# Git as a Story

## A Version Control Journey for Everyone

# 🧭 Why Version Control Matters

You already use version control informally:

- `report_final_v3_reallyfinal.xlsx`
- Multiple presentation drafts
- Document copies "just in case"

**Sound familiar?**

## The Questions We Ask

When someone hands you a file, you want to know:

- **Who** made this?
- **Where** did the data come from?
- **Why** did they make these changes?
- **How** did it change over time?
- Does it have the **latest** updates?

## What You Really Want

Like a lab notebook for your work:

- **Undo** mistakes
- **Go back in time** to see what changed
- **Audit trail** of edits
- **See which experiments** were run
- **Understand** what worked before

**Git was designed to serve this need.**

## Building from Scratch

Let's design this system together.

**Simple approach:** Make a copy every time we like our work

- `report_v1` → `report_v2` → `report_v3`

Works ok for single files. But what about projects?
`report_v1` also depends on other files (images, spreadsheet, charts, other code files)

- It can get unmanageable quickly

# Building from Scratch

**Option 2:**

Google Docs and Office 365 save automatically...

**But what if:**

- You step away mid-edit
- The snapshot catches a broken state
- You didn't mean to save yet

# The Best of Both Worlds

**Option 3:**

Let's create a **COMMIT BUTTON** that:

- Takes a snapshot when **you** decide
- Saves the **entire folder**, not just one file
- Lets you add a **message** describing the change
- Doesn't create file explosion (`v1`, `v2`, `v3`...)

**Why save the whole folder?**

Otherwise: `report_v20` + `spreadsheet_v3` + `image_v6` = chaos!

# 📸 Commit: Your Time Machine Control

Click COMMIT to:

1. Save a snapshot of your work
2. Write a message: "What changed and why?"
3. Create a known good state

Now you have a reliable history to fall back on.

## 2. Going Backwards and Forwards

If every commit has a message describing what changed...

**We need a LOG BUTTON** to see the list of changes over time.

**We need a CHