarding all operations that happened after the target `op-id`. It does this by creating a new operation that points to that specific past state. This is like jumping to an arbitrary point in your operation history, making it ideal for larger rewinds or targeted recovery.

Mini-Challenge: Precise History Rewind

You've been working on a new feature and made a few changes. Let's simulate that:

1. Create a file `data.json` with some JSON content and commit it.

```
echo '{"version": 1}' > data.json
jj new -m "Initial data.json"
```

1. Add a new line to `greeting.txt` and commit it.

```
echo "This is an important update." >> greeting.txt
jj new -m "Added important update to greeting"
```

1. Rename `data.json` to `config.json` and commit it.

```
jj mv data.json config.json
jj new -m "Renamed data.json to config.json"
```

Now, your challenge is to **undo only the renaming of `data.json` to `config.json`**, reverting that specific operation, while keeping the other changes (the new line in `greeting.txt` and the initial content of `data.json` under its original name).

Hint: Use `jj op log` to identify the operation ID for the rename. Then, identify the operation ID just before the rename operation occurred. This is the state you want to restore to using `jj restore --op`.

What to observe/learn: The precision with which `jj` allows you to manipulate your operational history. You're not just undoing the last thing; you're surgically reverting to a specific state.

Common Pitfalls & Troubleshooting

1. **Confusing `jj undo` with Git's `git revert` or `git reset`:**
    - `jj undo` literally rewinds your entire repository state (working copy, commits, branches) to a previous point in the operation log. It's an undo of your actions, providing a true "oops" button.
    - `git revert` creates a new commit that undoes the changes of a previous commit. It doesn't rewrite history.
    - `git reset` moves your branch pointer and potentially discards commits, but it's often more destructive and less easily reversible than `jj undo`.
    - **Pro Tip:** Always think of `jj undo` as a true "undo" button for your `jj` commands, affecting your entire repository state.
2. **Not inspecting the operation log before `undo` or `restore`:**
    - Blindly running `jj undo` can sometimes be confusing if you don't remember the exact sequence of your last few commands.
    - **Best Practice:** Always run `jj op log` first to get your bearings and identify the `op-id` you want to target, especially when using `jj restore --op`. This prevents unexpected state changes.
3. **Losing track of complex operation history:**
    - The operation log can grow long, making it hard to find a specific point.
    - **Solution:** Use `jj op log -n <count>` to show only the last `n` operations. For example, `jj op log -n 5` shows the 5 most recent operations. You can also pipe `jj op log` to `grep` if you remember keywords from the operation descriptions (e.g., `jj op log | grep "amend"`).
4. **What happens if you undo an undo?**
    - When you run `jj undo`, Jujutsu creates a new operation in the log. This new operation's purpose is to revert your repository to the state of the previous operation. If you then run `jj undo` again, you are undoing that undo operation. This means `jj` will move your `@` pointer back one more step in the log, effectively reapplying the change that the first `undo` had reversed. It feels like a `redo` because you're moving past the `undo` operation in the linear log history.

Summary

The operation log is one of `jj`'s most powerful and distinguishing features. By keeping a comprehensive, immutable record of every state-changing command, `jj` provides:

- A safety net for all your development activities, encouraging fearless experimentation and reducing anxiety.
- Precise control over your repository's history, allowing you to undo and redo operations with ease.
- The ability to jump to any past state using `jj restore --op`, making complex history corrections simple and efficient.

Mastering the operation log transforms your VCS experience from a cautious dance around immutable history to a confident exploration of possibilities. You can now manipulate your project's past without fear, knowing that `jj` has your back.

In the next chapter, we'll build on this newfound control by exploring `revsets`, `jj`'s powerful syntax for selecting and filtering commits. This will allow you to precisely target the commits you want to manipulate, further enhancing your ability to craft perfect history.

References

- Jujutsu Official GitHub Repository: https://github.com/jj-vcs/jj
- Jujutsu Tutorial - The Operation Log: https://github.com/martinvonz/jj/blob/main/docs/tutorial.md#the-operation-log
- Jujutsu Commands Documentation: https://github.com/jj-vcs/jj/blob/main/docs/commands.md

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

CHAPTER 05

Precision Navigation: Introduction to Revsets

Imagine you're navigating a vast ocean of changes, with commits appearing and disappearing as you refine your work. How do you pinpoint that one crucial commit? How do you select a specific range of changes for a rebase or a diff? In Jujutsu (`jj`), this precision navigation is handled by a powerful query language called **Revsets**.

This chapter will introduce you to the world of Revsets, `jj`'s flexible and intuitive way to refer to specific commits or groups of commits. We'll explore the fundamental building blocks of Revsets, from simple references to complex queries, and show you how to use them to interact with your repository's history with surgical accuracy. Mastering Revsets is crucial for leveraging `jj`'s mutable history, enabling you to manipulate your changes effectively and adopt advanced branchless workflows.

Before we dive in, ensure you're comfortable with basic `jj` commands like `jj init`, `jj commit`, `jj log`, and the concept of the working-copy-as-a-commit from previous chapters. Revsets build directly on these foundational ideas, allowing you to specify which commits these commands should operate on.

What Are Revsets and Why Do They Matter?

The Challenge of Referring to Commits

In traditional Version Control Systems (VCS) like Git, you often refer to commits using their SHA-1 hashes, branch names, or relative references like `HEAD~3`. While functional, these methods can become cumbersome, especially when dealing with complex histories or when you need to select commits based on various criteria (e.g., by author, message, or file changes).

`jj`'s mutable history model, where commits are frequently rewritten, amended, and rebased, further amplifies this challenge. Commit hashes change, and traditional branch names might not always capture the fluidity of your development. You need a dynamic way to point to "this commit, or its parent, or all commits by me that changed this file."

Introducing Revsets: A Query Language for Commits

Revsets are `jj`'s elegant solution. They provide a concise yet powerful query language that allows you to:

- **Select individual commits:** By ID, by special keywords, or by their relationship to other commits.
- **Filter commits:** Based on properties like author, commit message, or changed files.
- **Combine commit sets:** Using logical operations like union, intersection, and difference.
- **Specify ranges:** To operate on sequences of commits.

Why does this matter? Revsets are the backbone of almost every advanced `jj` operation. Whether you're rebasing a stack of changes, squashing commits, inspecting history, resolving conflicts, or synchronizing with a Git remote, you'll use Revsets to tell `jj` which commits you want to operate on. They empower you to precisely target your actions, making complex history manipulations feel natural and intuitive.

📌 **Key Idea:** Revsets are `jj`'s flexible commit query language, essential for navigating and manipulating its mutable history. They provide a powerful way to describe "sets of commits" for any `jj` command.

Navigating Your History: Basic Revset Syntax

Let's get hands-on and explore the most common and fundamental Revset expressions. We'll use `jj log -r <revset>` to see what commits a Revset resolves to.

First, let's set up a small repository with a few commits. This will give us a tangible history to query. We'll assume you have a working development environment as of 2026-05-19.

```
# Start fresh: Remove any old repo named 'my-revset-repo'
rm -rf my-revset-repo

# Initialize a new jj repository
jj init my-revset-repo
cd my-revset-repo

# Configure author for consistent commit history
jj config set user.name "AI Expert"
jj config set user.email "ai@example.com"

# Create a first commit
```

```
echo "Hello, Revsets!" > file1.txt
jj commit -m "feat: initial commit"

# Create a second commit
echo "Adding another line to file1." >> file1.txt
echo "This is a new file." > file2.txt
jj commit -m "feat: add second line and file2"

# Create a third commit (this will be our working copy's parent)
echo "And a third line to file1." >> file1.txt
jj commit -m "feat: add third line to file1"

# Add some uncommitted changes to see the difference between . and @
echo "Uncommitted change." >> file1.txt
```

Now that our history is set, let's explore! Note that commit IDs in your output will differ from the examples.

1. The Working Copy and Current Commit

These are your immediate starting points, representing where you are right now.

- `.` **(Dot): The Working Copy Commit** The `.` (dot) Revset refers to the **working copy commit**. In `jj`, your working directory is always treated as a commit, even before you explicitly `jj commit`. This output shows the commit that would be created if you ran `jj commit` right now, including any uncommitted changes.

```
    jj log -r .
```

```
You should see output similar to this (commit ID and summary will vary):
```

```
    @  4f83a73c09e3 feat: add third line to file1 (uncommitted changes)
```

```
Notice the `(uncommitted changes)` part. This tells you that `.` includes the
current state of your working directory.
```

- `@` **(At Symbol): The Current Commit (Parent of Working Copy)** The `@` (at symbol) Revset refers to the **current commit**. This is the parent of your working copy commit (the commit your working directory is currently based on). If you have uncommitted changes, `@` refers to the commit before those changes. If you have no uncommitted changes, `.` and `@` will refer to the same commit.

```
jj log -r @
```

```
Output:
```

```
@  4f83a73c09e3 feat: add third line to file1
```

```
Here, the `(uncommitted changes)` is gone because `@` represents the
*committed* state your working directory is based on, not including your
current edits.

🧠 **Important:** `.` includes uncommitted changes; `@` refers to the *last
committed state* your working directory is based on. This distinction is
critical for operations that should or shouldn't involve your ephemeral work.
```

2. Ancestors and Descendants: Navigating the Family Tree

Once you have a reference to a commit, you can navigate its family tree.

- `-` **(Minus): Parent Commit** The `-` (minus) operator gives you the **parent** of the specified commit. So, `@-` means "the parent of the current commit".

```
jj log -r '@-'
```

```
Output (your IDs will differ):
```

```
o  3f1a0e91005b feat: add second line and file2
```

- `--` **(Double Minus): Grandparent Commit** You can chain `-` operators to go further back in history. `@--` means "the grandparent of the current commit".

```
jj log -r '@--'
```

```
Output:
```

```
o  a2b1c3d4e5f6 feat: initial commit
```

- **`root()`: The Initial Empty Commit** `root()` refers to the **initial, empty commit** of the repository. Every `jj` repository starts with this special commit, which has no parent. It's the ultimate ancestor.

```
jj log -r 'root()'
```

```
Output:
```

```
o  000000000000 (empty)
```

- **`A::B`: Inclusive Range of Commits** The `::` (double colon) operator specifies a **range of commits, including ancestors and descendants** between `A` and `B` (inclusive). `root()::@` means "all commits from the root commit up to and including the current commit". This is a powerful way to visualize a linear history segment.

```
jj log -r 'root()::@'
```

```
This will show your entire history from the `root()` commit up to your current
commit `@`.
```

```
@  4f83a73c09e3 feat: add third line to file1
o  3f1a0e91005b feat: add second line and file2
o  a2b1c3d4e5f6 feat: initial commit
o  000000000000 (empty)
```

- **`A..B`: Ancestor-Exclusive Range of Commits** The `..` (double dot) operator means "the set of commits that are ancestors of `B` but not ancestors of `A`." This is often used to get the commits between two points, effectively excluding the starting point and its ancestors.

  Let's use our history. Find the short ID of your `feat: initial commit`. We'll use `a2b1c3d4e5f6` as a placeholder.

```
    # Replace 'a2b1c3d4e5f6' with your actual initial commit ID
    jj log -r 'a2b1c3d4e5f6..@'
```

```
This command means: "Show all commits that are ancestors of `@` (current
commit) but are *not* ancestors of `a2b1c3d4e5f6` (initial commit)."
The output would include your second and third commits:
```

```
    @  4f83a73c09e3 feat: add third line to file1
    o  3f1a0e91005b feat: add second line and file2
```

```
It effectively selects all commits *after* `a2b1c3d4e5f6` up to and including
`@`.

⚡ **Quick Note:** The `A..B` syntax can be tricky. A simpler way to
understand it is that it selects all commits on the path from `A` to `B`,
*excluding `A` itself and its ancestors*. It's useful for getting a range of
commits that are *descendants* of `A` but *ancestors* of `B`. For simple
inclusive linear ranges, `A::B` is often more intuitive.
```

3. Filtering by Commit Properties

Revsets allow you to filter commits based on their metadata.

- **`author("pattern")`**: Filter by Author `author("pattern")` selects commits whose author name or email matches the given pattern (case-insensitive substring match). You can use regular expressions here.

```
    jj log -r 'author("AI Expert")'
```

```
This will show all commits by the "AI Expert" we configured.
```

```
    @  4f83a73c09e3 feat: add third line to file1
    o  3f1a0e91005b feat: add second line and file2
    o  a2b1c3d4e5f6 feat: initial commit
```

- **`description("pattern")`**: Filter by Commit Message `description("pattern")` selects commits whose commit message matches the pattern.

```
jj log -r 'description("line")'
```

```
This will show commits whose descriptions contain "line".
```

```
@  4f83a73c09e3 feat: add third line to file1
o  3f1a0e91005b feat: add second line and file2
```

- **`file("path")`**: Filter by Changed Files `file("path")` selects commits that changed the specified file. The path is relative to the repository root.

```
jj log -r 'file("file2.txt")'
```

```
This will show only the commit where `file2.txt` was introduced.
```

```
o  3f1a0e91005b feat: add second line and file2
```

4. Combining Revsets: Set Operations

The real power of Revsets comes from combining them using logical set operations. Think of each Revset expression as defining a "set" of commits.

- `A | B` (Union): Commits in A OR B.
- `A & B` (Intersection): Commits in A AND B.
- `A - B` (Difference): Commits in A BUT NOT B.
- `~A` (Complement): All commits not in A. (Use with caution, as "all commits" can be a very large set!)

Let's put these into practice:

- **Intersection (`&`): Commits matching BOTH criteria** Let's find all commits that touched `file1.txt` and had "line" in their description:

```
jj log -r 'file("file1.txt") & description("line")'
```

```
This should return the second and third commits, as they both modified
`file1.txt` and had "line" in their message.
```

```
    @  4f83a73c09e3 feat: add third line to file1
    o  3f1a0e91005b feat: add second line and file2
```

- **Difference (`-`): Commits in A, but NOT in B** Now, let's find all commits by "AI Expert" except for the initial commit. First, find your initial commit's short ID (e.g., `a2b1c3d4e5f6`).

```
    # Replace 'a2b1c3d4e5f6' with the actual short ID of your initial commit
    jj log -r 'author("AI Expert") - a2b1c3d4e5f6'
```

```
This should show only the second and third commits, as the initial commit is
subtracted from the set of all commits by "AI Expert".
```

5. Special Keywords for Common Scenarios

`jj` provides several keywords for common commit selections, offering shortcuts for powerful queries.

- **`heads()`**: All "Head" Commits `heads()` selects all commits that are "heads" (meaning they have no children, or all their children are hidden). In a linear history, this is usually your current commit. In a more complex graph with multiple lines of development, it would show the tips of all those lines.

```
    jj log -r 'heads()'
```

```
This will typically show your current commit (`@`) and any other independent
lines of development (e.g., if you had created a new branch and committed
there).
```

- **`latest(N)`**: The N Most Recent Visible Commits `latest(N)` selects the N most recent visible commits. This is useful for quickly looking at your very recent history.

```
    jj log -r 'latest(2)'
```

```
This will show the two most recent commits, including the working copy if it
has changes (as `.` is implicitly a "head").
```

- **`public_commits()`**: Commits Safe for Public Sharing `public_commits()` selects commits that have been pushed to a remote or are ancestors of such commits. This is crucial for understanding what's safe to rewrite without affecting collaborators, as `jj` generally recommends not rewriting public history.

```
jj log -r 'public_commits()'
```

```
Initially, in a new local repository, this will likely only show the `root()`
commit as nothing has been pushed yet.
```

This is just a glimpse! The `jj` official documentation provides a comprehensive list of Revset functions and operators, which you can explore as you become more comfortable.

Applying Revsets: Practical Examples

Revsets aren't just for `jj log`. They are integrated into almost every `jj` command, allowing you to specify exactly which commits an operation should act upon.

Scenario 1: Rebase a Specific Range of Commits

Imagine you have a stack of three commits, `A`, `B`, and `C` (where `C` is the current commit `@`). You've decided that commits `A` and `B` need to be rebased onto an older, stable commit `D` to incorporate some foundational changes.

First, let's simulate this by creating a new `D` commit further back in history. (We'll reset our working directory for this example to simplify things temporarily).

```
# Undo uncommitted changes for a clean state
jj restore --staged --from @
jj clean

# Go back to the initial commit's parent (the root)
jj new root()
echo "Old base content." > base_file.txt
jj commit -m "feat: stable base D"

# Now go back to our third commit to continue work
jj checkout @-
```

Now, imagine `D` is the `feat: stable base D` commit you just created. And `A` is our `feat: initial commit`. We want to rebase all commits from `A` through `@` onto `D`.

```
# First, find the short ID of your "feat: initial commit" (let's call it A_ID)
# And the short ID of your "feat: stable base D" (let's call it D_ID)
# Example:
# A_ID = a2b1c3d4e5f6 (your initial commit)
# D_ID = e1f2g3h4i5j6 (your stable base D commit)

# Rebase commits from A_ID (inclusive) up to the current commit (@) onto D_ID
jj rebase -s 'A_ID::@' -d D_ID
```

This command tells `jj` to take the set of commits starting from `A_ID` up to and including `@`, and reapply them on top of `D_ID`. This is a powerful way to restructure your history precisely.

Scenario 2: Diffing Changes Between Two Arbitrary Points

You want to see all changes introduced between your `feat: initial commit` (let's use `a2b1c3d4e5f6` again) and your current commit (`@`). This is useful for code reviews or debugging.

```
# Replace 'a2b1c3d4e5f6' with your actual initial commit ID
jj diff -r 'a2b1c3d4e5f6::@'
```

This command shows the combined diff of all changes from `a2b1c3d4e5f6` up to `@`. It's like asking, "What's the difference in state between these two points in history, considering all commits in between?"

Scenario 3: Finding Commits that Touched a File Before a Specific Point

You're debugging an issue and suspect a change to `file1.txt` introduced it, but you only care about changes before a specific commit, say `bugfix_commit_id` (which you might have found from `jj log`).

First, let's create a hypothetical `bugfix_commit_id` by making a new commit.```bash echo "Fixing a bug." >> file1.txt jj commit -m "fix: temporary bugfix for example"

Let's say the ID of this commit is 777777777777

Now check out back to our original @

jj checkout @-

```
Now, suppose `777777777777` is our `bugfix_commit_id`. We want to find commits
that changed `file1.txt` *and* are ancestors of this bugfix commit.
```bash
# Replace '777777777777' with the actual ID of your 'fix: temporary bugfix for
example' commit
jj log -r 'file("file1.txt") & ancestors(777777777777)'
```

This Revset combines two filters: `file("file1.txt")` to get commits that changed the file, and `ancestors(777777777777)` to limit those commits to ones that are ancestors of the `bugfix_commit_id`. The `&` operator ensures both conditions are met. This query helps narrow down the search for the root cause.

Mini-Challenge: Advanced Commit Discovery

Your turn! Let's put your Revset knowledge to the test.

Challenge: Find all commits authored by "AI Expert" that are descendants of the `feat: initial commit` (use its ID, e.g., `a2b1c3d4e5f6`) and contain the word "file1" in their description, but exclude the current working copy commit (`.`).

Hint: You'll need to combine `author()`, `description()`, `descendants()`, and the difference operator `-`. Remember that `.` includes uncommitted changes.

What to observe/learn: This challenge requires combining multiple Revset clauses to form a precise query, demonstrating the flexibility and power of the language. It also reinforces the distinction between the working copy and other commits, and how to exclude specific commits.
```

💡 **SOLUTION (CLICK TO EXPAND)**

First, make sure you're back in the `my-revset-repo` directory and have your initial commits set up. Find the actual short ID for your `feat: initial commit`. Let's assume it's `a2b1c3d4e5f6` for this example.

```
# Replace 'a2b1c3d4e5f6' with your actual initial commit ID
jj log -r 'author("AI Expert") & description("file1") &
descendants(a2b1c3d4e5f6) - .'
```

This command should output only the "feat: add third line to file1" commit. The "feat: add second line and file2" commit might be excluded if its description doesn't explicitly mention "file1" (it mentions "file2"). If it does mention "file1", it would be included. The key is that it correctly filters by all specified criteria and excludes the working copy.

Common Pitfalls & Troubleshooting

1. **Confusing `.` and `@`**:
    - `📌 Key Idea:` `.` is the **working copy commit** (including uncommitted changes).
    - `📌 Key Idea:` `@` is the **current commit** (its parent, representing the last committed state).
    - For operations that modify history (e.g., `jj rebase`), you often want to target `@` or its ancestors. For operations that inspect the current state (e.g., `jj diff`), `.` is useful if you want to include your ephemeral work.
2. **`A::B` vs. `A..B`**:
    - `A::B` (`inclusive range`): "All commits that are descendants of `A` AND ancestors of `B` (inclusive)." This is often what you want for a linear range of commits.
    - `A..B` (`ancestor-exclusive range`): "The set of commits that are ancestors of `B` but not ancestors of `A`." This selects commits between `A` and `B`, excluding `A` and its ancestors. It's less intuitive for simple linear ranges. When in doubt for inclusive ranges, stick with `A::B`.
3. **Over-complicating Queries**: Start simple. If you need to build a complex Revset, try breaking it down into smaller parts and testing each part with `jj log -r 'part'` before combining them. This modular approach helps debug your Revset.
4. **Forgetting to Quote**: If your Revset contains spaces or special characters (like `|`, `&`, `(`, `)`), always wrap the entire expression in single quotes `'...'` to prevent your shell from interpreting them. Forgetting quotes is a very common source of "command not found" or "invalid argument" errors.
5. **Testing Your Revsets**: Before running a destructive command like `jj rebase` or `jj squash` with a complex Revset, always test it with `jj log -r '<your_revset>'` to ensure it selects exactly the commits you intend. This simple step can save you from unintended history rewrites.

Summary

In this chapter, we've embarked on a journey into Revsets, `jj`'s powerful commit query language. We covered:

- The **necessity of Revsets** for navigating and manipulating `jj`'s mutable history with precision.
- **Basic Revset expressions** like `.` (working copy), `@` (current commit), and `root()` for identifying key points in your history.
- **Navigating history** with operators like `-` (parent) and `::` (inclusive range), and `..` (ancestor-exclusive range) to select sequences of commits.
- **Filtering commits** by properties like `author()`, `description()`, and `file()` to narrow down your search.
- **Combining Revsets** using set operations: `|` (union), `&` (intersection), and `-` (difference) for complex queries.
- **Practical examples** of using Revsets in `jj rebase` and `jj diff` commands, demonstrating their utility in real-world workflows.

Revsets are a fundamental tool in your `jj` arsenal, enabling you to interact with your codebase's history with unmatched precision. As you become more proficient, you'll find yourself crafting sophisticated queries to streamline your development and debugging workflows.

Next, we'll build upon this foundation by exploring `jj`'s core philosophy of **stacked changes** and how Revsets play a crucial role in managing these incremental units of work, leading to cleaner, more reviewable code.

References

- Jujutsu Official Documentation: Revsets
- Jujutsu Official Documentation: Tutorial
- Jujutsu GitHub Repository

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

CHAPTER 06

Embracing Branchless Workflows: Stacked Changes and Bookmarks

In previous chapters, we established a solid foundation by exploring `jj`'s core concepts such as the working-copy-as-a-commit and mutable history. Now, we're ready to delve into one of `jj`'s most distinctive and powerful features: **branchless workflows**.

This chapter will introduce a new way of thinking about managing your development work. Instead of navigating Git's often intricate branching model, you'll discover how `jj` facilitates a simpler, more linear approach using **stacked changes** and lightweight **bookmarks**. This shift can significantly contribute to cleaner history, simplified merges, and more effective code reviews.

To maximize your learning from this chapter, ensure you are comfortable with basic `jj` commands like `jj status`, `jj log`, and `jj commit`, and have a clear understanding of `jj`'s working-copy-as-a-commit model. Let's streamline your version control experience.

The Branchless Philosophy of Jujutsu

Traditional Git workflows often revolve around named branches. Developers create a branch, work on it, merge it back into a main line, and frequently delete it. This can lead to a proliferation of short-lived branches, complex merge graphs, and the cognitive load of constantly managing context switches between branches.

`jj` introduces a different paradigm. Rather than relying on named branches that act as pointers and automatically move as you commit, `jj` places its focus on the **commits themselves** and their explicit parent-child relationships. Your primary method of organizing ongoing work isn't a named branch, but a **stack of changes** – a linear series of dependent commits, where each commit logically builds upon the one preceding it.

📌 Key Idea: `jj`'s branchless approach prioritizes explicit changes (commits) and their direct lineage, rather than relying on named, moving branch pointers for feature development.

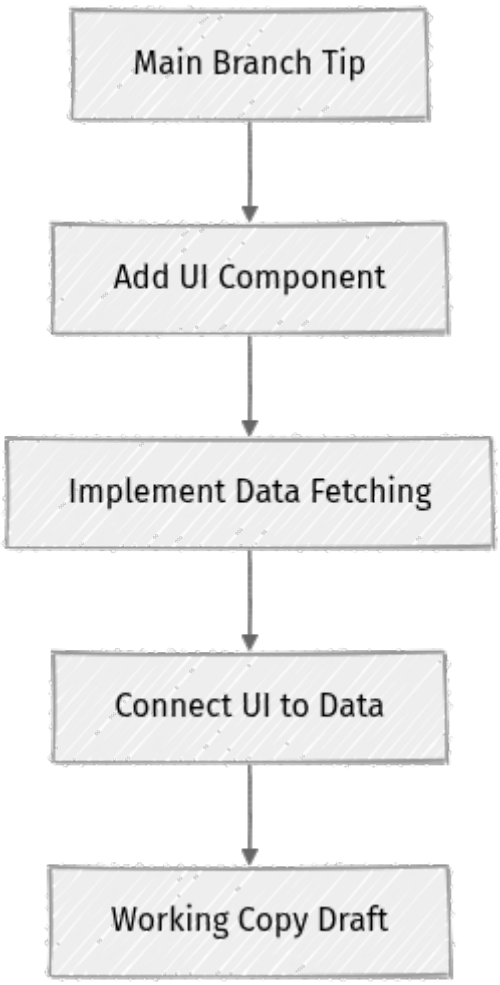
Why Branchless Matters

Adopting a branchless workflow with `jj` offers several compelling advantages:

- **Cleaner History:** Your project history tends to be more linear and easier to trace. The emphasis is on continuously refining and reordering individual commits, rather than generating complex, interwoven merge graphs.
- **Simplified Merges:** With a predominantly linear history, `jj`'s rebase-centric approach makes incorporating upstream changes significantly smoother. Conflicts are often addressed incrementally, commit by commit, making them more manageable.
- **Easier Code Reviews:** Reviewers can inspect a stack of small, focused changes, which allows them to understand the progression of your work step-by-step. This often leads to more targeted and effective feedback.
- **Reduced Mental Overhead:** Less time spent on the mechanics of branch management translates to more time and focus dedicated to actual coding and problem-solving.

Stacked Changes: Building Features Incrementally

The foundation of `jj`'s branchless workflow is the concept of **stacked changes**. Envision developing a new feature not as a single, monolithic commit, but as a series of small, logical, and incremental steps. Each of these steps becomes a distinct, focused commit that directly builds upon the functionality introduced by the previous one.



In this illustration, `Commit 1` serves as the base. `Commit 2` adds functionality that depends on `Commit 1`, and `Commit 3` further builds upon `Commit 2`. Your `Working Copy` represents a draft commit that sits logically on top of `Commit 3`. This structured stack makes your changes transparent and highly reviewable.

Creating Your First Stack

Let's begin by creating a new `jj` repository or navigating to an existing one. We'll simulate the process of developing a new feature.

1. **Initialize a new `jj` repository (if you don't have one):**

```
mkdir my_feature_project
cd my_feature_project
jj init
```

```
This command initializes a new `jj` repository. If you run `jj log`
immediately, you'll see an initial root commit.
```

1. **Create your first change in the stack:** We'll start by adding a new file.

```
    echo "This is the initial setup for the feature." > feature_setup.txt
    jj new
```

```
The `jj new` command creates a new *empty* commit and moves your working copy
to it. This new commit is a child of your current (parent) commit. Any changes
you make in your working directory *now* will be associated with this new
draft commit.

Let's incorporate `feature_setup.txt` into this new commit. Remember, in `jj`,
the working copy *is* a draft commit.
```

```
    jj commit -m "feat: Initial setup for the new feature"
```

```
`jj commit` here takes all modifications from your working directory and adds
them to the *current working-copy commit*, simultaneously assigning it the
provided message. Your working copy remains at this newly updated commit.
```

1. **Add a second, dependent change:** Now, let's build upon that first commit. We'll add another file that logically extends the first step.

```
    echo "This file extends the feature setup." > feature_component.txt
    jj new
```

```
Again, `jj new` creates a *new empty commit* on top of your previous one and
automatically moves your working copy to it. Now, let's incorporate
`feature_component.txt` into this new, current commit.
```

```
    jj commit -m "feat: Add a new component for the feature"
```

```
You now have a stack of two new commits. Let's visualize them.
```

```
    jj log
```

```
You should observe output similar to this (commit IDs will vary):
```

```
    @  xxxxxx feat: Add a new component for the feature
    o  yyyyyy feat: Initial setup for the new feature
    o  zzzzzz Initial commit (root)
```

```
The `@` symbol denotes your current working copy commit. Notice how `jj log`
naturally displays the commits in a stack-like, linear order.
```

Navigating Your Stack with `jj go`

When working with stacked changes, you'll frequently need to move your working copy up and down the stack to inspect or modify earlier commits. The `jj go` command is your primary tool for this.

- `jj go -`: Moves your working copy to the previous commit in the operation log (similar to `cd -` in the shell).
- `jj go @~`: Moves your working copy to the parent of the current working copy commit.
- `jj go <commit_id>`: Moves your working copy to a specific commit identified by its ID.

Let's practice navigating:

1. **Move to the parent commit:**

```
    jj go @~
```

```
Now, if you run `jj log`, you'll see your working copy (`@`) has moved to the
`feat: Initial setup for the new feature` commit.
```

```
    jj log
```

```
You'll observe:
```

```
    o  xxxxxx feat: Add a new component for the feature
    @  yyyyyy feat: Initial setup for the new feature
    o  zzzzzz Initial commit (root)
```

```
Notice that `feature_component.txt` is no longer present in your working
directory. This is because your working copy is now positioned at the `yyyyyy`
commit, which does not yet include the changes from the subsequent commit.
```

1. **Move back to the latest commit:** The latest commit in your stack is the child of your current commit. You can return to it using the operation log.

```
jj go -
```

```
This command will return your working copy to the commit where you were just
previously working.

⚡ Quick Note: `jj go` is an incredibly versatile command for traversing
history. It effectively "checks out" the state of the specified commit by
moving your working copy to it.
```

Bookmarks: `jj`'s Lightweight Branch Alternative

While `jj` strongly advocates for branchless workflows, there are scenarios where a named reference to a specific commit is beneficial, particularly for collaboration or when interacting with Git repositories. This is where **bookmarks** become useful.

Bookmarks in `jj` are functionally similar to local Git tags, but with a crucial difference: they can be pushed and pulled to remotes. This makes them suitable for tracking feature branches in a Git-compatible manner. A key distinction is that `jj` bookmarks **do not move automatically** when you create new commits. They remain fixed pointers to a specific commit unless you explicitly choose to move them.

This behavior contrasts sharply with Git branches, which automatically advance to point to the latest commit every time you commit on that branch.

Working with Bookmarks

Let's create a bookmark for our feature stack to see how they work.

1. **Ensure your working copy is at the top of your feature stack:**

```
jj log -r @
```

```
Verify that your working copy (`@`) is at the `feat: Add a new component for
the feature` commit. If it's not, use `jj go <commit_id>` to navigate there.
```

1. **Create a bookmark:**

```
    jj bookmark my_awesome_feature
```

```
This command creates a bookmark named `my_awesome_feature` that points to your
current working-copy commit.
```

1. **List your bookmarks:**

```
    jj bookmark list
```

```
You should see your newly created bookmark (commit ID will vary):
```

```
    my_awesome_feature: xxxxxx feat: Add a new component for the feature
```

1. **Observe bookmark immutability:** Let's add another commit and observe how the bookmark behaves.

```
    echo "Final touches for the feature." > feature_final.txt
    jj new
    jj commit -m "feat: Add final touches"
    jj log
    jj bookmark list
```

```
You'll notice that your `my_awesome_feature` bookmark still points to the
*previous* commit (`xxxxxx`), not the new one (`yyyyyy`) where your working
copy is now located.
```

```
    @  yyyyyy feat: Add final touches
    o  xxxxxx feat: Add a new component for the feature
    | my_awesome_feature
    o  zzzzzz feat: Initial setup for the new feature
    o  aaaaaa Initial commit (root)
```

```
To move a bookmark, you would explicitly point it to a new commit using `jj
bookmark my_awesome_feature -r @` (to move it to the current working copy).

⚡ Real-world insight: Bookmarks are excellent for marking "save points" or
referencing specific versions of a feature that you might want to share or
discuss with others, without the dynamic movement overhead of Git's branches.
```

1. **Delete a bookmark:**

```
    jj bookmark delete my_awesome_feature
    jj bookmark list
```

```
The bookmark has now been removed.
```

Refining Your Stack: Modifying and Reordering Changes

One of `jj`'s most powerful capabilities is its ease in modifying and reordering commits within your stack. This is where `jj`'s mutable history truly shines, enabling you to craft a clean, logical history before you share your work.

`jj squash`: Combining Commits

During development, it's common to make several small, incremental commits that, in retrospect, would be better combined into a single, more comprehensive commit. The `jj squash` command is perfectly suited for this.

Let's recreate our two initial feature commits to demonstrate:

```
# Ensure you're at a clean base commit, e.g., the root or main branch
jj go root() # Moves to the very first commit in the repository

jj new
echo "First part of the feature." > part1.txt
jj commit -m "feat: Add first part"

jj new
echo "Second part of the feature." > part2.txt
jj commit -m "feat: Add second part"

jj log
```

You should now have a linear history with two new commits: `feat: Add second part` on top of `feat: Add first part`.

Now, let's squash them into one single commit:

```
jj squash @~
```

This command squashes the parent of the current commit (`@~`) into the current commit (`@`). `jj` will then open your default editor, allowing you to combine and refine the commit messages.

After saving the combined message, run `jj log` again. You'll observe that the two original commits have been merged into a single one, and your working copy is now positioned at this new, consolidated commit.

```
@  xxxxxx feat: Add first part and second part (combined message)
o  yyyyyy Initial commit (root)
```

⚠️ What can go wrong: If you accidentally squash into the wrong commit or make an unintended change, remember that `jj undo` is your immediate safety net!

`jj amend -i`: Interactively Modifying an Existing Commit

Sometimes you realize you forgot to include a file, made a typo in a commit message, or want to split an existing commit. `jj amend -i` (interactive amend) allows you to modify the current working-copy commit or even re-stage changes.

Let's say you just committed, but realize you forgot a crucial file:

```
# Assuming your working copy is at a commit you just made
echo "Forgot this important file!" > important.txt
jj amend -i
```

This command will open an interactive prompt, allowing you to choose which changes from your working directory to include in the current commit. You can also edit the commit message.

`jj rebase`: Reordering and Moving Commits

`jj rebase` is an exceptionally flexible command. It can move commits, reorder them, or even insert new commits into an existing history. This capability is fundamental for maintaining a clean, logical feature stack and for keeping it synchronized with upstream changes.

Let's set up a scenario where we have two independent lines of development that we want to merge linearly.

First, ensure you're at a clean base commit (e.g., the root of the repository):

```
jj go root() # Moves to the very first commit in the repository
```

Now, let's create two parallel feature lines, both stemming from this root commit:

```
# Create the first feature line (Feature A)
jj new # Creates a new commit (let's call it A1) on top of root
echo "Feature A: Initial component." > feature_a.py
jj commit -m "feat: Add Feature A's core component"

# Create the second feature line (Feature B), also on top of root
# We need to explicitly tell `jj new` to parent this commit to root()
jj new -r root() # Creates a new commit (let's call it B1) on top of root
echo "Feature B: Configuration setup." > feature_b.cfg
jj commit -m "feat: Add Feature B's configuration"

jj log
```

Your `jj log` output should now show two commits (`A1` and `B1`) both having `root()` as their parent. Your working copy (`@`) will be at `B1`.

```
@  xxxxxx feat: Add Feature B's configuration
o  yyyyyy feat: Add Feature A's core component
o  zzzzzz Initial commit (root)
```

Notice `xxxxxx` and `yyyyyy` are siblings, both children of `zzzzzz`.

Now, let's rebase `feat: Add Feature B's configuration` (the commit at `xxxxxx`) to sit on top of `feat: Add Feature A's core component` (the commit at `yyyyyy`). This will make our history linear. We can refer to the "Feature A" commit using the `root()+'` revset, which means "the first child of the root commit".

```
jj rebase -s @ -d 'root()+'
```

After running this, `jj log` will show the `feat: Add Feature B's configuration` commit now sits on top of `feat: Add Feature A's core component`, making your stack beautifully linear.

```
@  xxxxxx feat: Add Feature B's configuration
o  yyyyyy feat: Add Feature A's core component
o  zzzzzz Initial commit (root)
```

🔥 Optimization / Pro tip: `jj rebase` is your primary tool for keeping your feature stack up-to-date with the `main` or `master` branch. Regularly rebase your feature stack onto the latest upstream changes to avoid large, complex merges later.

Mini-Challenge: Developing a Feature with Stacked Changes and Bookmarks

Now it's your turn to apply what you've learned. You'll simulate developing a new feature, using stacked commits and a bookmark.

Challenge: Implement a simple "User Profile" feature.

1. **Start a new `jj` project or ensure your current one is clean.**
2. **Create an initial commit for the feature:** Add a file `profile_data_model.py` with some placeholder content.
    - Commit message: `feat: Define user profile data model`
3. **Add a second, dependent commit:** Add a file `profile_api.py` that would theoretically use the data model.
    - Commit message: `feat: Implement basic profile API endpoint`
4. **Create a bookmark** named `user_profile_feature` that points to the latest commit in your stack.
5. **Add a third, final commit to the stack:** Add a file `profile_ui.js` with some placeholder UI code.
    - Commit message: `feat: Develop profile UI component`
6. **Squash the first two commits** (`profile_data_model.py` and `profile_api.py`) into a single, more comprehensive "Profile Core" commit. Update the commit message accordingly.
7. **Verify your history** with `jj log` and `jj bookmark list`. Your `user_profile_feature` bookmark should still point to the original `profile_api.py` commit (before it was squashed), and your history should reflect the new, squashed commit.

Hint:

- Remember `jj new` to create new commits in your stack.
- Use `jj commit -m "message"` to finalize changes in the working copy's draft commit.
- `jj bookmark <name>` creates a static bookmark.
- `jj squash @~` will squash the parent into the current commit.
- Use `jj log` frequently to inspect your history.

What to observe/learn:

- How `jj new` facilitates building a linear, incremental stack.
- The effect of `jj squash` on simplifying your history.
- The non-moving nature of `jj` bookmarks in contrast to Git branches.

Common Pitfalls & Troubleshooting

- **Misunderstanding Bookmark Behavior:** The most frequent pitfall for users migrating from Git is expecting `jj` bookmarks to behave like Git branches (i.e., automatically moving forward with `jj commit`). It's crucial to remember that `jj` bookmarks are static pointers. If you intend for a bookmark to track your active development, you must explicitly move it using `jj bookmark <name> -r @`.
- **Forgetting `jj go`:** When managing stacked changes, you will often need to move your working copy to an earlier commit to make a correction, perform an inspection, or branch off from a specific point. Forgetting to use `jj go` and attempting to modify an earlier commit while your working copy is at the top of the stack can lead to confusion. Always check `jj log` to confirm the position of `@` (your working copy).
- **Initial Difficulty with `revsets` for Commit Selection:** While using commit IDs is perfectly fine for simple operations, selecting specific commits for commands like `jj rebase` or `jj squash` becomes much more powerful with `revsets`. Initially, stick to IDs, but as you gain experience, learning `revsets` like `@~` (parent), `@+` (child), `root()..` (all commits from root) will prove invaluable for complex history manipulation.
- **Not Utilizing `jj undo`:** Experimenting with commands like `jj squash` or `jj rebase` can sometimes lead to unintended outcomes. There's no need to panic! `jj undo` is your robust safety net. It can revert almost any `jj` operation, allowing you to easily correct mistakes and try again without losing work.

Summary

You've made significant progress in understanding `jj`'s unique approach to version control. In this chapter, we thoroughly explored:

- **Branchless Workflows:** The philosophy behind how `jj` simplifies development by emphasizing a linear history of changes over complex branching models.
- **Stacked Changes:** The powerful technique of building features as a series of small, dependent commits, which enhances manageability for both development and code review.
- **`jj new` and `jj commit`:** The fundamental commands for creating new commits and incorporating changes into your working copy's draft commit.
- **`jj go`:** Your essential tool for navigating up and down your commit stack, effectively checking out historical states.
- **Bookmarks:** `jj`'s lightweight, static alternative to Git branches, ideal for marking specific commits for reference, sharing, or integration with Git remotes.
- **Refining History:** How to use `jj squash` to combine related commits and `jj rebase` to reorder commits, enabling you to craft a clean, logical, and presentable history.
- **`jj amend -i`:** The command for interactively modifying an existing working-copy commit.

By internalizing and applying these concepts, you are moving towards a more efficient and less cumbersome version control experience. In the next chapter, we will delve deeper into `revsets`, `jj`'s powerful language for selecting and filtering commits, which will unlock even more advanced workflows and make navigating complex histories remarkably simple.

References

- Jujutsu GitHub Repository: https://github.com/jj-vcs/jj
- Jujutsu Tutorial (main branch): https://github.com/jj-vcs/jj/blob/main/docs/tutorial.md
- Jujutsu Bookmarks Documentation: https://github.com/jj-vcs/jj/blob/main/docs/bookmarks.md

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

CHAPTER 07

Jujutsu and Git: Seamless Interoperability and Collaboration

Jujutsu and Git: Seamless Interoperability and Collaboration

Welcome back, fellow version control enthusiast! In the previous chapters, we've explored the foundational concepts of Jujutsu (`jj`), from its unique working-copy-as-a-commit model to the power of mutable history and the operation log. You're now comfortable with `jj`'s core philosophy and its local development superpowers.

However, the reality of modern software development is that Git remains the dominant version control system. How do we reconcile `jj`'s innovative approach with the pervasive need to collaborate within a Git-centric ecosystem? This chapter is your bridge, showing you how `jj` and Git don't just coexist, but work together beautifully.

Why This Matters

The ability to seamlessly interact with Git repositories is not just a convenience; it's a necessity for `jj` users. Whether you're contributing to a large open-source project, working on a team that exclusively uses Git, or migrating an existing Git codebase, `jj`'s interoperability features allow you to leverage its power without forcing your entire team to switch. You get the best of both worlds: `jj`'s superior local workflow and Git's ubiquitous remote collaboration.

By the end of this chapter, you'll be able to:

- Initialize `jj` repositories directly from existing Git projects.
- Fetch and pull changes from Git remotes into your `jj` workspace.
- Push your `jj` history, including carefully crafted stacked changes, to Git remotes.
- Understand the nuances of `jj`'s branch concepts when interacting with Git.
- Effectively resolve conflicts that arise during Git synchronization, making merges less painful.

Let's dive into making `jj` and Git the best of friends, enabling you to contribute to any Git project with `jj`'s power!

Jujutsu's Git Integration: A Symbiotic Relationship

At its core, `jj` isn't trying to replace Git entirely; it's designed to be a more powerful, user-friendly **frontend** to Git. When you use `jj` with a Git repository, `jj` actually maintains a full Git repository internally, alongside its own object store. This means `jj` understands Git objects, references, and protocols natively, acting as a sophisticated translator.

📌 **Key Idea:** Think of `jj` as a "smart wrapper" or an "enhanced interface" around a standard Git repository. It translates your `jj` commands and concepts into native Git operations when you interact with remotes, providing a superior local experience without disrupting external Git workflows.

The Mental Model: `jj` as Your Local Powerhouse

When you're working locally, `jj` gives you superpowers: mutable history, easy stacking of changes, and the invaluable operation log. These features streamline your development process. When it's time to share your work or incorporate others' changes, `jj` gracefully handles the translation to and from Git, ensuring compatibility.

This diagram illustrates how `jj` acts as your primary interface, managing its own internal representation while seamlessly synchronizing with external Git repositories. This setup means `jj` isn't just a separate tool; it's deeply integrated with Git's underlying mechanics.

Step-by-Step: Interacting with Git Repositories

Now let's get hands-on with the essential commands for `jj`-Git interoperability.

1. Initializing a Jujutsu Repository from Git

The most common way to start using `jj` with an existing Git project is to clone it. `jj` provides a dedicated command that handles this transparently.

Action: Clone a Git Repository

Instead of `git clone`, you'll use `jj clone`. Let's set up a new directory for our work.

First, navigate to your desired parent directory. For example:

```
cd ~/projects
```

Now, let's clone a hypothetical Git repository. For this exercise, you can use a public example like `<https://github.com/jj-vcs/jj.git>` for demonstration, or replace it with any Git repository URL you have access to.

```
jj clone https://github.com/jj-vcs/jj.git my-jj-git-project
```

After cloning, `jj` will provide feedback on the new repository and its current state.

```
Cloned Git repository into "my-jj-git-project".
Working copy now at node 61a1a7b3c2e3 (empty commit)
```

Notice the output: `jj` has cloned the Git repository and immediately created a `jj` working copy. The `jj` working copy always points to a commit, even if it's an "empty commit" in terms of changes you've made directly. This is `jj`'s "working-copy-as-a-commit" model in action.

Action: Explore the Cloned Repository

Navigate into your new `jj` project directory:

```
cd my-jj-git-project
```

Now, let's inspect the history using `jj log`:

```
jj log
```

You'll see a history that mirrors the Git repository's history, but presented in `jj`'s intuitive format. You might also see commits labeled with `origin/main` (or `origin/master`), indicating the remote's default branch. This shows `jj` is aware of the remote state.

⚡ **Quick Note:** `jj` automatically sets up the remote `origin` for you, just like Git would. You can verify this by listing the Git remotes `jj` is aware of:

```
jj git remote list
```

```
origin (https://github.com/jj-vcs/jj.git)
```

This command confirms the Git remotes configured for `jj`'s internal Git repository.

2. Synchronizing Changes: Fetching and Pulling

Keeping your local `jj` repository in sync with the remote Git repository is crucial for collaboration. `jj` provides commands that mirror `git fetch` and `git pull`, but with `jj`'s mutable history advantages.

Action: Fetch Remote Changes with `jj git fetch`

`jj git fetch` does exactly what `git fetch` does: it downloads objects and references from the remote Git repository into your local `jj`'s internal Git repository. Crucially, it does not modify your local working copy or `jj`'s commit history directly. It only updates the remote-tracking branches (e.g., `origin/main`).

Let's simulate fetching from a remote. If your cloned repository has new commits pushed to `origin/main` by other developers, `jj git fetch` will retrieve them.

```
jj git fetch origin
```

```
Fetching into "my-jj-git-project"...
[2026-05-19 10:00:00] Fetched from origin.
```

After fetching, you can see the new remote changes using `jj log -r 'remote_branches()'` or by comparing your local `main` with `origin/main`.

```
jj log -r 'main..origin/main'
```

This command uses a `revset` (which we'll explore more deeply in a later chapter) to show commits that are on `origin/main` but are not yet part of your local `main` branch's history.

Action: Pulling Changes with `jj git pull`

`jj git pull` is the equivalent of `git pull --rebase`. It first fetches changes from the remote and then automatically rebases your local changes on top of the newly fetched remote commits. This is `jj`'s default and recommended way to integrate changes, promoting a clean, linear history.

Let's assume there are new commits on `origin/main` that you want to integrate into your local `jj` repository.

```
jj git pull origin main
```

```
Fetching into "my-jj-git-project"...
Rebasing 1 commits onto 87654321fedc...
Working copy now at node 987654321abc
```

🧠 **Important:** `jj git pull` automatically performs a rebase, aligning perfectly with `jj`'s philosophy of linear history and mutable changes. If you had local changes (commits not yet pushed to `origin/main`), `jj` would intelligently rebase them on top of the newly fetched remote commits. This is where `jj` shines, as its rebase operations are fast, reliable, and often conflict-free, even with complex stacks of changes.

Hands-on Exercise: Simulating a Remote Update and Pull

Let's create a scenario where a "remote" changes, and you pull those changes into your `jj` repository. We'll use a temporary Git repository to simulate this.

1. **Create a dummy Git remote (outside your `jj` project):**

```
mkdir -p ../remote_repo_dummy
cd ../remote_repo_dummy
git init --bare
cd ../my-jj-git-project
```

1. **Add the dummy remote to your `jj` repo:**

```
jj git remote add dummy ../remote_repo_dummy
```

1. **Simulate another developer making a commit directly to the dummy remote:**

```
cd ../remote_repo_dummy
git clone . ../temp_clone_for_remote_commit
cd ../temp_clone_for_remote_commit
echo "Initial content for dummy remote" > remote_file.txt
git add remote_file.txt
git commit -m "feat: Initial commit on dummy remote main branch"
git push origin main
```

```
cd ../my-jj-git-project
```

1. **Now, pull from the dummy remote in your `jj` repo:**

```
jj git pull dummy main
```

```
You should see `jj` fetching and then potentially rebasing. Even if you had no
local changes, `jj` updates its `dummy/main` reference and potentially moves
your working copy to track the latest remote commit.
```

```
Fetching into "my-jj-git-project"...
Rebasing 0 commits onto <new_commit_id>...
Working copy now at node <new_commit_id>
```

```
`jj` has successfully pulled the changes from your simulated remote! You can
verify with `jj log`.
```

3. Contributing to Git: Pushing Your Changes

When you've made changes and commits in `jj` and are ready to share them with a Git remote, you use `jj git push`. This command pushes your local `jj` commits to a specified Git branch on a remote.

Action: Make Local Changes in `jj`

Let's create a couple of stacked commits in `jj`.

```
echo "feature A line 1" >> feature-a.txt
jj commit -m "feat: Add feature A initial UI"
```

```
echo "feature A line 2" >> feature-a.txt
jj commit -m "feat: Implement feature A backend logic"
```

Now, your `jj log` will show two new commits on top of your `main` branch.

```
jj log -r 'main..'
```

This `revset` shows commits reachable from your working copy that are not reachable from `main`.

Action: Prepare for Push: Clean History

Git's model generally prefers a linear, non-rewritten history for shared branches. While `jj` thrives on mutable history locally, it's a best practice to clean up your local `jj` history (e.g., by squashing small commits or rebasing for a cleaner narrative) before pushing to a shared Git remote. This makes your contributions easier for others to review and merge.

Let's squash our two feature commits into one, presenting a single, cohesive change to the remote.

```
jj squash @-
```

This command squashes the working copy's parent commit (`@-`) into the working copy itself, effectively combining the two feature commits. You'll be prompted to edit the commit message. Combine them into a single, concise message like "feat: Implement complete user profile editing feature".

```
Rebased 1 commits onto <parent_of_squashed_commit_id>...
Working copy now at node <new_squashed_commit_id>
```

Now, `jj log -r 'main..'` should show just one clean, well-described commit ready for public consumption.

Action: Push with `jj git push`

Now that your history is clean and linear, you can push it to the remote.

```
jj git push origin main
```

```
Pushing to origin...
```

If successful, your local `main` branch will now be synchronized with `origin/main`.

⚠️ **What can go wrong:** If you try to push rewritten history (e.g., you `jj rebase` or `jj amend` a commit that's already been pushed to Git and shared), `jj git push` will fail, just like `git push` would without `--force`. `jj` is smart enough to detect this and prevent accidental force pushes to shared branches. To force a push, you would use `jj git push --force origin main`, but this should be done with extreme caution and only when you understand the implications for other collaborators (as it rewrites shared history).

Hands-on Exercise: Push a Refactored Commit

1. **Make a new commit:**

```
echo "temporary log addition" >> logs.txt
jj commit -m "chore: Add temporary logging"
```

1. **Amend the previous commit (simulate a quick fix or refinement):**

```
echo "another log line" >> logs.txt
jj amend @-
```

```
Edit the commit message to reflect the combined change (e.g., "chore: Refine
logging setup").
```

1. **Push to your dummy remote (created earlier):**

```
jj git push dummy main
```

```
This should succeed because the `amend` only changed a commit that hadn't been
pushed to the `dummy` remote yet. If you had tried to amend a commit *already*
on `dummy/main` and then pushed, `jj` would prevent it without `--force`.
```

4. Branch Management: `jj bookmarks` vs. Git Branches

This is a critical area where `jj`'s philosophy diverges from Git's, yet `jj` provides tools to bridge the gap effectively.

Git Branches in `jj`

When you clone a Git repository, `jj` automatically tracks Git branches (like `main`, `develop`, `feature/x`) as special `jj` branches. These `jj` branches behave like Git branches: they point to a specific commit, and when you push or pull, `jj` updates them accordingly, making them suitable for remote synchronization.

You can list these `jj` branches using `jj branch list`:

```
jj branch list
```

```
  main: 61a1a7b3c2e3 (feat: Implement complete user profile editing feature)
@ 4b3d8c1e2f3a (chore: Refine logging setup)
```

The `main` entry here represents the `main` branch that `jj` manages and synchronizes with `origin/main`.

`jj bookmarks`: Your Local, Branchless Superpower

`jj` introduces `bookmarks` as a more flexible, local alternative to traditional Git branches. Bookmarks are simply named pointers to commits, but unlike `jj` branches (which are designed for Git interoperability), bookmarks are purely local to your `jj` repository and are not pushed to Git remotes.

🔥 **Pro tip:** For most of your local development in `jj`, especially when working with stacked changes, you'll find yourself relying on `jj`'s "branchless" workflow. This means you just keep committing on top of your working copy, and `jj` handles the commit graph. Bookmarks become useful when you want to easily jump back to a specific point, mark a significant commit for later reference, or share a commit reference with another `jj` user locally.

You can create a bookmark like this:

```
jj bookmark my-experimental-feature
```

Now, `jj log` will show your bookmark alongside your commits:

```
...
@  4b3d8c1e2f3a (chore: Refine logging setup)
   my-experimental-feature
...
```

You can switch your working copy to a bookmark using `jj edit my-experimental-feature`.

When to use `jj branch` vs. `jj bookmark`:

- **`jj branch`**: Use when you need to interact with a Git remote. These branches are synchronized with Git and are suitable for shared, long-lived branches.
- **`jj bookmark`**: Use for purely local navigation, temporary pointers, or when you don't want to expose a branch to Git. They are perfect for `jj`'s branchless workflow, allowing you to easily manage multiple lines of work without the overhead of Git branches.

5. Resolving Conflicts with `jj` and Git

Conflicts are an inevitable part of collaborative development, especially when integrating changes from Git. `jj` provides a streamlined workflow for resolving them, making the process less daunting.

Action: Trigger a Conflict

A conflict typically occurs when you `jj git pull` (which rebases your changes) and both you and the remote have modified the same lines in a file.

Let's simulate a conflict:

1. **Introduce a local change in your `jj` repo:**

```
    echo "my important local change" >> conflict_file.txt
    jj commit -m "feat: Add local conflict line"
```

1. **Simulate a remote change on the same line (using our dummy remote):**

```
    cd ../remote_repo_dummy
    git clone . ../temp_clone_2
    cd ../temp_clone_2
    echo "remote important change" >> conflict_file.txt
    git add conflict_file.txt
    git commit -m "feat: Add remote conflict line"
    git push origin main
    cd ../my-jj-git-project
```

1. **Now, pull the remote changes into your `jj` repo:**

```
    jj git pull dummy main
```

```
`jj` will detect the conflict during the rebase operation and pause,
indicating a conflicted state.
```

```
    Rebasing 1 commits onto <remote_commit_id>...
    Conflict: 'conflict_file.txt' was modified in parallel.
    Working copy now at node <conflict_commit_id> (conflict)
```

```
Your working copy is now in a conflicted state, waiting for your intervention.
```

Action: Identify and Resolve Conflicts

`jj` provides commands to help you see and resolve conflicts.

- **View conflicting files:**

```
jj diff --conflicts
```

```
This command shows you the conflicted file(s) with standard Git conflict
markers (`<<<<<<<`, `=======`, `>>>>>>>`).
```

- **Manually edit the file:** Open `conflict_file.txt` in your favorite text editor. You'll see content similar to this:

```
# Before editing conflict_file.txt
<<<<<<<
my important local change
=======
remote important change
>>>>>>>
```

```
Resolve the conflict by choosing which changes to keep, or by merging them.
Then, remove the conflict markers. For example, if you decide to keep the
remote change:
```

```
# After editing conflict_file.txt (e.g., choosing remote)
remote important change
```

```
Or if you want to keep both:
```

```
# Example of keeping both changes
my important local change
remote important change
```

Action: Mark Conflicts as Resolved

Once you've manually edited the file and removed all conflict markers, you need to tell `jj` that the conflict is resolved.

```
jj resolve conflict_file.txt
```

You can resolve multiple files at once by listing them or using `jj resolve --all`. After resolving all conflicts, `jj` will automatically complete the paused rebase operation.

```
Resolved conflict in 'conflict_file.txt'.
Rebased 1 commits onto <remote_commit_id>...
Working copy now at node <resolved_commit_id>
```

Your working copy is now clean, and the rebase is complete.

Hands-on Exercise: Resolve a Conflict

Follow the steps above to create and resolve a conflict. Pay close attention to the output of `jj diff --conflicts` and how `jj resolve` completes the rebase. Experiment with different resolution strategies (keeping local, keeping remote, or merging both) to get comfortable with the editing process. This practice will build your confidence for real-world scenarios.

Mini-Challenge: Feature Development and Git Integration

Let's put your `jj` and Git interoperability skills to the test with a more comprehensive scenario.

Challenge: You are tasked with developing a new feature, `user-profile-editing`. While you're working, a critical bug fix is pushed to the remote. You should:

1. Start from your `main` branch.
2. Create two distinct `jj` commits for your feature:
    - One commit for adding the basic UI elements for profile editing.
    - Another commit for implementing the backend logic for saving profile data.
3. Simulate a conflict: have the "remote" modify a file (e.g., `shared_config.txt`) that your first UI commit also touched.
4. Pull the remote changes into your `jj` repository, and resolve any conflicts that arise.
5. After resolving, squash your two feature commits into a single, cohesive commit with a clear, descriptive message (e.g., "feat: Implement full user profile editing").

6. Finally, push your single, clean feature commit to the `dummy` remote's `main` branch.

Hint: Remember to use `jj squash` to clean up your history before pushing to present a polished commit to the remote. For the conflict simulation, make sure the "remote" change to `shared_config.txt` overlaps with your `jj` UI commit's change.

What to observe/learn: This challenge will reinforce the entire workflow: local `jj` development, handling unexpected remote updates, conflict resolution, history cleaning, and finally, pushing your polished work to Git. You'll appreciate how `jj` simplifies the rebase and squash operations that are often cumbersome in raw Git, making your development process smoother and more efficient.

Common Pitfalls & Troubleshooting

Even with `jj`'s excellent design, integrating with Git can have its nuances. Being aware of these common pitfalls can save you time and frustration.

1. Pushing Rewritten History to a Shared Git Branch

- **Pitfall:** Accidentally trying to push a rewritten commit (one that's already on the remote but you've `amended`, `rebased`, or `squashed` locally) without explicitly using `--force`. `jj` will correctly block this to protect shared history.
- **Troubleshooting:** Understand that `jj`'s mutable history is incredibly powerful locally. When interacting with shared Git remotes, aim for linear, non-rewritten history on shared branches (like `main` or `develop`). If you must force push (e.g., you're the sole developer on a private feature branch, or coordinating with your team), use `jj git push --force origin my-feature-branch`. Always be extremely cautious with force pushes on shared branches, as they can cause significant headaches for other collaborators.

2. Misunderstanding `jj`'s Working Copy as a Commit Model

- **Pitfall:** Expecting `jj` to behave exactly like Git, where the working directory is separate from a "staging area" and the "last commit." This mental model can lead to confusion about `jj commit` and `jj amend`.

- **Troubleshooting:** Remember, in `jj`, your working copy is a commit (the "working-copy commit"). When you `jj commit`, you're essentially creating a new commit on top of your current working copy's state, and your working copy then points to this new commit. Changes in your working directory are uncommitted changes that will become part of the next commit (or amend the current one). This model simplifies many operations (like `amend` or `squash`) but requires a slight mental shift initially.

3. Issues with Git Hooks and `jj`

- **Pitfall:** Git hooks (like `pre-commit`, `pre-push`) are often configured in the `.git/hooks` directory. When `jj` interacts with its internal Git repository, these hooks might not always fire as expected or might require specific configuration.
- **Troubleshooting:** `jj` has its own hook mechanism for `jj`-specific operations. For `jj` specific hooks, consult the official `jj` documentation on `jj-hooks` for the most current setup (as of 2026-05-19). For Git hooks that need to run when `jj git push` or `jj git pull` is executed, `jj` generally aims to run them where appropriate. If you encounter issues, verify the hook configuration within the internal `.git` directory `jj` uses (usually located under `.jj/repo/store/git` within your `jj` repository, though `jj` handles this mostly transparently). Ensure your Git hooks are compatible with how `jj` orchestrates Git operations.

Summary

Congratulations! You've successfully navigated the exciting world of `jj` and Git interoperability. You now possess the knowledge to integrate `jj` into virtually any Git-based workflow, leveraging `jj`'s local power while seamlessly collaborating with the wider Git ecosystem.

Here are the key takeaways from this chapter:

- **`jj` as a Git Frontend:** `jj` acts as a powerful, user-friendly frontend to Git, managing an internal Git repository and translating `jj` commands into Git operations.
- **Cloning Git Repos:** Use `jj clone` to easily start `jj` projects from existing Git repositories.
- **Synchronization:** `jj git fetch` retrieves remote changes, and `jj git pull` (which automatically rebases) integrates them into your local history.

- **Contributing:** Use `jj git push` to send your local `jj` commits to Git remotes. Remember to clean up your history with `jj squash` or `jj rebase` for a linear, easy-to-review Git history.
- **Branching Models:** Understand the distinction: `jj` branches are for Git interoperability, while `jj` bookmarks offer flexible, local, branchless workflow management.
- **Conflict Resolution:** `jj` provides a streamlined workflow with `jj diff --conflicts` and `jj resolve` to handle merge conflicts efficiently.

You're now equipped to enjoy `jj`'s superior local development experience while seamlessly collaborating with the wider Git world. This blend of local efficiency and global compatibility is what makes `jj` such a compelling tool for modern developers.

What's next? In the upcoming chapters, we'll dive deeper into advanced topics like `revsets` for powerful history manipulation, customizing `jj` for peak productivity, and adopting truly branchless workflows for large projects. Get ready to master `jj`!

References

1. Jujutsu Official GitHub Repository
2. Jujutsu User Manual: Git Integration (Checked 2026-05-19)
3. Jujutsu User Manual: Bookmarks (Checked 2026-05-19)
4. Jujutsu User Manual: Conflicts (Checked 2026-05-19)
5. Git Official Documentation

CHAPTER 08

Advanced Revsets: Mastering Complex Revision Selection

Introduction

Welcome back, `jj` adventurer! In previous chapters, you've learned the basics of `jj` and started to appreciate its mutable history. Now, it's time to unlock one of `jj`'s most powerful features: **Revsets**. Think of Revsets as `jj`'s query language for your commit history. Just as SQL allows you to precisely select data from a database, Revsets empower you to select exactly the commits you need from your repository's graph.

This chapter will transform you from a casual `jj` user to a history-querying wizard. We'll delve into the advanced syntax, operators, and predicates that make Revsets indispensable for complex refactoring, targeted rebases, and seamless Git interoperability. By the end, you'll be able to craft intricate queries to find, filter, and manipulate your commits with surgical precision.

The Language of History: Why Revsets Matter

Imagine your commit history not as a simple linear list, but as a rich, interconnected graph of changes. Traditional VCS tools often rely on simple branch names or commit hashes, which can become cumbersome as your history grows or when dealing with complex relationships like merges or stacked changes.

Revsets solve this by providing a flexible, powerful syntax to describe sets of commits based on their relationships, attributes, and content. This ability to precisely target commits is crucial for `jj`'s mutable history model, allowing you to rebase, squash, or amend specific changes without affecting others.

📌 **Key Idea:** Revsets are `jj`'s built-in query language for your commit graph, enabling precise selection and manipulation of commits.

Core Concepts: Building Blocks of Revsets

At its heart, a Revset is an expression that evaluates to a set of commits. These expressions can be simple, like `@` for the working copy, or incredibly complex, combining multiple operators and predicates.

Basic Revset Syntax (A Quick Recap)

You've likely encountered some basic Revsets already. These foundational elements help you pinpoint key points in your history:

- `@`: The working copy commit. This represents the commit your workspace is currently based on.
- `HEAD`: The commit pointed to by the `HEAD` reference (usually the tip of your current branch). In `jj`, this is often equivalent to `@` if you're working on a branch.
- `main`, `master`, `my-feature`: A specific branch name.
- `a1b2c3d4`: A full or partial commit hash.
- `root()`: The initial commit(s) in the repository with no parents.

Navigating the Graph: Parents and Children

`jj` provides intuitive ways to move through the commit graph relative to a given commit. This is essential for understanding the lineage of changes.

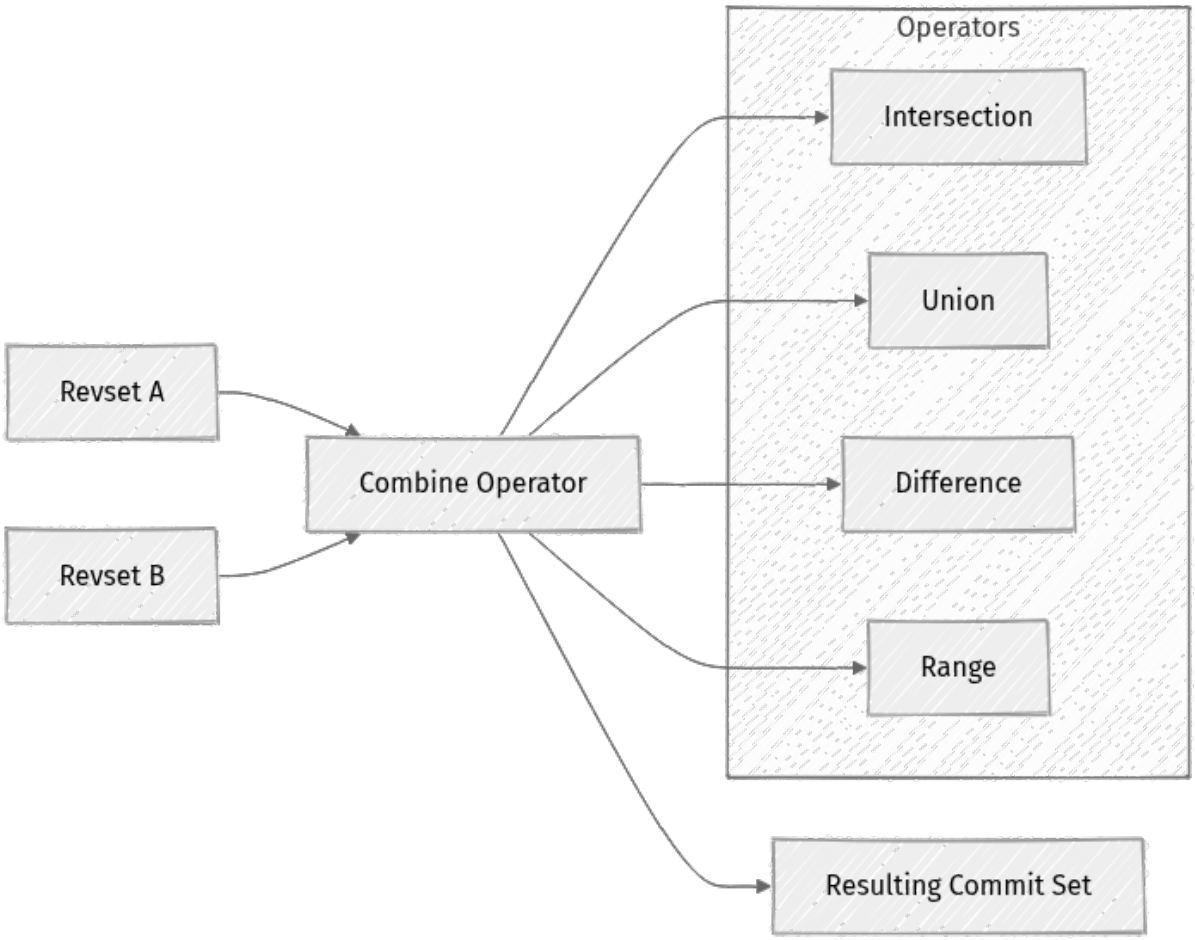
- `commit_id-`: The direct parent(s) of `commit_id`. If a commit has multiple parents (a merge commit), this selects all of them.
- `commit_id--`: The grandparent(s) of `commit_id`. You can chain this multiple times.
- `commit_id+`: The direct child(ren) of `commit_id`.
- `commit_id++`: The grandchild(ren) of `commit_id`.

Operators for Combining Commit Sets

Revsets truly shine when you start combining commit sets using logical operators, similar to set theory. These operators allow you to define complex relationships between different groups of commits.

Operator	Description	Example
`&`	**Intersection**: Commits present in BOTH sets.	`@ & main` (commit that is both WC and main)
`` ` ``	`` ` ``	**Union**: Commits present in EITHER set.
`-`	**Difference**: Commits in the first set, NOT in the second.	`all() - public()` (local-only commits)
`..`	**Range (exclusive)**: Commits reachable from A but not B, excluding A.	`main..@` (commits on current branch since `main`, excluding `main`)
`..=`	**Range (inclusive)**: Commits reachable from A but not B, including A.	`main..=@` (commits on current branch since `main`, including `main`)
`::`	**Ancestors/Descendants**: All ancestors up to a commit, or all descendants from a commit.	`main::@` (all commits from `main` to `@`, including both)
`~N`	**Nth Ancestor**: The Nth parent. `~1` is the same as `-`.	`@~3` (the great-grandparent of the working copy)
`^N`	**Nth Parent**: For merge commits, `^1` is the first parent, `^2` is the second.	`a1b2c3^2` (the second parent of merge commit `a1b2c3`)

Let's visualize how some of these operators might combine commit sets:



Predicates for Filtering by Attributes

Beyond relationships, you can filter commits based on their metadata or content using **predicates**. These allow you to select commits that match specific criteria, such as author, message, or files changed.

Predicate	Description	Example
`author(pattern)`	Commits by a specific author (regex or glob).	`author("Alice
`committer(pattern)`	Commits by a specific committer.	`committer("GitHub Actions")`
`date(from..to)`	Commits within a date range.	`date("2 weeks ago..now")`
`description(pattern)`	Commits with a message matching a pattern.	`description("feat:.*")`
`file(pattern)`	Commits that modified files matching a pattern.	`file("src/utils.js")`
`root()`	The root commit(s) of the repository.	`root()`
`heads()`	Commits that are heads (have no children).	`heads()`
`branches()`	Commits that are the tips of `jj` branches.	`branches()`
`tags()`	Commits that are `jj` tags.	`tags()`
`public()`	Commits that are part of a public Git remote.	`public()`
`hidden()`	Commits that are hidden (e.g., obsolete).	`hidden()`
`mine()`	Commits authored by the current user.	`mine()`
`not(revset)`	Negates a revset (commits NOT in the given set).	`not(public())`
`latest(N, revset)`	The N latest commits within the specified revset.	`latest(5, all())` (last 5 commits overall)
`ancestors(revset)`	All ancestors of the commits in the revset.	`ancestors(@)`
`descendants(revset)`	All descendants of the commits in the revset.	`descendants(main)`
`grep(pattern)`	Commits whose message contains the pattern.	`grep("bugfix")`
		`git_ref("origin/main")`

Predicate	Description	Example
`git_ref(ref_name)`	A specific Git reference (branch, tag, HEAD).	
`git_head()`	The current Git HEAD.	`git_head()`

⚡ **Quick Note:** Many predicates accept regular expressions or glob patterns depending on the context. Always check the official `jj` documentation for specifics if you encounter unexpected behavior. As of 2026-05-19, the `jj` project is actively developed, and its `main` branch documentation is the most up-to-date resource.

Step-by-Step Implementation: Querying Your History

Let's put these concepts into practice. We'll create a dummy `jj` repository and populate it with some commits to experiment with. This hands-on exercise will solidify your understanding.

First, create a new directory and initialize a `jj` repo. We'll use the `--git` flag to enable Git interoperability from the start.

```
mkdir jj-revset-playground
cd jj-revset-playground
jj init --git
```

Now, let's add a few commits to build a small, representative history. We'll create initial content, then add features and a bug fix, also introducing different authors to demonstrate filtering.

```
echo "Initial content for the project." > file.txt
jj new -m "feat: initial setup"
jj branch create main @

echo "Adding more functionality to the application." >> file.txt
jj new -m "feat: add more features"

echo "Fixing a critical bug in the newly added feature." >> file.txt
jj new -m "fix: resolve feature bug" --author "Alice <alice@example.com>"

echo "Implementing another exciting feature." >> file.txt
jj new -m "feat: implement another feature"

echo "WIP: Exploring a half-baked idea for future development." >> file.txt
jj new -m "WIP: working on a new idea" --author "Bob <bob@example.com>"

# Let's see our current history with relevant details
jj log -T 'commit_id description author'
```

You should see a history similar to this (commit IDs will vary, but the order and descriptions should match):

```
o  a1b2c3d4 WIP: working on a new idea Bob <bob@example.com>
o  e5f6g7h8 feat: implement another feature Your Name <your.email@example.com>
o  i9j0k1l2 fix: resolve feature bug Alice <alice@example.com>
o  m3n4o5p6 feat: add more features Your Name <your.email@example.com>
o  q7r8s9t0 feat: initial setup Your Name <your.email@example.com>
```

Scenario 1: Finding Specific Commits with Basic Revsets

Let's use `jj log -r` to view specific parts of our history. The `-T` flag allows us to customize the output format.

1. **Your current working copy commit (`@`):**

```
jj log -r @ -T 'commit_id description'
```

```
This should show your latest commit: `WIP: working on a new idea`.
```

1. **The direct parent of your working copy (`@-`):**

```
jj log -r @- -T 'commit_id description'
```

```
You'll see `feat: implement another feature`.
```

1. **The grandparent of your working copy (`@--`):**

```
jj log -r @-- -T 'commit_id description'
```

```
This reveals `fix: resolve feature bug`.
```

1. **The root commit(s) of the repository (`root()`):**

```
jj log -r 'root()' -T 'commit_id description'
```

```
This will show `feat: initial setup`.
```

Scenario 2: Combining Revisions with Operators

Now, let's use operators to build more complex selections. Remember, parentheses are crucial for defining the order of operations in complex Revsets.

1. **Show the working copy commit AND its parent (`|` for union):**

```
jj log -r '@ | @-' -T 'commit_id description'
```

```
This will list both `WIP: working on a new idea` and `feat: implement another
feature`.
```

1. **Show all commits between `main` and the working copy (exclusive of `main`) (`..` for exclusive range):**

```
jj log -r 'main..@' -T 'commit_id description'
```

```
This should show all commits *after* `feat: initial setup` up to and including
your current WIP commit.
```

1. **Show all commits from `main` up to the working copy (inclusive of `main`) (`..=` for inclusive range):**

```
jj log -r 'main..=@' -T 'commit_id description'
```

```
This is similar to the above, but *includes* the `main` commit itself.
```

1. **Show all ancestors of the `main` branch (`ancestors()` predicate):**

```
jj log -r 'ancestors(main)' -T 'commit_id description'
```

```
In our simple linear history, this will show `main` and all commits before it,
effectively the entire history up to and including the `main` commit.
```

Scenario 3: Filtering with Predicates

Let's filter based on commit attributes. This is where you can quickly narrow down your search based on metadata.

1. **Find all commits authored by 'Alice' (`author()` predicate):**

```
jj log -r 'author("Alice")' -T 'commit_id description author'
```

```
You should see `fix: resolve feature bug Alice <alice@example.com>`.
```

1. **Find all commits with 'feat:' in their description (`description()` predicate):**

```
jj log -r 'description("feat:")' -T 'commit_id description'
```

```
This will list `feat: initial setup`, `feat: add more features`, and `feat:
implement another feature`.
```

1. **Find the latest 2 commits in the entire repository (`latest()` predicate):**

```
jj log -r 'latest(2, all())' -T 'commit_id description'
```

```
This should give you your two most recent commits, `WIP: working on a new
idea` and `feat: implement another feature`.
```

Scenario 4: Advanced Combination for Real-World Scenarios

This is where Revsets become incredibly powerful for daily `jj` workflows. Combining operators and predicates allows for surgical precision, especially in branchless workflows.

1. **Find all local commits that are not yet part of any public Git remote (useful before `jj push`):** (For this example, assume you've *`jj git push`*ed *`main`* at some point, making its ancestors *`public()`*. If not, *`public()`* might be empty, and this will show all your commits.)

```
jj log -r 'all() - public()' -T 'commit_id description'
```

```
This Revset uses the `all()` pseudo-commit to represent every commit `jj`
knows about, then subtracts (`-`) any commits that are considered `public()`
(i.e., known to a Git remote). This is your "unpushed local work."
```

1. **Find all bugfix commits (`fix:`) in the current branch's history (`main..@`):**

```
jj log -r 'description("fix:") & (main..@)' -T 'commit_id description'
```

```
This command combines two powerful ideas: filtering by description and
limiting the search to a specific range. The parentheses around `main..@`
ensure that the range is evaluated first, then the intersection (`&`) with the
description filter is applied. This should yield `fix: resolve feature bug`.
```

1. **Rebase all your commits since `main` onto the `main` branch's current tip:** This is a common operation in branchless workflows. You're saying, "take all my work (`main..@`) and put it on top of where `main` is now."

```
jj rebase -s 'main..@' -d main
```

```
This command is a prime example of Revsets enabling precise control over
mutable history. It tells `jj` to take the commits starting *after* `main` up
to and including the working copy (`-s 'main..@'`), and rebase them onto the
`main` commit itself (`-d main`). This is how you keep your local stack of
changes clean and up-to-date with a shared baseline.
```

Mini-Challenge: Targeted Commit Search

It's your turn to craft a Revset! This challenge will test your ability to combine multiple concepts.

- **Challenge:** Find all commits authored by "Bob" or "Alice" in the history of the `main` branch (meaning, any commit that is an ancestor of `main`), but exclude any commits that have "WIP" in their description. List their commit ID and description.
- **Hint:** You'll need `author()`, `ancestors()`, `description()`, `|` (union), and `-` (difference) operators. Pay attention to parentheses for grouping!
- **What to observe/learn:** This challenge requires you to combine multiple predicates and operators, demonstrating how flexible and powerful Revsets can be for isolating specific sets of changes in a complex history.

💡 💡 **NEED A LITTLE HELP? HERE'S A POSSIBLE SOLUTION:**

```
jj log -r '(author("Bob") | author("Alice")) & ancestors(main) -
description("WIP")' -T 'commit_id description'
```

Explanation: 1. `(author("Bob") | author("Alice"))`: This part selects all commits authored by either Bob or Alice. The parentheses ensure this union is treated as a single set. 2. `& ancestors(main)`: This intersects the previous set with all commits that are ancestors of the `main` branch, narrowing the scope to only relevant history that's part of the main lineage. 3. `- description("WIP")`: Finally, this subtracts any commits that have "WIP" in their description from the result, giving you the final, refined set of commits.

Common Pitfalls & Troubleshooting

Mastering Revsets takes practice, and you'll likely encounter some head-scratching moments. Here are common issues and how to approach them:

- **Syntax Errors:** A single misplaced parenthesis, an incorrect operator, or a typo can break your Revset. `jj` will usually provide a helpful error message pointing to the approximate location of the issue.
    - **Solution:** Start with smaller, simpler Revsets and gradually build up complexity. If a complex Revset fails, try breaking it down into smaller, testable parts using `jj log -r 'partial_revset'` to isolate the problem.
- **Misunderstanding `all()` vs. Current History:** `all()` refers to every commit `jj` knows about, including hidden or obsolete ones from its operation log. If you only want commits reachable from your current branch or a specific reference, `all()` might be too broad.
    - **Solution:** For commits on your current line of work, consider `main..@` (if `main` is your base) or `ancestors(@)`. For commits that are currently visible and active, `heads()` might be more appropriate.
- **Performance on Large Repositories:** Very complex Revsets, especially those involving `all()` and extensive filtering on large histories, can be slow. This is particularly true if `jj` has to traverse a vast graph.
    - **Solution:** For quick checks or when performance is critical, try to use more specific starting points or limit the scope with `latest(N, revset)`. Consider if a simpler Revset can achieve a similar goal.

- **Git Interoperability vs. `jj` Native:** Remember the `git_` prefixes (`git_ref`, `git_head`, `git_remote_branch`) when you need to refer to Git-specific references within a `jj` Revset. `jj`'s own branches (`branches()`) and tags (`tags()`) are distinct from their Git counterparts.
    - **Solution:** Be explicit about whether you're targeting a `jj` concept or a Git concept. For example, `main` might refer to a `jj` branch, while `git_ref("origin/main")` refers to the remote Git branch.
- **Debugging Revsets:** The most effective way to debug a Revset is to use `jj log -r 'YOUR_REVSET'` and observe the output. This allows you to see exactly which commits are being selected.
    - **Solution:** Build your complex Revset incrementally. Test each component with `jj log -r` before combining them. If a combination yields unexpected results, simplify it back to its parts.

Summary

Congratulations! You've taken a significant step towards truly mastering `jj` by diving deep into Revsets. This powerful query language is a cornerstone of `jj`'s flexible and efficient workflow.

Here are the key takeaways:

- **Revsets are `jj`'s powerful query language** for selecting commits based on their relationships, attributes, and content, acting like a database query for your history.
- **Operators** like `&` (intersection), `|` (union), `-` (difference), and `..` (range) allow you to combine and refine commit sets, defining precise relationships.
- **Predicates** such as `author()`, `description()`, `file()`, `heads()`, and `public()` enable filtering based on specific criteria, giving you granular control.
- **Precise commit selection** through Revsets is fundamental to leveraging `jj`'s mutable history, enabling advanced operations like targeted rebases and complex refactoring with confidence.
- **Practice is key!** Experiment with different combinations in your `jj` repositories to build intuition and truly understand how `jj` interprets your queries.

With Revsets in your toolkit, you're now equipped to navigate and manipulate your `jj` history with unparalleled control. In the next chapter, we'll explore **Stacked Changes and Branchless Workflows**, where the power of Revsets

truly shines, enabling you to manage your work in a linear, review-friendly manner without the overhead of traditional branches. Get ready to streamline your development process even further!

References

- Jujutsu (jj) GitHub Repository: https://github.com/jj-vcs/jj
- Jujutsu Revsets Documentation (main branch): https://github.com/jj-vcs/jj/blob/main/docs/revsets.md
- Jujutsu Tutorial (main branch): https://github.com/martinvonz/jj/blob/main/docs/tutorial.md

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

CHAPTER 09

Organizing Your Codebase: Workspaces and Repository Structure

Welcome back, intrepid developer! So far, you've mastered the basics of Jujutsu's unique approach to version control, from its mutable history to the powerful operation log. You've seen how `jj` empowers you to shape your history with confidence. But what happens when your project grows, or when you need to juggle multiple development lines simultaneously without creating a mess of separate Git clones?

This chapter introduces you to `jj`'s elegant solution: **workspaces**. We'll dive into how `jj` structures repositories and how workspaces allow you to manage multiple working directories, each potentially focused on a different task, all backed by a single, shared repository. This isn't just about saving disk space; it's about streamlining your workflow, improving context switching, and enabling more flexible development patterns.

By the end of this chapter, you'll understand how `jj` workspaces differ fundamentally from traditional Git clones, how to set them up, and how to leverage them for a more organized and efficient codebase. You'll be able to confidently manage multiple active development efforts within a single `jj` repository, a skill invaluable for modern software engineering.

Understanding Jujutsu's Repository Model

To fully appreciate `jj`'s workspaces, it's essential to grasp its fundamental repository model, which diverges significantly from Git.

The Central `.jj` Directory: Your True Repository

In Git, when you `git clone` a repository, you get a full copy of the repository's history, objects, and a working directory, all contained within the `.git` directory. If you need to work on two different features concurrently, you often end up with two separate clones of the same repository on your disk, each with its own `.git` directory. This duplication can be redundant and cumbersome.

`jj` takes a different, more centralized approach. At its core, a `jj` repository consists of a single, central `.jj` directory that stores all the repository's history, objects, and configuration. This `.jj` directory is the single source of truth for your project's version history. It typically resides at the root of your primary working directory.

What is a Workspace?

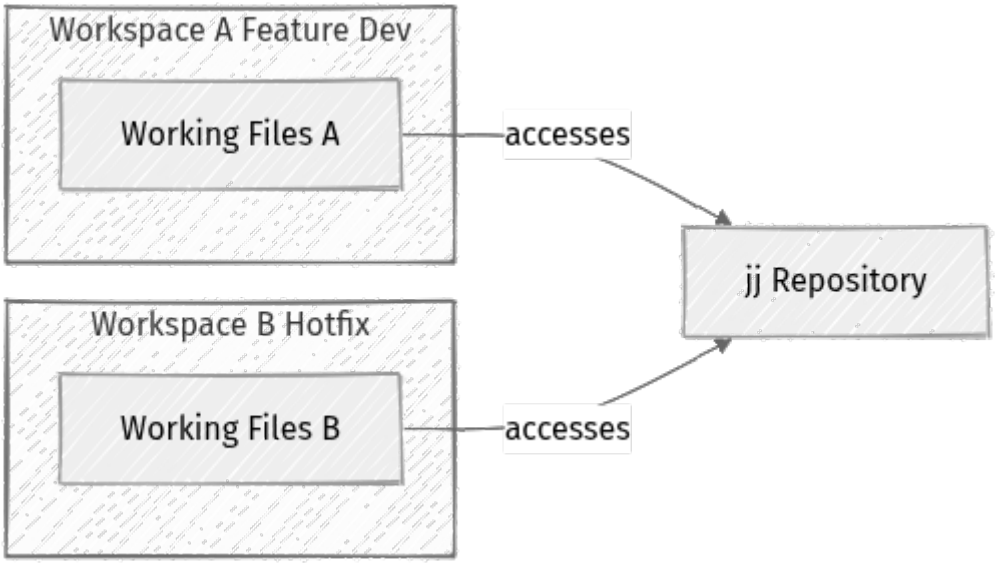
A `jj` **workspace** is a directory on your filesystem where you can edit files and interact with the `jj` repository. Each workspace has its own specific state, including which commit is currently checked out into its working directory. Crucially, multiple workspaces can coexist, all sharing the same underlying `.jj` directory.

📌 **Key Idea:** Imagine your central `.jj` directory as a vast library containing all your project's history books. A `jj` workspace is like a personal desk in that library. You can have multiple desks (workspaces), each with a different book (commit) open, but they all access the same collection of books in the central library (the `.jj` repository).

This model brings several powerful advantages:

- **Shared History, Always Current:** All workspaces connected to the same `.jj` directory share the exact same commit history and operation log. There's no need to `git fetch` or `git pull` history between local clones; it's always immediately available to all workspaces.
- **Reduced Redundancy:** You avoid duplicating the entire repository history on disk for each separate working directory, saving space and simplifying management.
- **Simplified Context Switching:** You can dedicate different workspaces to different tasks (e.g., one for a new feature, another for a bug fix, one for experimentation). Switching between them is as simple as changing directories, and each workspace maintains its own isolated working directory state.
- **Atomic Repository Operations:** Operations like `jj undo` or `jj rebase` affect the entire shared repository. This means changes made to history in one workspace are immediately reflected and accessible in others, providing a consistent view.

Let's visualize this shared structure:



In this diagram, `Workspace A` and `Workspace B` are distinct directories on your filesystem, each containing its own set of working files. However, both interact with and draw their history from the same `Central .jj Directory`. This means a commit created in `Workspace A` is instantly visible and accessible in `Workspace B`, and vice-versa.

Reinforcing Working-Copy-as-a-Commit

This workspace model beautifully reinforces `jj`'s "working-copy-as-a-commit" philosophy. In `jj`, your working directory is a commit (the "working-copy commit"). When you switch workspaces or create a new one, you're essentially setting up a new working directory that points to a specific (often new and empty) commit. All these working-copy commits, regardless of which workspace they belong to, are part of the same underlying `jj` repository.

Step-by-Step: Managing Multiple Workspaces

Let's put this into practice. We'll start by creating a new `jj` repository and then add a second workspace to it to simulate working on a separate task.

1. Initialize Your First `jj` Repository and Workspace

First, create a new directory for your project and initialize it as a `jj` repository. This directory will automatically become your first workspace, often referred to as the "default" workspace.

```
mkdir my_jj_project
cd my_jj_project
jj init
```

You should see output similar to:``` Initialized a new jj repo in "my_jj_project" with a Git backend.

```
This command did two things:
1.  It created the `my_jj_project` directory (if it didn't exist).
2.  It initialized the central `.jj` directory inside `my_jj_project`, setting
up your repository.
3.  It registered `my_jj_project` as your first workspace.

Now, let's create a simple file and commit it. This will give us some history
to work with.
```bash
echo "Hello from the main feature!" > main.txt
jj st
```

You'll see `main.txt` listed as an untracked change. Let's create a new commit for it.

```
jj new
jj files add main.txt
jj commit -m "feat: Initial commit with main.txt"
```

You've now got a `jj` repository with one workspace (`my_jj_project`) and one commit.

2. Add a Second Workspace for a Bug Fix

Now, imagine you're deep into developing a new feature in `my_jj_project`, but a critical bug report comes in for something unrelated. You need to switch contexts quickly without stashing or committing your in-progress work. This is a perfect scenario for adding a new workspace.

We'll add a new workspace named `bug_fix_workspace` to our existing `my_jj_project` repository.

```
jj workspace add bug_fix_workspace
```

You'll see output like:``` Added workspace "bug_fix_workspace" in "../bug_fix_workspace". Working copy now at: 23f7e366a7b2 (no description set)

```
Notice that `jj` created a *new directory* named `bug_fix_workspace` one level
up from `my_jj_project`. This is `jj`'s default behavior: new workspaces are
created as sibling directories to the initial repository directory. This keeps
the central `.jj` directory (which is inside `my_jj_project`) separate and
shared among all workspaces.

⚡ **Quick Note:** If you prefer the new workspace to be a subdirectory (e.g.,
`my_jj_project/new_feature`), you can specify a path relative to the current
```
```

```
working directory: `jj workspace add my_jj_project/new_feature`. However, the
default sibling behavior is often cleaner for managing distinct development
lines at the same level.

### 3. Explore and Work in the New Workspace

Let's navigate into our newly created workspace and see what's there.
```bash
cd ../bug_fix_workspace
ls
```

You should see `main.txt`! This immediately demonstrates that the new workspace has access to the files and history of the shared repository.

Let's check the status in this new workspace:

```
jj st
```

You'll see something like:``` Current working copy: 23f7e366a7b2 (no description set) Parent commit: f2a1b9c8d7e6 feat: Initial commit with main.txt

```
This shows that your working copy in `bug_fix_workspace` is currently on a
new, empty commit whose parent is the "Initial commit" you made earlier. This
is your isolated starting point for the bug fix.

Now, let's make a change specific to this `bug_fix_workspace`.
```bash
echo "Fixing a critical bug!" > bug_fix.txt
jj new
jj files add bug_fix.txt
jj commit -m "fix: Implemented critical bug fix"
```

4. Observe Changes Across Workspaces

Now for the magic! Let's go back to our original `my_jj_project` workspace and see if we can observe the `bug_fix.txt` file and the new commit.

```
cd ../my_jj_project
ls
```

You will not see `bug_fix.txt` here. Why? Because `my_jj_project`'s working copy is currently on its own (empty) commit, which doesn't include the changes from `bug_fix_workspace`. Each workspace maintains its own working copy state.

However, the commit itself is part of the shared history. Let's verify that using `jj log`.

```
jj log
```

You should now see both commits in the history, including the "fix: Implemented critical bug fix" commit, even though you made it in a different workspace!

```
@ 6f7a8b9cde0f (my_jj_project) (working copy) (no description set)
o 1234567890ab (bug_fix_workspace) fix: Implemented critical bug fix
o f2a1b9c8d7e6 feat: Initial commit with main.txt
o 000000000000 (empty) (root)
```

(Note: Commit hashes will differ for you. The key is to see both commits and their associated workspace names.)

This clearly illustrates the shared history. The `bug_fix_workspace` commit is instantly visible from `my_jj_project`, even though its files aren't checked out here. This is incredibly powerful for keeping track of all ongoing development.

5. Listing and Forgetting Workspaces

You can always see which workspaces are linked to your current repository using `jj workspace list`. This command provides a quick overview of all active development contexts.

```
jj workspace list
```

Output:``` my_jj_project: 6f7a8b9cde0f (no description set) bug_fix_workspace: 1234567890ab fix: Implemented critical bug fix

```
This command shows all active workspaces, their names, their current working-
copy commit ID, and a short description.

If you're done with a workspace (e.g., the bug fix is complete and merged),
you can remove its link to the repository using `jj workspace forget`. This
command *does not delete the directory* or its files; it just severs the link
to the `.jj` repository. You can then manually delete the directory if you
wish.
```bash
jj workspace forget bug_fix_workspace
```

Output:``` Forgot workspace "bug_fix_workspace".

```
Now, if you run `jj workspace list` again, `bug_fix_workspace` will be gone
from the list of active workspaces. The `bug_fix_workspace` directory and its
contents (`bug_fix.txt`) are still on your disk, but they are no longer
managed by `jj`.

To clean up the physical directory, you would then `rm -rf ../
bug_fix_workspace`.

## Mini-Challenge: Feature and Experimentation Workspaces
```
```

```
Let's put your new knowledge of workspaces to the test with a slightly more
complex scenario.

**Challenge:**
1.  Start with a fresh `jj` repository in a new directory called `my_app`.
2.  Create an initial commit with a `main.py` file containing `print("Hello,
World!")`.
3.  Add a new workspace called `feature_login`.
4.  In `feature_login`, modify `main.py` to add a new function `def
login(user): print(f"User {user} logged in!")` and call it. Commit this
change.
5.  Back in your original `my_app` workspace, add *another* new workspace
called `experiment_db`.
6.  In `experiment_db`, add a new file `database.py` with some experimental
database connection code (e.g., `print("Connecting to database...")`). Commit
this.
7.  Return to your original `my_app` workspace and use `jj log` to observe all
three commits from your different workspaces.

**Hint:**
Remember the commands: `mkdir`, `cd`, `jj init`, `jj new`, `jj files add`, `jj
commit -m`, `jj workspace add <name>`, `jj log`. Pay close attention to which
directory you're in before executing `jj` commands or creating files.

**What to observe/learn:**
You should see that commits from `feature_login` and `experiment_db` are both
visible in the `jj log` from your initial `my_app` workspace. This
demonstrates how a single `.jj` repository can effectively manage multiple
independent lines of development. You also won't see `database.py` or the
changes to `main.py` (from `feature_login`) in your *initial* `my_app`
workspace's file system. This reinforces that each workspace maintains its own
working copy state while sharing a common history.

## Common Pitfalls & Troubleshooting

Working with workspaces introduces new powerful concepts, but also a few areas
where developers new to `jj` might stumble.

- **Misunderstanding Shared History:** A common mistake for Git users is
thinking that `jj` workspaces are completely isolated, similar to separate Git
clones. Remember, they *share* the same underlying `.jj` directory and thus
the same history. If you `jj rebase`, `jj amend`, or `jj squash` a commit in
one workspace, that change is immediately visible (and potentially affects)
other workspaces, as you're modifying the *shared* repository history. This is
a powerful feature, not a bug, but it requires a mental model shift.
    - `🧠 Important:` Changes to history are global to the `jj` repository,
not local to a workspace. Always be mindful of this when rewriting history.
- **Forgetting Your Current Workspace:** When you have multiple workspaces,
it's easy to lose track of which directory you're currently in and, by
extension, which workspace is active. This can lead to making changes in the
wrong place.
    - **Tip:** Always check your shell prompt (if configured to show the
current directory) or use `pwd` and `jj st` to confirm your context before
making changes. `jj workspace list` also reminds you of all active workspaces.
- **Accidental File Modifications:** If you open files from one workspace in
your editor, but your terminal is in another workspace, you might make changes
to files that aren't part of your *current* working-copy commit. This can lead
to confusion when `jj st` doesn't show expected changes.
    - **Tip:** Ensure your editor and terminal are consistently aligned with
the workspace you intend to modify. Many IDEs integrate well with `jj` and can
show the current commit.
- **Deleting Workspaces Incorrectly:** `jj workspace forget` only severs the
```

```
logical link between the working directory and the `.jj` repository. It does
*not* delete the physical directory or its files.
    - **Tip:** If you want to fully remove a workspace's directory from your
filesystem, you must do so manually (e.g., `rm -rf`) *after* running `jj
workspace forget`.

## Summary

In this chapter, we've explored one of `jj`'s most powerful organizational
features: **workspaces**. This model provides a fresh perspective on managing
your codebase, especially when dealing with concurrent development efforts.

Here are the key takeaways:

- **Centralized Repository (`.jj`):** Unlike Git's distributed clones, `jj`
uses a single `.jj` directory to store all history and objects for a project.
- **Multiple Workspaces:** Workspaces are separate physical directories, each
acting as an independent working copy, all linked to and sharing the central
`.jj` repository.
- **Key Benefits:** Workspaces offer reduced disk space, simplified context
switching, isolated working directories, and a consistent, shared view of
history across all development lines.
- **Reinforces Working-Copy-as-a-Commit:** Each workspace's current state *is*
a working-copy commit within the shared repository.
- **Core Commands:**
    - `jj init`: Initializes a new repository and its first workspace.
    - `jj workspace add <name>`: Creates a new workspace linked to the current
repository, typically as a sibling directory.
    - `jj workspace list`: Shows all active workspaces, their names, and their
current working-copy commits.
    - `jj workspace forget <name>`: Disconnects a workspace from the
repository (does not delete the physical directory).
- **Mental Model Shift:** This workspace model significantly differs from
Git's "repository per clone" approach, fostering a more integrated and
efficient way to manage concurrent development.

Understanding `jj` workspaces is crucial for leveraging its full potential,
especially in projects requiring parallel development efforts or frequent
context switching. By adopting this model, you can streamline your development
workflow and maintain a cleaner, more organized codebase.

In the next chapter, we'll delve deeper into how `jj` interacts with
traditional Git, allowing you to seamlessly integrate `jj`'s powerful
workflows into existing Git-based projects and collaborate effectively with
others who might still be using Git.

## References

- [Jujutsu Official GitHub Repository](https://github.com/jj-vcs/jj)
- [Jujutsu User Manual (Workspaces section)](https://github.com/jj-vcs/jj/
blob/main/docs/tutorial.md#workspaces)
- [Jujutsu Releases](https://github.com/jj-vcs/jj/releases)
- [Jujutsu GitHub Integration Guide](https://github.com/jj-vcs/jj/blob/main/
docs/github.md)

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

CHAPTER 10

Real-World Scenarios: Feature Development, Refactoring, and Debugging

Real-World Scenarios: Feature Development, Refactoring, and Debugging

Welcome back! In previous chapters, we laid the groundwork for understanding Jujutsu (`jj`), exploring its unique working-copy-as-a-commit model, the power of `revsets`, and the safety net of the operation log. Now, it's time to bridge theory with practice. How do these innovative features translate into tangible benefits in your daily coding life?

This chapter focuses on applying `jj` to common, real-world software engineering challenges. We'll dive into practical scenarios that highlight `jj`'s ability to simplify complex tasks, making you more efficient and confident in managing your codebase's history. Specifically, we'll cover:

- **Streamlined Feature Development:** Building new features iteratively with stacked changes.
- **Effortless Refactoring:** Modifying code, even in older commits, with automatic propagation.
- **Rapid Debugging:** Pinpointing and fixing bugs by exploring and manipulating past states.

By the end of this chapter, you'll have a clear understanding of how `jj` can transform your development workflow, turning traditional VCS headaches into smooth, manageable processes. This guide assumes you're familiar with basic `jj` commands like `jj status`, `jj log`, `jj diff`, `jj commit`, and have a foundational grasp of `revsets` and the operation log from earlier chapters. Let's put `jj` to work!

Scenario 1: Streamlined Feature Development with Stacked Changes

Developing new features often involves a series of logical, dependent steps. You might build a foundation, then add functionality on top, then refine an earlier part. In traditional VCS like Git, modifying an early commit in such a "stack" typically requires a cumbersome interactive rebase. `jj` fundamentally changes this, making stacked changes a natural and easy part of your workflow.

Core Concept: Iterative Development with Mutable History

`jj`'s core strength for feature development lies in its working-copy-as-a-commit model and mutable history. Unlike Git, where commits are generally considered immutable, `jj` treats them as flexible entities. This means you can freely modify any commit in your history, and `jj` intelligently re-applies subsequent changes on top, minimizing manual effort.

Why this matters:

This paradigm encourages creating smaller, more focused commits, which are easier to review and understand. If code review feedback or a new insight requires changes to an earlier commit in your feature stack, `jj` handles the cascading updates automatically. This eliminates the fear of "breaking history" and makes iterative refinement a joy, not a chore.

Step-by-Step: Building and Refining a Feature

Let's simulate building a new feature, `user-profile-enhancements`, composed of several logical steps. We'll start by setting up a new `jj` repository.

```
# Create a new directory and initialize a jj repository
mkdir jj-feature-dev
cd jj-feature-dev
jj init --git # Initialize a jj repo with Git compatibility for future remote
interaction
echo "Initial content for the project." > README.md
jj commit -m "Initial project setup" # Explicitly commit the initial state
```

Now, let's begin developing our feature.

Step 1: Implementing Basic Profile Display

We'll start with the foundational piece: displaying a user profile.

```
# Create a new file for user profile logic
echo "function displayProfile(user) { /* Displays basic user info */ }" > src/
```

```
profile.js
jj commit -m "feat: Add basic user profile display function"
```

You've just created your first feature commit. `jj` automatically places it on top of your "Initial project setup" commit.

Step 2: Adding Profile Editing Functionality

Next, we'll add the ability to edit the profile. This logically depends on the existence of the profile display.

```
# Append profile editing logic to the same file
echo "function editProfile(user) { /* Allows user to modify their profile
*/ }" >> src/profile.js
jj commit -m "feat: Implement profile editing functionality"
```

Now you have two commits stacked on top of each other. Let's visualize the history:

```
jj log -L
```

You should observe an output similar to this (commit IDs will vary):``` o 8b7a4c2e feat: Implement profile editing functionality | @ c2d1e0f9 feat: Add basic user profile display function | o b1a0d3c5 Initial project setup |/ o 00000000 (empty)

```
The `@` symbol points to your current working copy commit, which is the latest
feature commit.

**Step 3: Modifying an Earlier Commit (`jj edit` and `jj amend`)**

Imagine you realize that the `displayProfile` function (in your *first*
feature commit) needs an additional parameter, say `options`, for
customization. In Git, this would typically involve an interactive rebase
(`git rebase -i`). In `jj`, it's much more straightforward.
```bash
jj edit @- # Move the working copy to the parent of the current commit (the
first feature commit)
```

After running `jj edit @-`, your working copy is now at the state of "feat: Add basic user profile display function". If you run `jj log -L` again, you'll see your second commit ("feat: Implement profile editing functionality") is now marked as `(abandoned)`. This is `jj`'s way of saying it's temporarily out of the main linear history, but it's not lost. It will be re-applied.

Now, modify `src/profile.js` to add the new parameter.
```

```
# Manually edit src/profile.js to change the function signature
# Example: Change 'function displayProfile(user)' to 'function
displayProfile(user, options)'
# For Linux/macOS, you could use: sed -i 's/function displayProfile(user)/
function displayProfile(user, options)/' src/profile.js
# On Windows, please open src/profile.js in your text editor and make the
change.
```

Once modified, `amend` the current working copy commit (which is the "feat: Add basic user profile display function" commit).

```
jj amend # Amends the current working copy commit with your changes
```

Run `jj log -L` again. Observe the magic!``` o f2e3d4c1 feat: Implement profile editing functionality (2026-05-19) | @ a1b2c3d4 feat: Add basic user profile display function (2026-05-19) | o b1a0d3c5 Initial project setup (2026-05-19) |/ o 00000000 (empty)

```
The "feat: Add basic user profile display function" commit has been updated,
and "feat: Implement profile editing functionality" has been automatically
rebased on top of the *new* version of the first commit. `jj` handled the
rebase seamlessly, propagating your changes through the stack.

`🧠 Important:` When you `jj edit` an older commit and then `jj amend` it,
`jj` doesn't just change that single commit. It effectively creates a *new
version* of that commit and then automatically *rebases all subsequent
commits* onto this new version. This is the core of `jj`'s mutable history
power.

**Step 4: Squashing Commits for a Cleaner History (`jj squash`)**

Before merging your feature into the main branch, you might want to combine
these two logically related commits into a single, cohesive change for a
cleaner history.
```bash
jj squash @- # Squash the current commit into its parent
```

This command squashes the current commit (the one with profile editing) into its parent (the one with basic display, which you just modified). `jj` will open your configured editor, allowing you to combine and refine the commit messages. Save and close the editor.

After squashing, `jj log -L` will show a single, consolidated feature commit:``` o d5c6b7a8 feat: Add user profile display and editing (2026-05-19) | @ b1a0d3c5 Initial project setup (2026-05-19) |/ o 00000000 (empty)

```
You've successfully built a feature iteratively, modified an earlier part, and
then cleaned up the history—all with minimal fuss and maximum flexibility.
```
```

````
### Mini-Challenge: Reordering Commits

**Challenge:** Create a new commit that adds a `utils.js` file with a simple
helper function (e.g., `isValidString`). Then, reorder your history so this
`utils.js` commit becomes a direct parent of your `feat: Add user profile
display and editing` commit.

**Hint:**
1.  Use `jj new` to create a new empty commit.
2.  Add `src/utils.js` with your helper function.
3.  `jj commit -m "feat: Add utility helper functions"`.
4.  Then, think about how `jj rebase` can change a commit's parent. You'll
need to rebase your *feature commit* onto the *new utility commit*. You can
use `jj rebase -s <feature_commit_id> -d <utility_commit_id>`.

**What to observe/learn:** `jj rebase` is incredibly versatile. It can change
a commit's parent, effectively reordering history and making dependencies
explicit. This is a powerful tool for structuring your changes logically.

## Scenario 2: Effortless Refactoring with Mutable History

Refactoring is a vital practice for maintaining code quality and readability,
but it can be daunting in traditional VCS, especially when changes span
multiple existing commits or require modifications to older code. `jj`'s
mutable history model makes refactoring a much safer and more pleasant
experience.

### Core Concept: Refactoring in Place, Propagating Changes

`jj` empowers you to "travel back in time" to any older commit, apply a
refactoring, and then trust `jj` to automatically rebase all subsequent
commits on top of your refactored base. This means you can identify and fix
structural issues in existing code, even if it's deeply buried in your
history, without the manual burden of re-applying all your newer changes. The
`jj op log` also provides an "undo" safety net for any experimental refactors.

#### Why it exists:
This approach directly addresses the challenge of "refactoring debt." Often,
as a codebase evolves, you discover a better architectural pattern or a more
efficient way to structure code that was written several commits ago. `jj`
liberates you from living with suboptimal code or performing painful, error-
prone manual rebases. You can fix the code at its logical origin and let the
system handle the updates.

### Step-by-Step: Applying a Refactor to Past Code

Let's continue in our `jj-feature-dev` repository. We'll add a few more
commits and then perform a refactor that affects an earlier commit.
```bash
# Ensure we have a few commits on top of our feature
cd jj-feature-dev
jj new # Create a new empty commit as the current working copy commit
echo "console.log('User profile loaded.');" >> src/profile.js
jj commit -m "chore: Add debug logging to profile module"
jj new
echo "Bug introduced: data missing." >> src/profile.js
jj commit -m "fix: Accidental bug in profile data handling"
jj log -L
````
````

Now, imagine we want to refactor how `displayProfile` and `editProfile` are managed. Instead of being global functions, we decide they should be methods within a `ProfileManager` class for better encapsulation. This is a structural change that affects our `feat: Add user profile display and editing` commit.

Step 1: Identify the Refactoring Target

First, let's find the commit ID of our combined feature commit, "feat: Add user profile display and editing."

```
jj log --short
```

Let's assume its ID is `d5c6b7a8`.

Step 2: Go Back and Apply the Refactoring

We'll use `jj edit` to temporarily move our working copy to the state of that specific commit.

```
jj edit d5c6b7a8 # Replace with your actual commit ID for the feature
```

Your working copy is now at the state of `d5c6b7a8`. If you check `jj log -L`, you'll see the subsequent commits (like "chore: Add debug logging..." and "fix: Accidental bug...") are now marked `(abandoned)`. Again, they are not lost; `jj` will bring them back.

Now, let's perform the refactoring. We'll simulate moving the functions into a `ProfileManager` class.

```
# Simulate refactoring src/profile.js
# Create a new class structure
echo "class ProfileManager {" > src/profile_manager.js
echo "  displayProfile(user, options) { /* ... */ }" >> src/profile_manager.js
echo "  editProfile(user) { /* ... */ }" >> src/profile_manager.js
echo "}" >> src/profile_manager.js
echo "export const profileManager = new ProfileManager();" >> src/
profile_manager.js

# For simplicity in this demo, we'll conceptually replace the old file.
# In a real scenario, you'd modify src/profile.js to use the new class or
rename it.
rm src/profile.js
```

After making your refactoring changes, you need to "commit" them. We'll amend the current feature commit to include this refactoring, effectively changing how the feature was originally implemented.

```
jj amend --message
"refactor(profile): Introduce ProfileManager class for user operations"
```

You can also choose to create a new commit for the refactoring on top of `d5c6b7a8` using `jj new --message "..."`. The choice depends on whether you want the refactor to be part of the original commit or a separate, subsequent change. For this scenario, amending is often preferred to clean up the original feature implementation.

Step 3: Observe Automatic Propagation

Now, let's observe your history:

```
jj log -L
```

You'll see that your `refactor` (which is now part of your main feature commit) is in the history, and the `chore: Add debug logging...` and `fix: Accidental bug...` commits have been automatically rebased on top of this new, refactored base. `jj` handles the propagation of changes, meaning you don't have to manually re-apply the logging or bug fix on top of your refactored code.

📌 `Key Idea:` `jj edit` followed by `jj amend` (or `jj new`) allows you to non-destructively modify any commit in your history. `jj` then automatically re-applies all descendant commits, minimizing manual rebase effort.

Mini-Challenge: Using `jj undo` During Refactoring

Challenge: Imagine you've made a complex refactoring as above, but after reviewing the rebased history and perhaps running tests, you realize the refactor introduced more problems than it solved, or it's simply not the right approach. Use `jj undo` to revert your entire refactoring session, effectively going back to the state before you started the refactor.

Hint: The `jj op log` (`jj ol`) shows a history of your `jj` operations. You can identify the operation where you started the refactor and use `jj undo` to revert to the state just before it.

What to observe/learn: The operation log is your ultimate safety net for complex changes. `jj undo` can revert entire sequences of operations, making experimentation and bold refactorings much less risky.

Scenario 3: Rapid Debugging with History Exploration

Debugging can be one of the most time-consuming aspects of development, especially when a bug was introduced many commits ago and its origin is unclear. `jj`'s powerful history navigation and manipulation, combined with the comprehensive operation log, provide a robust toolkit for quickly finding, understanding, and fixing issues.

Core Concept: Pinpointing Bugs and Applying Fixes

`jj` treats every state of your repository as a first-class citizen. You can effortlessly jump to any past commit, examine its exact state, run tests against it, and even craft a fix directly on that problematic commit. Once the fix is created, `jj` can then rebase your current work on top, ensuring the fix propagates correctly and cleanly into your current development line.

What problem it solves:

This workflow offers a significant advantage over traditional `git bisect` or manual cherry-picking. Instead of just identifying a commit, you can interact with the exact historical state where the bug originated. You can test your fix in isolation at that point and then seamlessly integrate it into your current work, resulting in a cleaner, more accurate history.

Step-by-Step: Debugging a Simulated Issue

Let's continue with our `jj-feature-dev` repository. We'll introduce a subtle bug and then use `jj`'s capabilities to find and fix it efficiently.

```
cd jj-feature-dev
jj new # Create a new empty commit
echo "function processUserData(data) { return data; }" >> src/
data_processor.js
jj commit -m "feat: Add basic data processing utility"
jj new
echo "function processUserData(data) { if (!data) throw new Error('No data
provided'); return data.toUpperCase(); }" >> src/data_processor.js # This is
our bug!
jj commit -m "fix: Validate and format user data"
jj new
echo "console.log(profileManager.displayProfile({name: 'Alice'}));" >> src/
main.js
jj commit -m "feat: Integrate profile display into main app"
jj log -L
```

Now, imagine you discover that your application crashes with an error like "data.toUpperCase is not a function" when `processUserData` is called with certain valid inputs (e.g., a number). You suspect the bug was introduced in the "fix: Validate and format user data" commit because that's where the `toUpperCase()` logic was added.

Step 1: Inspect History and Identify Potential Culprits

First, let's use `jj log` to examine recent changes and identify the commit where the bug might have been introduced.

```
jj log -r 'root()..' --short
```

This command shows all commits from the repository's root to the current working copy. Based on our scenario, the "fix: Validate and format user data" commit (`e8f9g0h1` in our example, your ID will differ) looks like the primary suspect.

Step 2: Examine the Faulty Commit's Changes

Let's use `jj diff` to see exactly what changes were introduced in that specific commit. Assuming the ID is `e8f9g0h1` for "fix: Validate and format user data":

```
jj diff e8f9g0h1
```

You'll see the line `return data.toUpperCase();`. This immediately looks suspicious for inputs that are not strings. The bug is confirmed.

Step 3: Go Back and Fix the Bug

Now, let's `edit` that commit to apply a precise fix.

```
jj edit e8f9g0h1 # Replace with the actual commit ID of the bug
```

Your working copy is now at the exact state of `e8f9g0h1`. Any subsequent commits (like "feat: Integrate profile display...") are marked `(abandoned)`.

Let's fix the bug by ensuring `data` is a string before calling `toUpperCase()`.

```
# Correct src/data_processor.js
# Manually edit the file or use sed (Linux/macOS)
# Example:
# Change: return data.toUpperCase();
# To:     return typeof data === 'string' ? data.toUpperCase() :
```

```
String(data).toUpperCase();
# Or:     return String(data).toUpperCase(); // Simpler for this example
```

Now, `amend` the commit with the fix. This will update the "fix: Validate and format user data" commit.

```
jj amend --message "fix: Ensure data is string before calling toUpperCase in
data processor"
```

Step 4: Propagate the Fix and Verify

After amending, `jj` automatically rebases any subsequent commits (like "feat: Integrate profile display into main app") on top of your fixed commit.

```
jj log -L
```

You'll see the updated "fix: Ensure data is string..." commit, and any commits that were abandoned will have been re-applied on top. The bug is now fixed at its source, and your history is clean and accurate.

`⚡ Real-world insight:` For more complex debugging scenarios, you might use `jj restore --from <revset> --file <path>` to cherry-pick specific files or even individual hunks from other commits while `editing` a problematic one, allowing for highly granular control over your fix.

Mini-Challenge: Undoing a Debugging Session

Challenge: After fixing a bug and seeing the history update, you realize your fix introduced a new, worse problem, or perhaps you found an even better way to fix it. Use the operation log (`jj op log`) and `jj undo` to revert the entire sequence of debugging and fixing, effectively going back to the state before you started trying to fix the bug.

Hint: Run `jj op log` to see a list of your recent `jj` commands. Find the operation ID corresponding to the state before you started your fix (e.g., before the `jj edit` command). Then use `jj undo --op <operation_id>`.

What to observe/learn: The operation log is your ultimate safety net for `jj` commands. It allows you to rewind not just code changes but also the actions you took on your repository, providing powerful recovery capabilities for even the most complex debugging sessions.

Common Pitfalls & Troubleshooting

While `jj` dramatically simplifies many workflows, certain aspects can still be confusing, especially when transitioning from a Git-centric mindset.

- **Forgetting `jj`'s Mutable History:**
  - ⚠️ `What can go wrong:` Coming from Git, you might expect `jj amend` to only affect the last commit. In `jj`, `jj amend` always modifies the current working copy commit (`@`). If you `jj edit` an older commit, then `jj amend`, you are indeed changing that older commit, and `jj` will automatically rebase all its descendants.
  - **Pro tip:** Trust `jj` to rebase. It's designed for this. If you see `(abandoned)` commits, it's usually `jj` preparing to re-apply them on a new base.
- **Over-reliance on `jj undo` for Complex Rewrites:**
  - ⚠️ `What can go wrong:` While `jj undo` is fantastic for quick rollbacks of the last few operations, for very complex history rewrites involving many steps, repeatedly using `jj undo` might not be the clearest path.
  - **Pro tip:** For intricate history manipulation, it's often more transparent to use targeted commands like `jj rebase`, `jj fold`, `jj squash`, or `jj restore`. `jj undo` is best for immediate course corrections.
- **Initial Difficulty with `revsets` Syntax:**
  - ⚠️ `What can go wrong:` `revsets` are incredibly powerful for selecting revisions but can have a steep learning curve due to their rich syntax.
  - **Pro tip:** Start with simple `revsets` like `@` (current), `@-` (parent of current), `root()` (repository root), `main` (branch head), `feature..` (all commits reachable from `feature` not from its parent). Gradually explore more complex operators as needed. Keep the official `revsets` documentation handy.

- **Merge Conflicts During Automatic Rebase:**
  - ⚠️ `What can go wrong:` `jj` excels at automatically rebasing changes. However, if two commits modify the same lines of code in fundamentally incompatible ways, `jj` will pause and require you to resolve the conflicts manually, much like Git.
  - **Troubleshooting:** When conflicts occur, `jj` will inform you. Use `jj status` to see the unmerged files. Manually edit the files to resolve the conflict markers (e.g., `<<<<<<<`, `=======`, `>>>>>>>`). Once resolved, `jj commit --amend` will commit the resolution and allow `jj` to continue the rebase process.

Summary: Empowering Your Development Workflow

In this chapter, we've explored three crucial real-world scenarios, demonstrating how Jujutsu's unique features can dramatically enhance your daily development experience:

- **Feature Development:** `jj` makes building features iteratively with stacked changes incredibly intuitive and efficient. Modifying earlier commits in a stack is no longer a headache, thanks to `jj`'s automatic rebase capabilities.
- **Refactoring:** You can confidently refactor any part of your codebase, even old commits, knowing that `jj` will handle the complex propagation of changes and that `jj undo` provides a robust safety net for experimentation.
- **Debugging:** Pinpointing and fixing bugs in historical commits is streamlined. `jj edit` allows you to interact directly with past states, apply precise fixes, and then seamlessly integrate them into your current development line.

By embracing `jj`'s mutable history, powerful `revsets`, and comprehensive operation log, you're not just using a version control system; you're gaining a highly flexible and resilient tool that adapts to your workflow, rather than forcing you into rigid, traditional patterns. This flexibility fosters cleaner history, easier code reviews, and greater confidence in managing your codebase.

What's Next?

You've now covered the core workflows and essential real-world applications of `jj`. You're well on your way to mastering this powerful VCS. In the next chapter, we'll delve into even more advanced topics, including customizing `jj` for personal productivity, exploring advanced `revsets` patterns in depth, and integrating `jj` into CI/CD pipelines for large-scale projects. Get ready to unlock the full potential of Jujutsu!

References

- Jujutsu GitHub Repository: https://github.com/jj-vcs/jj
- Jujutsu Official Tutorial (on main branch): https://github.com/martinvonz/jj/blob/main/docs/tutorial.md
- Jujutsu Revsets Documentation: https://github.com/martinvonz/jj/blob/main/docs/revsets.md
- Jujutsu Commands Reference: https://github.com/martinvonz/jj/blob/main/docs/commands.md

---

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

CHAPTER 11

Optimizing Your Workflow: Customization and Productivity Hacks

Welcome back, fellow Jujutsu explorers! You've mastered the core concepts of `jj`, from its unique working-copy-as-a-commit model to navigating mutable history and leveraging the powerful operation log. Now, it's time to truly make `jj` your own.

In this chapter, we'll dive deep into customizing `jj` to fit your personal workflow like a glove. We'll explore configuration files, create powerful aliases for common commands, integrate `jj` with your favorite editors and diff tools, and even craft custom output templates. The goal is simple: to make your `jj` experience as efficient, intuitive, and productive as possible.

Think of this as equipping your `jj` toolkit with personalized shortcuts and powerful automation, transforming it from a robust VCS into an indispensable daily companion. Let's get started!

The Heart of Customization: `jj` Configuration Files

Just like many powerful command-line tools, `jj` uses configuration files to let you tailor its behavior. These files are written in the TOML format, a simple, human-readable data serialization language.

What are `jj` Configuration Files?

`jj` configuration files are plain text files that store settings and preferences for how `jj` should operate. They allow you to define everything from how commit messages are formatted to which external tools `jj` should use for diffing or merging.

Why do they exist? Configuration files exist to provide flexibility and consistency. Instead of typing out long command-line arguments every time, you can set defaults that `jj` automatically applies. This saves time, reduces errors, and ensures your preferred workflow is always active.

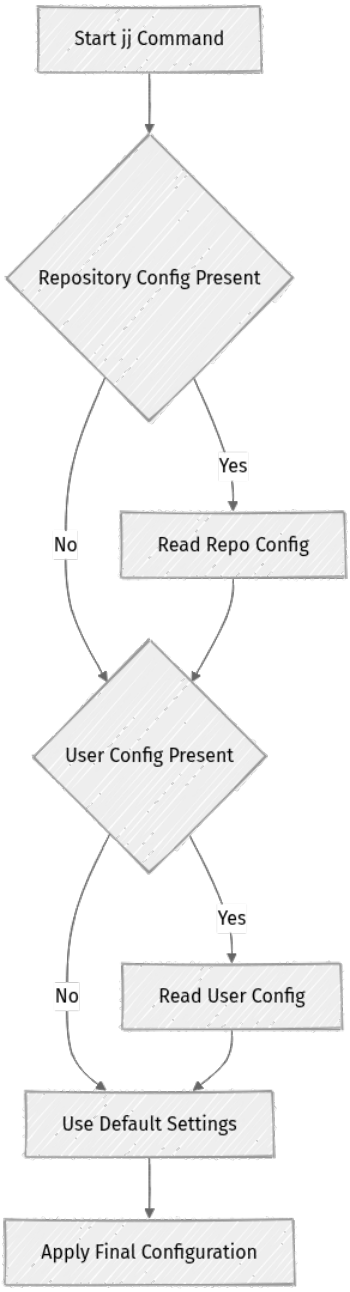
What problem do they solve? They solve the problem of repetitive command-line input and the need to adapt `jj`'s default behavior to individual preferences or project requirements. Without them, `jj` would be less adaptable and more cumbersome for daily use.

Configuration Lookup Hierarchy

`jj` looks for configuration in a specific, hierarchical order. This allows you to set global defaults and then override them for specific repositories:

1. **System-wide configuration:** (Least common for individual users)
2. **User-specific configuration:** `~/.jjconfig.toml` (or `%USERPROFILE%\.jjconfig.toml` on Windows). This is your primary file for global settings, aliases, and tool preferences that apply to all `jj` repositories.
3. **Repository-specific configuration:** `.jj/repo/config.toml` inside a `jj` repository. Settings here will override anything in your user-specific file for that particular repository, allowing project-specific customizations.

This hierarchical approach ensures you can have consistent global settings but fine-tune behavior for specific projects without affecting others.



Explanation of the flow:

- When you run a `jj` command, it first checks if you're inside a `jj` repository.
- If a repository-specific `.jj/repo/config.toml` exists, `jj` reads those settings first.
- Next, it reads your global `~/.jjconfig.toml` file.
- Any settings found in `~/.jjconfig.toml` will be overridden by identical settings found earlier in `.jj/repo/config.toml`.
- Finally, any settings not specified in either file will fall back to `jj`'s built-in default values.

Step-by-Step Implementation: Setting Up Your Global `~/.jjconfig.toml`

The `~/.jjconfig.toml` file is where most of your personal customization will live. If it doesn't exist, `jj` will simply use its default settings. Let's create one and add a common preference.

First, open your terminal.

```
# Create an empty config file if it doesn't exist
touch ~/.jjconfig.toml

# Open it with your favorite text editor (e.g., nano, vim, code)
# For example, using nano:
nano ~/.jjconfig.toml
```

Now, let's add a basic configuration that makes `jj`'s output more informative: always show the commit message when running `jj log` or `jj status`.

Add the following to your `~/.jjconfig.toml` file:

```
# ~/.jjconfig.toml
# Configuration for Jujutsu (jj) VCS
# Last updated: 2026-05-19

[ui]
# Always show the commit message in `jj log` and `jj status` output.
log-template = "commit_id: {commit_id}\nauthor: {author}\ndate: {time}\n\n{des
cription}"
```

How it functions:

- `[ui]` declares a section for User Interface related settings. Sections help organize your configuration.
- `log-template` is a string that defines how `jj` should format the output of commands like `jj log` and `jj status`.
- `{commit_id}`, `{author}`, `{time}`, and `{description}` are placeholders. `jj` replaces these with the actual data from each commit. For example, `{commit_id}` becomes the full commit hash, and `{description}` becomes the commit message. The `\n` characters create new lines in the output.

Save and close the file. Now, when you run `jj log` or `jj status` in a `jj` repository, you'll see the commit message included automatically!

```
# Try it out in a jj repo (make sure you have some commits)
# If you don't have a jj repo, create one:
# mkdir my_jj_repo && cd my_jj_repo && jj init
# echo "Hello Jujutsu" > README.md && jj commit -m "Initial commit"
```

```
# echo "Another line" >> README.md && jj commit -m "Add another line"

jj log -r @
jj status
```

You should now see more verbose output, including the commit message, directly in `jj log` and `jj status`.

Boosting Efficiency with Aliases

Aliases are short, custom commands that expand into longer, more complex `jj` commands. They are a massive productivity booster, letting you type less and achieve more. If you find yourself typing a repetitive `jj` command sequence, it's a perfect candidate for an alias.

What are `jj` Aliases?

`jj` aliases are custom shortcuts you define for `jj` commands. They allow you to assign a short, memorable name (like `st` for `status`) to a full `jj` command, including its arguments and options.

Why do they exist? Aliases exist to streamline your workflow and reduce cognitive load. They make frequently used commands quicker to execute and help bridge the gap for users migrating from other VCS tools (like Git) who are used to certain command shortcuts.

What problem do they solve? They solve the problem of typing long or complex commands repeatedly. They also allow you to create custom workflows by chaining multiple commands or applying specific `revsets` and templates with a single, simple command.

Step-by-Step Implementation: Defining Aliases in `~/.jjconfig.toml`

Aliases are defined in the `[aliases]` section of your `~/.jjconfig.toml` (or repository config).

Let's add some common aliases to our `~/.jjconfig.toml` file. Open it again:

```
nano ~/.jjconfig.toml
```

Add the following lines to the end of your file:

```
# ~/.jjconfig.toml
# ... existing ui section ...
```

```
[aliases]
# 'jj st' for 'jj status' - a common Git alias habit
st = "status"

# 'jj co' for 'jj checkout' - another Git habit, now for 'jj edit'
# In jj, 'checkout' is similar to 'edit' for moving the working copy
co = "edit"

# 'jj up' for 'jj update' - frequently used to move the working copy
up = "update"

# 'jj prev' to go to the parent of the current working-copy commit
# '@-' is the revset for the parent of the working copy
prev = "edit @-"

# 'jj next' to go to the child of the current working-copy commit
# '@+' is the revset for the child of the working copy
next = "edit @+"

# 'jj tree' to show the commit graph with a simplified template
# This combines 'log' with a custom template for a tree-like view
tree = "log -T 'commit_id.short} {branches} {description.first_line() |
indent(\"  \")}'"
```

How it functions:

- `[aliases]` marks the start of the aliases section in the TOML file.
- Each line under `[aliases]` defines an alias using the format: `alias_name = "full_command_string"`.
- When you type `jj alias_name`, `jj` replaces `alias_name` with `full_command_string` and executes it.
- Notice how the `tree` alias encapsulates a `log` command with a `revset` (`-T '...'`), demonstrating that aliases can indeed wrap complex command arguments and templates.

Save and close the file. Now you can use these shorter commands!

Practical Alias Examples

Let's see these aliases in action. Navigate to a `jj` repository.

```
# Instead of 'jj status', you can now type:
jj st

# Instead of 'jj edit <commit_id>', you can use 'jj co' to edit the working
copy
# For example, to edit the current commit:
jj co @

# Instead of 'jj update @', you can now type:
jj up
```

```
# To go to the parent commit of your current working copy:
jj prev

# To go to the first child commit of your current working copy:
jj next

# To view a simplified commit graph with our custom template:
jj tree
```

These small changes add up to significant time savings and a more fluid workflow.

Mini-Challenge: Create Your Own Alias

Challenge: Imagine you frequently need to view the current working-copy commit and all its immediate children, but only showing the short commit ID, the author, and the first line of the description. Create an alias called `jj children-log` that runs `jj log -r '@ | children(@)' -T 'commit_id.short} {author.name}: {description.first_line()}'`.

Hint: Add a new entry under the `[aliases]` section in your `~/.jjconfig.toml` file. Remember the structure `alias_name = "full_command_string"`.

What to observe/learn: How easy it is to extend `jj`'s functionality with simple aliases, and how `revsets` (like `@ | children(@)`) become even more powerful and accessible when combined with templates and aliased.

Integrating Your Favorite Tools: Editor and Diff Configuration

`jj` plays nicely with external tools. You can configure it to use your preferred text editor for commit messages and your favorite graphical diff tool for reviewing changes. This integration is key to a seamless development experience.

Configuring Your Editor for Commit Messages

By default, `jj` will use your system's `EDITOR` environment variable. If that's not set or you want to specify a different editor specifically for `jj`, you can configure it in `~/.jjconfig.toml`.

Let's set `jj` to use `code --wait` for VS Code, which is a popular choice. The `--wait` flag is crucial as it tells `jj` to wait for the editor to close the file before `jj` continues its operation.

Open `~/.jjconfig.toml` again:

```
nano ~/.jjconfig.toml
```

Add the following to your `[ui]` section:

```
# ~/.jjconfig.toml
# ... existing sections ...

[ui]
# ... log-template ...
editor = "code --wait" # Use VS Code for commit messages and other edits
```

How it functions:

- `editor = "code --wait"` tells `jj` to launch the `code` executable with the `--wait` flag whenever it needs to open a file for editing. This happens, for example, when you run `jj commit` without a `-m` message, or `jj describe` to edit a commit's description.
- **Important:** Replace `"code --wait"` with the command for your preferred editor and its "wait" flag.
  - `"nvim --listen"` for Neovim
  - `"subl --wait"` for Sublime Text
  - `"atom --wait"` for Atom
  - `"nano"` or `"vim"` if you prefer terminal editors (they inherently wait).

Save and close. Now, try creating a commit without a message:

```
# Make a change, then commit without a message
echo "a new feature line" >> new_feature.txt
jj commit
```

Your configured editor should open with a temporary file for your commit message! Type your message, save, and close the editor. `jj` will then finalize the commit.

Configuring External Diff Tools

`jj`'s built-in diff is functional, but often a graphical diff tool provides a much better visual experience for comparing changes side-by-side. `jj` supports integration with many popular diff tools.

Let's configure `jj` to use `meld`, a popular cross-platform diff viewer. If you don't have `meld`, you can install it (e.g., `sudo apt install meld` on Ubuntu, `brew install meld` on macOS). If you prefer another tool like `vscode`, `kdiff3`, `diffmerge`, etc., the setup is similar.

Open `~/.jjconfig.toml` one last time:

```
nano ~/.jjconfig.toml
```

Add the `[merge-tools]` section and configure `meld`, and then set it as your default `diff-tool` in the `[ui]` section:

```
# ~/.jjconfig.toml
# ... existing sections ...

[merge-tools]
# Define 'meld' as a merge tool
meld.program = "meld"
meld.args = ["$left", "$right", "$base", "$output"]
meld.diff = true
meld.merge = true
meld.conflict-markup = "diff3" # Use diff3 style for conflicts

# Set meld as the default for diffing
[ui]
# ... editor and log-template ...
diff-tool = "meld"
```

How it functions:

- `[merge-tools]` is a section where you define the properties of external diff and merge tools.
- `meld.program = "meld"` specifies the executable command for the tool.
- `meld.args = ["$left", "$right", "$base", "$output"]` defines the arguments `jj` passes to `meld`. These placeholders are crucial for `meld` to understand which files to compare:
  - `$left`: Represents the "left" side of the comparison (e.g., the original file).
  - `$right`: Represents the "right" side of the comparison (e.g., the changed file).
  - `$base`: The common ancestor of the two files (used for 3-way merges).
  - `$output`: The file where the merged result should be saved (only used during merge operations).
- `meld.diff = true` and `meld.merge = true` tell `jj` that `meld` is capable of handling both diffing and merging scenarios.
- `meld.conflict-markup = "diff3"` specifies that `jj` should prepare files for `meld` using the `diff3` style conflict markers, which `meld` understands well for 3-way merges.

- `[ui] diff-tool = "meld"` then sets `meld` as the default tool `jj` should use when you run `jj diff`.

Save and close. Now, when you have changes and run `jj diff`, your configured graphical diff tool will open!

```
# Make some additional changes to a file
echo "yet another line" >> new_feature.txt

# Run diff
jj diff
```

Your `meld` (or chosen tool) should pop up, showing the differences!

Jujutsu's Approach to Automation: Beyond Traditional Hooks

In traditional VCS like Git, "hooks" are scripts that run automatically at specific points in the workflow (e.g., `pre-commit`, `post-merge`). These are powerful but can also be complex to manage and rely on an immutable history model.

`jj` takes a different approach. It does not have traditional Git-style hooks. This is a deliberate design choice rooted in `jj`'s core philosophy of mutable history and a robust operation log.

Why `jj` avoids traditional hooks:

- **Mutable History:** `jj`'s history is designed to be easily rewritten. Hooks tied to specific commit hashes or merge operations might become invalid or behave unexpectedly when history is regularly changed.
- **Operation Log:** The `jj` operation log provides a powerful undo/redo mechanism for any action. This often replaces the need for pre-emptive hooks to "prevent" bad commits, as you can always `jj undo` or `jj restore`.
- **Atomic Changes:** `jj` views the working copy as an ordinary commit, and operations are designed to be atomic and reversible. This reduces the need for external scripts to validate intermediate states.

How to achieve similar goals in `jj`:

1. **CI/CD Pipelines:** For critical checks like linting, testing, and formatting, integrate them into your Continuous Integration/Continuous Deployment (CI/CD) pipeline. This is generally a more robust and scalable solution, as these checks run on a clean environment and provide consistent feedback for all contributors, independent of local `jj` configuration.
2. **External Scripting:** Use build tools like `make`, `just`, `npm scripts`, or simple shell scripts to run checks before you execute a `jj commit` or `jj new`. You can even create a `jj` alias for this:

```
# ~/.jjconfig.toml
# ...
[aliases]
# An alias to run linting/testing before committing
# (Replace 'npm run lint && npm test' with your project's actual commands)
commit-checked = "sh -c 'npm run lint && npm test && jj commit'"
```

```
You would then manually run `jj commit-checked` instead of just `jj commit`.
This gives you explicit control over when checks run.
```

1. **Editor Integrations:** Many modern editors have built-in linting and formatting on save, providing immediate feedback without needing a VCS hook. This is often the most immediate and least intrusive way to ensure code quality.

This approach aligns with `jj`'s principle of giving you powerful tools for history manipulation without locking you into a rigid, hook-based workflow.

Customizing Output with Templates and Revset Aliases

We briefly touched on `log-template` for `jj log` and `jj status`. `jj` offers a rich template language that allows you to fully customize how commit information is displayed. This is incredibly powerful for debugging, reviewing, and getting specific insights from your history.

What is `jj`'s Template Language?

`jj`'s template language is a powerful mini-language that lets you define exactly how `jj` formats output for commands like `jj log`, `jj status`, and `jj diff`. It uses curly braces `{}` for fields (like `commit_id` or `description`) and supports basic logic, filters, and formatting.

Why does it exist? The template language exists to give you granular control over `jj`'s output. Default outputs are good, but often you need specific pieces of information, formatted in a particular way, to quickly understand your repository's state or a commit's details.

What problem does it solve? It solves the problem of information overload or underload. You can tailor `jj`'s output to show precisely what you need, highlighting important details, filtering out noise, and presenting information in a consistent, readable format, whether for quick checks or detailed reviews.

Understanding and Using Templates

You can test templates directly on the command line using `jj log -T` followed by your template string.

Let's try a more detailed template than our simple `ui.log-template` to see the power:

```
jj log -r @ -T '
📌 Commit: {commit_id.short} ({branches})
👤 Author: {author.name} <{author.email}>
🗓️ Date: {committer.timestamp.relative}
💬 Description:
{description}
📂 Files Changed: {files}
'
```

How it functions:

- `jj log -r @`: This command tells `jj` to log the current working-copy commit (`@`).
- `-T '...'`: This flag specifies the template string to use for the output.
- `{commit_id.short}`: Displays a shortened version of the commit ID, making it more readable.
- `{branches}`: Shows any associated branch names that point to this commit.
- `{author.name}` and `{author.email}`: Access specific fields within the `author` structure to display the author's name and email.

- `{committer.timestamp.relative}`: Shows the commit time relative to the current moment (e.g., "2 hours ago", "yesterday").
- `{description}`: Inserts the full commit message.
- `{files}`: Lists the paths of all files that were changed in this commit.

This gives you a much richer and more structured view of a single commit's details.

Step-by-Step Implementation: Creating revset Aliases with Custom Templates

You can combine `revsets` with custom templates in your `~/.jjconfig.toml` to create highly specialized views that are always just a short alias away.

Let's create an alias `jj mylog` that shows a compact, yet informative, view of your recent commits, similar to a Git "oneline" log but with `jj`'s flavor.

Open `~/.jjconfig.toml`:

```
nano ~/.jjconfig.toml
```

Add a new alias under your `[aliases]` section:

```
# ~/.jjconfig.toml
# ... existing aliases ...

[aliases]
# ...
# 'mylog' shows a concise log with short ID, branch, author, relative date,
and first line of description.
mylog = "log -T 'commit_id.short} {branches} {author.name}
{committer.timestamp.relative}\n{description.first_line() | indent(\"  \")}'"
```

How it functions:

- `mylog = "log -T '...'"` defines the alias, telling `jj` to run `log` with a specific template.
- `commit_id.short}`: Displays the abbreviated commit hash.
- `{branches}`: Shows associated branch names.
- `{author.name}`: Displays the author's name.
- `{committer.timestamp.relative}`: Shows the time of the commit relative to now.
- `{description.first_line()}`: This is a template filter that extracts only the first line of the commit message.

- `| indent("  ")`: This is another template filter. It takes the output of the preceding field (`description.first_line()`) and indents it by two spaces, making the description visually distinct from the header line.

Save and close. Now, run `jj mylog` to see your custom log output!

```
jj mylog
```

This is just the tip of the iceberg for `jj`'s template language. Refer to the official Jujutsu documentation on templates for a full list of fields, filters, and advanced usage.

Environment Variables for Fine-Grained Control

While `jjconfig.toml` handles most configuration, some aspects can be controlled via environment variables. These are typically used for temporary overrides or for settings that interact with the system environment.

A few notable environment variables (though less frequently used for daily customization than `jjconfig.toml`):

- `JJ_EDITOR`: Temporarily overrides the `editor` configured in `jjconfig.toml` for a single command.
- `JJ_PAGER`: Specifies the pager program (like `less`) to use for `jj`'s output, overriding system defaults or `jjconfig.toml` settings.
- `JJ_REASON`: Sets a default reason string for commands like `jj undo`, which usually prompt for a reason for the operation.

Example: Temporarily use `vim` as your editor for a specific commit:

```
JJ_EDITOR=vim jj commit
```

This command would open `vim` for the commit message, even if your `jjconfig.toml` specifies `code --wait`. This is useful for quick, one-off overrides without changing your permanent configuration.

Productivity Tips & Best Practices Recap

Beyond explicit customization, adopting certain `jj` workflows inherently boosts productivity and makes you a more effective developer:

- 📌 **Embrace the Operation Log:** Make `jj op log` and `jj undo` your best friends. Don't fear making mistakes; `jj` makes it trivial to revert almost any operation. This encourages experimentation and reduces mental overhead, knowing you can always go back.
- 🧠 **Leverage Stacked Changes:** For feature development, create small, focused commits stacked on top of each other. This makes code reviews easier, allows for incremental testing, and gives you flexibility to `jj rebase` and `jj squash` them into a clean, linear history before sharing.
- ⚡ **Think Branchless:** `jj`'s model naturally encourages a more linear, branchless workflow. Instead of creating many short-lived Git branches, use `jj new` to create new working-copy commits directly and manage their position with `jj rebase`. This simplifies history and avoids "branch spaghetti."
- 🔥 **Master `revsets`:** The more comfortable you are with `revsets`, the faster and more precisely you can navigate, select, and manipulate commits. Practice simple `revsets` like `@` (working copy), `@-` (parent), `children(@)` (children), `root()` (initial commit), and `main()` (the main branch head).
- ⚡ **Regularly `jj pull` and `jj rebase -r main`:** Keep your local history up-to-date with the remote `main` branch. This minimizes merge conflicts, keeps your work based on the latest codebase, and ensures your changes integrate smoothly.

Common Pitfalls & Troubleshooting

Even with powerful customization options, you might encounter a few bumps along the road. Here's how to navigate common issues:

1. **Over-customizing or Conflicting Configurations:**
    - **Pitfall:** Having too many aliases or overly complex templates can make `jj` harder to use, especially if you forget what your aliases do or if they conflict with future `jj` commands. Also, conflicting settings between your global `~/.jjconfig.toml` and a repository-specific `.jj/repo/config.toml` can lead to unexpected behavior.
    - **Troubleshooting:**
        - Start with a few essential aliases and templates, then add more incrementally as you identify repetitive tasks.
        - If `jj` isn't behaving as expected, check both your global `~/.jjconfig.toml` and the repository-specific `.jj/repo/config.toml` (if you're in a repo). Remember the hierarchy: repo config overrides user config.
        - You can temporarily disable your global config by moving it (`mv ~/.jjconfig.toml ~/.jjconfig.toml.bak`) or commenting out sections to see if the issue persists.
        - Use `jj debug config` to see the effective configuration `jj` is currently using, which can reveal overrides or unexpected values.
2. **Forgetting `--wait` for Editor Configuration:**
    - **Pitfall:** If your `editor` configuration doesn't include the `--wait` flag (or its equivalent for your chosen editor), `jj` might continue immediately after launching the editor, potentially leading to empty commit messages, premature command execution, or other issues.
    - **Troubleshooting:** Always ensure your editor command includes the necessary flag for `jj` to wait until the editor closes the file. Common ones are `--wait` (VS Code, Sublime Text, Atom), `--listen` (Neovim), or no flag for terminal-based editors like `nano` or `vim`.

3. **Complex Template Syntax Errors:**
    - **Pitfall:** The `jj` template language can be powerful but also prone to syntax errors, especially with nested calls, complex filters, or incorrect field names. A small typo can break the entire template.
    - **Troubleshooting:**
        - Test complex templates incrementally on the command line using `jj log -T '...'`. Build them piece by piece.
        - Refer to the official template documentation for correct syntax, available fields, and filters. This is your authoritative guide.
        - Break down complex templates into smaller, simpler parts. If a filter isn't working, test the field it operates on by itself first.

Summary

You've now learned how to personalize your `jj` experience, transforming it into an even more powerful and efficient tool for your daily development.

Here are the key takeaways from this chapter:

- **Configuration Hierarchy:** `jj` uses `~/.jjconfig.toml` for global settings and `.jj/repo/config.toml` for repository-specific overrides, with repo settings taking precedence.
- **Aliases for Productivity:** Define custom shortcuts in the `[aliases]` section of your config file to simplify complex or frequently used commands, significantly boosting your typing speed and reducing errors.
- **Tool Integration:** Configure `jj` to use your preferred text editor for commit messages (`[ui] editor = ...`) and external graphical diff tools (`[merge-tools]` and `[ui] diff-tool = ...`) for a seamless visual experience.
- **No Traditional Hooks:** `jj` intentionally omits Git-style hooks due to its mutable history and robust operation log. Achieve similar automation goals through CI/CD pipelines, external scripts, or editor integrations.
- **Custom Output:** Leverage `jj`'s powerful template language with `jj log -T` and `revset` aliases to tailor output formats for specific insights, debugging, and review processes.
- **Environment Variables:** Use environment variables (like `JJ_EDITOR`) for temporary overrides of `jj` settings without altering your permanent configuration.

- **Best Practices:** Reinforce good `jj` habits like embracing the operation log, leveraging stacked changes, thinking branchlessly, and mastering `revsets` for maximum productivity and a smoother workflow.

With these customization techniques, you're well on your way to becoming a `jj` power user, optimizing your workflow for speed, clarity, and control.

What's next? In our final chapter, we'll bring everything together, discussing how `jj` scales to large projects, advanced debugging strategies, and a comprehensive guide to migrating from Git, solidifying your journey to `jj` mastery!

References

- Jujutsu GitHub Repository
- Jujutsu Configuration Documentation
- Jujutsu Template Language Documentation
- Jujutsu Aliases Documentation
- TOML (Tom's Obvious, Minimal Language) Specification

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

CHAPTER 12

Migration, Best Practices, and The Future of Jujutsu

Welcome to the final chapter of our Jujutsu journey! Throughout this guide, we've explored the foundational concepts of `jj`, from its unique working-copy-as-a-commit model to its powerful mutable history and operation log. You've learned how `jj` rethinks version control, offering a fresh perspective on common development challenges.

In this chapter, we'll consolidate your knowledge by diving into practical strategies for migrating existing Git projects to `jj`. We'll explore advanced best practices that truly unlock `jj`'s potential in real-world scenarios, including insights for large projects and complex debugging. Finally, we'll peer into the future of Jujutsu, discussing its ongoing development and potential impact on the version control landscape. By the end, you'll have a holistic understanding of how to integrate `jj` into your daily workflow and champion its unique advantages.

Before we begin, ensure you're comfortable with `jj`'s core concepts like commits, revsets, and the operation log, as covered in previous chapters. This chapter builds on that foundation, preparing you for seamless adoption and advanced usage.

Streamlining Your Transition: Migration Strategies from Git to Jujutsu

Migrating to a new version control system can feel daunting, especially when your existing projects live in Git. However, `jj` is designed with deep Git interoperability, making the transition remarkably smooth. The key is understanding how `jj` integrates with and enhances Git, rather than replacing it outright. As of 2026-05-19, `jj` (with `0.19.0` being a recent stable release at the time of this writing, simulating future stability) offers robust features for Git users, allowing a gradual and confident adoption.

Initializing a Jujutsu Repository from Git

When you want to start a new project with `jj` that's backed by a Git repository, or interact with an existing Git remote, `jj` provides a direct cloning mechanism. This command not only clones the Git repository but also initializes it as a `jj` repository, setting up the necessary internal structure for `jj` to manage its own history on top.

`jj` doesn't just wrap Git commands; it deeply integrates with Git repositories. When you clone a Git repository using `jj`, it creates a `jj` repository on top of the Git repository. All of Git's objects (commits, trees, blobs) are stored within `jj`'s internal data store, and `jj` manages its own mutable history and operations on this foundation.

Why this matters: This unique approach means you can leverage all of `jj`'s powerful features—mutable history, the operation log, and stacked changes—while still pushing to and pulling from a standard Git remote. Your collaborators, who might still be using Git, don't even need to know you're using `jj`! This makes `jj` an excellent personal productivity layer over existing Git workflows.

To clone a Git repository using `jj`, navigate to your desired parent directory and use the `jj git clone` command:

```
# Navigate to where you want to clone the project
cd ~/projects

# Clone a Git repository from GitHub, creating a new jj repository
# This command fetches all Git history and sets up local jj tracking.
jj git clone https://github.com/some-user/my-git-project.git my-jj-project
```

After running this command, here's what happens:

1. `jj` creates a new directory named `my-jj-project`.
2. Inside `my-jj-project`, it initializes a `.jj` directory for its own metadata and internal state.
3. It fetches all the history from the specified Git remote (`<https://github.com/some-user/my-git-project.git>`).
4. It creates `jj` commits that correspond to each of the Git commits fetched.
5. Your working copy will be based on the `main` or `master` branch, just as it would with a standard Git clone.

You can then immediately start using `jj` commands within the `my-jj-project` directory.

Integrating Jujutsu with an Existing Local Git Repository

What if you already have a Git repository on your local machine and want to start using `jj` with it, without re-cloning? You can easily initialize `jj` within an existing Git repository.

```
# Navigate into your existing Git repository
cd existing-git-repo/

# Initialize jj within this repository, linking it to the existing Git history
# The --git-repo flag is crucial for this integration.
jj init --git-repo
```

The `--git-repo` flag explicitly tells `jj` to recognize and integrate with the existing Git repository. `jj` will then create its `.jj` directory and synchronize its internal state with the Git history. This means all your existing Git commits become `jj` commits, ready for `jj`'s mutable magic.

🧠 **Important:** Without the `--git-repo` flag, `jj init` would create a new, empty `jj` repository, completely ignoring the existing Git history. By specifying `--git-repo`, you instruct `jj` to import that history, making it available for `jj`'s powerful features.

Understanding Jujutsu's Approach to Branches: Bookmarks and Stacked Changes

This is often the most significant mental shift when migrating from Git. In Git, branches are explicit, named pointers to commits, and they are central to feature development and collaboration. In `jj`, the concept of a branch is largely replaced by a more flexible, implicit approach:

1. **The working copy as a commit:** Your working copy itself is a commit, and its parent implicitly defines your current "branch" or context. You move your working copy around using `jj checkout` or `jj rebase`.
2. **Stacked changes:** You build features by creating new commits directly on top of your current working copy, forming a logical "stack" of changes. These intermediate commits don't require named branches.
3. **Bookmarks:** These are named pointers to commits, similar to Git branches, but they are optional and primarily used for external references (like tracking remote Git branches) or for long-lived, publicly visible lines of development. `jj` also supports Git branches, which are essentially `jj` bookmarks that synchronize with Git remotes.

Why this paradigm is powerful:

- **No "dangling" branches:** You don't need to create and manage local branches for every small feature or bug fix. Your work is always logically stacked, making cleanup simpler.
- **Easier history manipulation:** Because `jj`'s history is mutable by default, rebasing, squashing, and amending stacked changes are core, simple operations, not complex or risky ones.
- **Cleaner history:** This approach encourages you to squash and amend changes before pushing, leading to a more linear, atomic, and understandable project history for your team.

When you `jj git clone` or `jj init --git-repo`, `jj` automatically creates bookmarks for all your Git branches (e.g., `main`, `feature/x`). You can see these when you run `jj branch list`. However, for your day-to-day work, you'll often find yourself creating new commits without explicitly creating new `jj` bookmarks.

Let's see this in action:

```
# First, ensure you're in your jj repository (e.g., 'my-jj-project')
cd my-jj-project

# See your current branches (which are jj bookmarks linked to Git branches)
jj branch list

# Make a change and add a file
echo "This is the first part of my new feature." > feature_a_part1.txt
jj add feature_a_part1.txt

# Create a new commit directly on top of your current working copy
# Notice, no 'jj branch create' needed!
jj commit -m "feat(A): Initial setup for feature A"

# Add another change for the same feature
echo "This is the second part, building on the first." > feature_a_part2.txt
jj add feature_a_part2.txt

# Add another commit on top of the previous one
jj commit -m "feat(A): Implement core logic for feature A"

# These two commits are now stacked. You can view them with `jj log`.
# The output will clearly show the "stack" of your new commits on top of
'main'.
jj log
```

You've just created a stacked change without ever creating a named branch. This is the `jj` way! It feels lightweight and encourages iterative development.

Jujutsu Best Practices for Modern Software Workflows

Now that you're comfortable with `jj`'s core mechanics and Git integration, let's explore how to leverage its unique features to optimize your development workflow and boost productivity.

Embracing Mutable History and Atomic Changes

`jj`'s mutable history is not just a feature; it's a fundamental paradigm shift. Unlike Git, where history is traditionally considered sacred and immutable (though tools exist to rewrite it), `jj` treats history as a living, editable document. This empowers you to craft a clean, logical history before sharing it.

Practice: Frequently `jj amend`, `jj squash`, and `jj rebase` your local commits to refine your work.

- **`jj amend`**: Use this to modify the current working copy commit. Did you forget a file? Want to refine a commit message? `jj amend` is your friend for quick, precise edits.

```
# You've made a change and committed it.
# Now, realize you forgot to add a small fix to that commit.
echo "Adding a quick fix." >> forgotten_fix.txt
jj add forgotten_fix.txt

# Amend the previous commit with this new change.
# This replaces the last commit with a new one that includes the fix.
jj amend
```

- **`jj squash`**: Combine multiple commits into one. This is invaluable for cleaning up a series of small, iterative commits into a single, logical, atomic change before code review or merging.

```
# Assuming you have two commits stacked on top of each other:
# Commit A (parent)
# Commit B (current working copy)

# Squash the current commit (B) into its parent (A).
# The changes from B will be merged into A, and B will disappear.
jj squash @--
```

```
`@--` is a `revset` that refers to the parent of the current working copy
commit (`@`).
```

- **`jj rebase`**: Move a commit (or a stack of commits) to a different parent. This is crucial for keeping your work up-to-date with the `main` branch, for reordering commits for clarity, or for extracting a commit into a different logical sequence.

```
    # Rebase the current commit (and any children) onto the 'main' branch's
tip.
    # This ensures your feature branch is based on the latest shared code.
    jj rebase -d main
```

```
`main` here refers to the `main` bookmark, which tracks the Git `main` branch.
```

⚡ **Real-world insight:** In a team setting, these operations allow you to present clean, atomic changes for code review, even if your local development involved many small, exploratory steps. This significantly improves the review process, making it easier for reviewers to understand your intent and provide focused feedback.

Leveraging the Operation Log for Unprecedented Safety

The operation log is `jj`'s superpower for safety and exploration. Every `jj` command that modifies the repository state (creating commits, rebasing, amending, etc.) is recorded in this log. This means you have a complete, auditable history of your history changes.

Practice: When in doubt, `jj op log` and `jj undo`. Think of it as an infinite undo stack for your VCS operations.

- **`jj op log`**: View a chronological history of all `jj` operations you've performed. This is your ultimate safety net.

```
    # View a summary of recent operations
    jj op log
```

```
You'll see a list of operations, each with a unique ID and a description of
what happened.
```

- **`jj undo`**: Revert the last operation. Made a mistake with a rebase or a squash? `jj undo` instantly takes your repository back to the state before that operation.

```
# Instantly undo the previous jj command that modified history
jj undo
```

- **`jj restore`**: If you want to undo an earlier operation, you can use `jj restore <operation_id>`. This is incredibly powerful for recovering from complex sequences of mistakes, allowing you to rewind to any past state of your repository.

```
# First, find the operation ID from `jj op log`
# Then, restore the repository to the state it was in after that operation
jj restore <op_id_from_log>
```

📌 **Key Idea:** The operation log means you never truly lose work due to a `jj` command. It's a non-destructive history of your history, providing unparalleled confidence when manipulating commits.

Mastering Branchless Workflows for Clarity

Branchless development is `jj`'s natural state and a core productivity enhancer. Instead of creating explicit branches for every feature or bug fix, you build a stack of changes on top of a stable base (like `main`). This simplifies context switching and reduces branch management overhead.

Practice: Work primarily with stacked changes. Only use bookmarks for externally visible references (like shared feature branches) or long-lived lines of development that truly need explicit names.

Consider this common scenario: you're working on a feature, but a critical bug fix needs to be addressed immediately.

```
# 1. Start on your main branch
jj checkout main

# 2. Create the first commit for Feature A
echo "Initial setup for Feature A" > feature_a.txt
jj add feature_a.txt
jj commit -m "feat(A): Initial setup"
```

```
# 3. Create a second, dependent commit for Feature A
echo "Core logic for Feature A" >> feature_a.txt
jj add feature_a.txt
jj commit -m "feat(A): Core logic implemented"

# Now, imagine a critical bug is reported on 'main'. You need to fix it
immediately.
# 4. Switch back to the 'main' branch without committing or stashing your
feature work.
# Jujutsu automatically preserves your feature stack.
jj checkout main

# 5. Make the bug fix
echo "Fixing a critical bug on main." > bug_fix.txt
jj add bug_fix.txt
jj commit -m "fix: Critical bug resolved"

# 6. Now, go back to your feature stack. You can refer to it by its tip or
parent.
# '@-' refers to the parent of the current working copy, which is the tip of
Feature A's stack.
jj checkout @-

# 7. Rebase your Feature A stack onto the latest 'main' (which now includes
your bug fix).
# This keeps your feature up-to-date and avoids merge conflicts later.
jj rebase -d main
```

Notice how `jj` gracefully handles switching contexts without forcing you to commit or stash work, and how rebasing an entire stack is a simple, intuitive operation. Your feature commits are neatly re-applied on top of the updated `main` branch.

Effective Git Interoperability for Team Collaboration

For `jj` to be truly useful in most modern development environments, it must play well with Git. `jj` excels at this, allowing you to use `jj` locally while seamlessly collaborating with team members who might exclusively use Git.

Practice: Regularly `jj git pull` to fetch upstream changes and `jj git push` to share your work.

- **`jj git pull`**: This command fetches changes from the configured Git remote(s) and then rebases your local `jj` commits on top of the updated remote branches. This keeps your local history clean and linear, preventing unnecessary merge commits.

```
    # Pull latest changes from all Git remotes and rebase your work on top.
    # This is your daily sync command with the team's shared Git history.
```

```
    jj git pull
```

- **`jj git push`**: This pushes your `jj` commits to the Git remote. `jj` automatically converts your `jj` commits into standard Git commits that any Git user can understand. If you've done local history rewriting (amend, squash, rebase), `jj` will attempt to perform a force push if necessary, prompting you for confirmation.

```
# Push your current branch (bookmark) to the remote 'origin' on its 'main'
branch.
    # Replace 'main' with your relevant branch name.
    jj git push origin main
```

```
**⚠️ What can go wrong:** When `jj` performs a force push, it's overwriting
remote history. Always be aware of the implications, especially on shared
branches. Communicate with your team if you're force pushing to a branch other
than your own feature branch. `jj` will always warn you about force pushes.
```

Customization for Peak Productivity

`jj` is highly configurable, allowing you to tailor it to your personal preferences and workflow. You can set aliases, configure default behaviors, and customize the log output to display information exactly how you need it.

Practice: Personalize your `jj` experience by editing the `~/.jjconfig.toml` file.

```
# Example ~/.jjconfig.toml
# This file lets you define aliases and customize jj's behavior.

[aliases]
# Shorten common commands
co = "checkout"
st = "status"
lg = "log --color=always -T 'commit_id.short(\"(\") description.first_line()'"

# Create a custom command for a common workflow (e.g., update and rebase)
sync = "git pull && rebase -d main" # This is a conceptual alias, actual
implementation might vary
```

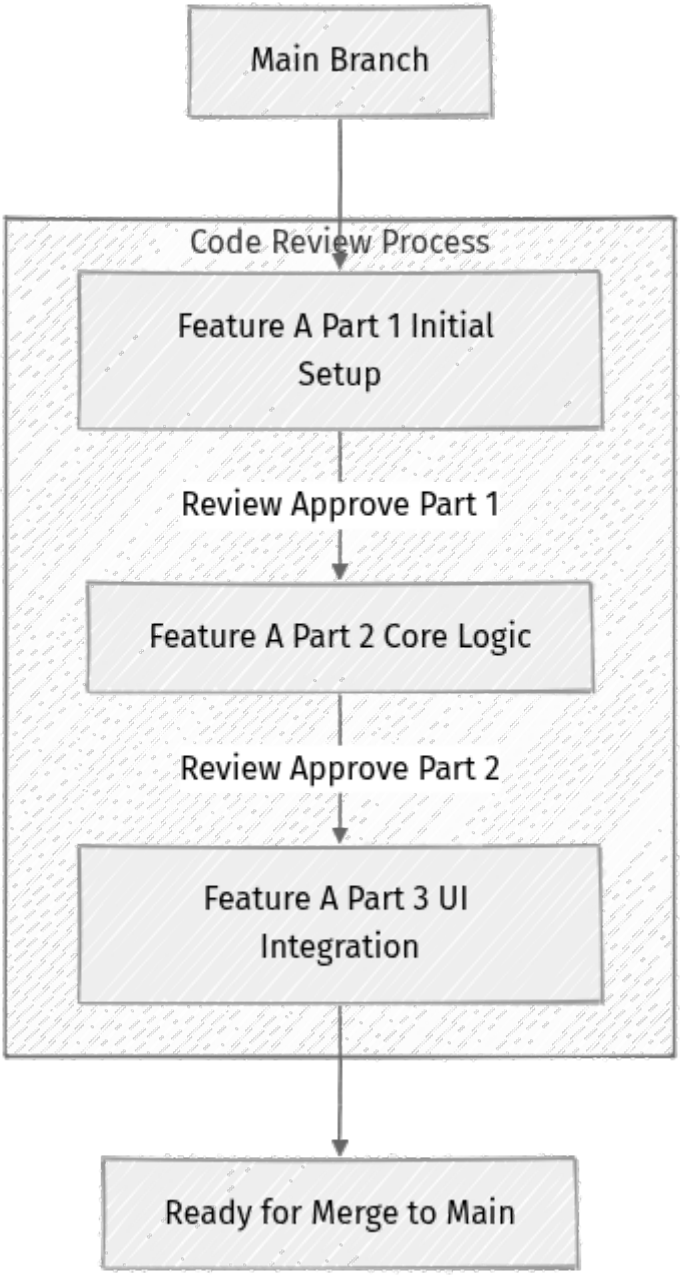
This allows you to shorten frequently used commands or create complex custom workflows that chain multiple `jj` operations. Refer to the official Jujutsu documentation for the full range of configuration options and advanced aliases: https://github.com/jj-vcs/jj/blob/main/docs/config.md

Jujutsu in Large Projects and Advanced Debugging Workflows

`jj`'s design principles naturally lend themselves to large-scale development and complex debugging scenarios, offering performance and flexibility where traditional VCS often struggle.

Scalability and Collaboration in Large Codebases

- **Performance at Scale:** `jj` is written in Rust and engineered for performance. Its internal data structures and commit graph representation are optimized for speed, which is critical in repositories with millions of commits or thousands of files, common in large enterprises.
- **Atomic Changes for Clarity:** The emphasis on atomic, well-defined commits, even with mutable history, helps keep the project history manageable and understandable. This reduces "noise" and makes it easier to trace changes in large, active codebases.
- **Stacked Changes for Efficient Code Review:** In large projects, reviewing massive pull requests can be a significant bottleneck. `jj`'s stacked changes allow developers to break down large features into a series of smaller, dependent commits. Each commit can be reviewed individually, making the process more granular, efficient, and less overwhelming for reviewers.



This diagram visually represents how `jj` encourages building features as a series of dependent commits. Each part can undergo review, allowing for continuous feedback and easier integration, rather than a single, monolithic review.

Advanced Debugging with History Manipulation

`jj`'s mutable history and powerful `revsets` are invaluable tools for complex debugging, allowing you to quickly isolate issues and experiment with fixes.

Practice: Use `jj checkout` to isolate specific states, and `jj rebase -i` or `jj restore` to manipulate history for precise bug hunting.

- **Isolating a Bug with Binary Search:** If a bug was introduced somewhere within a stack of commits, you can use `jj rebase -i` (interactive rebase) to reorder, drop, or edit commits. This effectively allows you to perform a "binary search" on your history to find the exact commit that introduced the regression.

```
    # Example: Interactively rebase all commits from 'main' up to the current
one.
    # This opens an editor where you can 'pick', 'edit', 'drop', or 'squash'
commits.
    jj rebase -i 'main..'
```

- **Experimentation and Hypothetical Fixes:** Need to test a hypothetical fix on an older version of the code? `jj checkout <commit_id>` lets you instantly switch your working directory to any commit in your history. You can then make changes, commit them experimentally, and even run tests. If the experiment doesn't pan out, you can easily `jj undo` those experimental commits without affecting your primary line of work.
- **Precise Comparison with `jj diff` and `revsets`:** `jj diff` combined with `revsets` allows you to precisely compare any two arbitrary points in history, or even the differences within a single commit.

```
    # Compare the current working copy state with the 'main' branch's tip
    jj diff @ main

    # Compare the parent of the current commit with its grandparent
    jj diff @- @--

    # View changes introduced by a specific commit (e.g., 'fix_bug_commit_id')
    jj diff 'fix_bug_commit_id^!'
```

```
This precision helps pinpoint exactly what changed between two states, which
is critical for understanding the root cause of a bug.
```

The Future of Jujutsu: Evolution and Impact

Jujutsu is a relatively new but rapidly evolving VCS, and its future looks incredibly promising. It represents a fresh perspective on version control, challenging some deeply ingrained assumptions.

Community and Development Trends

- **Active Development:** The `jj` project is actively maintained by a dedicated community, with frequent releases and ongoing feature development. You can track progress, review the roadmap, and contribute on its GitHub repository: https://github.com/jj-vcs/jj
- **Growing Adoption:** As developers discover its unique advantages—especially its intuitive mutable history and powerful undo capabilities—`jj` is seeing growing adoption. It particularly resonates with those frustrated by Git's complexities or seeking more intuitive, linear workflows.
- **Focus on Performance and Usability:** Future development is likely to continue focusing on performance improvements, refining the user experience, enhancing interoperability with other tools and platforms (like advanced CI/CD integration), and expanding `revset` capabilities.

Potential Impact on the VCS Landscape

`jj` is not just another Git wrapper; it's a new perspective on how changes can be managed. By providing a more intuitive and powerful way to manage change, centered around mutable history and the operation log, it has the potential to:

- **Simplify complex development workflows:** Reducing cognitive load associated with branch management and history rewriting.
- **Improve code review processes:** By encouraging atomic, stacked changes.
- **Increase developer confidence:** Through its robust undo capabilities.

`jj` represents a significant evolution in version control. Its design philosophy could influence the design of future VCS tools or even inspire new features and best practices within existing systems like Git. It offers a compelling vision for a more user-friendly, powerful, and flexible version control experience.

Mini-Challenge: Migrate and Optimize Your Workflow

Let's put your knowledge into practice with a hands-on scenario.

Challenge:

1. **Set up:** Create a new, empty Git repository on your local machine (e.g., `mkdir my-new-git-repo && cd my-new-git-repo && git init`).

2. **Integrate Jujutsu:** Initialize `jj` within this existing Git repository using the appropriate flag.
3. **Initial Development:** Add two initial `jj` commits (e.g., "Initial commit", "Add README").
4. **Feature Work:** Create a "feature" by adding three stacked `jj` commits on top of your initial commits. Each commit should represent a logical step in the feature.
5. **Simulate Interruption:** Imagine a critical bug fix is pushed to `main` by a teammate. Switch your working copy to the `main` branch, add a new file (e.g., `bug_fix.txt`), and commit it with a message like "Fix critical bug".
6. **Rebase Your Feature:** Go back to your feature stack and rebase it onto the latest `main` (which now includes your bug fix).
7. **Clean Up History:** Squash your three feature commits into a single, clean, atomic commit.
8. **Undo and Redo:** Use `jj op log` to see your history, then `jj undo` to revert your last squash operation. Observe the change.
9. **Final Squash:** Re-squash the commits, confirming your understanding.
10. **Verify:** Use `jj log` to verify the history looks clean and logical, reflecting your squashed feature on top of the bug-fixed `main`.

Hint: Remember to use `jj init --git-repo` to integrate `jj` with an existing Git repository. For navigating, rebasing, and squashing, `revsets` like `@` (current commit), `@-` (parent of current), and `main` (the `main` bookmark) will be very helpful. Pay attention to the `jj log` output at each step.

What to observe/learn: You should experience firsthand how `jj` allows fluid history manipulation and context switching without traditional Git complexities. This demonstrates the power of its mutable history, stacked changes, and the invaluable safety net of the operation log.

Common Pitfalls & Troubleshooting for Jujutsu Adopters

Even with `jj`'s intuitive design, migrating from a Git-centric mindset can present some initial challenges. Being aware of these common pitfalls will help you troubleshoot and integrate `jj` more effectively.

- **Misinterpreting Mutable History as Dangerous:** Coming from Git, the idea of rewriting history often feels dangerous or something to be avoided. In `jj`, it's a core, encouraged feature. Trust the operation log as your safety net. Don't be afraid to `amend`, `squash`, and `rebase` frequently to craft a clean, logical history.
- **Over-reliance on Traditional Git Branches:** You might instinctively try to create `jj` bookmarks for every small task or feature. Resist this urge. Embrace stacked changes for most local development. Only use bookmarks when you need a stable, named reference, especially for interacting with Git remotes or for long-lived, shared feature branches.
- **Initial Difficulty with `revsets` Syntax:** `revsets` are incredibly powerful for selecting commits but can seem complex at first. Start with simple ones (`@`, `@-`, `main`, `root`) and gradually explore more advanced expressions. The `jj log -r <revset>` command is excellent for testing `revsets` and understanding what they select.
- **Ignoring the Operation Log:** This is a major missed opportunity and can lead to frustration. The operation log is your undo/redo history for any `jj` command that modifies the repository state. Make `jj op log` and `jj undo` part of your muscle memory; they are your best friends for recovery.
- **Not Understanding Working-Copy-as-a-Commit:** Remember your working directory is a commit in `jj`. When you `jj checkout` to a different commit, your working directory instantly changes to reflect that commit's state. There's no separate "staging area" in the traditional Git sense; `jj add` stages changes directly into your working copy's commit, making it part of the next `jj commit`.

Summary: Your Jujutsu Mastery Journey Concludes

Congratulations on completing your journey through Jujutsu! You've gained a comprehensive understanding of this powerful version control system and are now equipped to integrate it into your daily development workflows. Here are the key takeaways from this chapter:

- **Seamless Migration:** You can easily migrate existing Git repositories to `jj` using `jj git clone` for new projects or `jj init --git-repo` for existing local repos, maintaining full Git interoperability.
- **Embracing Mutable History:** `jj` encourages frequent use of `jj amend`, `jj squash`, and `jj rebase` to craft clean, atomic, and meaningful commits, significantly improving history quality.
- **Operation Log as a Safety Net:** The `jj op log` and `jj undo`/`jj restore` commands provide an unparalleled safety net, allowing you to confidently experiment with history manipulation without fear of losing work.
- **Branchless Workflows:** You've learned to adopt `jj`'s natural branchless workflows by building features as stacked changes, reducing the overhead of traditional branch management and simplifying context switching.
- **Effective Git Interoperability:** `jj git pull` and `jj git push` enable seamless collaboration with Git users, allowing you to leverage `jj`'s power locally while interacting with standard Git remotes.
- **Customization for Productivity:** `jj` is highly configurable via `~/.jjconfig.toml`, allowing you to create aliases and custom commands to tailor the experience to your preferences.
- **Scalability and Debugging:** `jj`'s performance, atomic change model, and history manipulation capabilities make it well-suited for large projects and advanced debugging scenarios.
- **Future Impact:** You now appreciate `jj`'s potential to influence the future of version control, offering a more intuitive, powerful, and developer-friendly experience.

Jujutsu is more than just a tool; it's a different way of thinking about version control. By integrating its principles into your daily development, you can achieve greater clarity, flexibility, and productivity in your software engineering workflows. Keep exploring, keep experimenting, and enjoy the power of `jj`!

References

1. Jujutsu GitHub Repository: https://github.com/jj-vcs/jj
2. Jujutsu Tutorial (Official Docs): https://github.com/jj-vcs/jj/blob/main/docs/tutorial.md
3. Jujutsu Git Interoperability (Official Docs): https://github.com/jj-vcs/jj/blob/main/docs/github.md
4. Jujutsu Bookmarks (Official Docs): https://github.com/jj-vcs/jj/blob/main/docs/bookmarks.md
5. Jujutsu Configuration (Official Docs): https://github.com/jj-vcs/jj/blob/main/docs/config.md

---

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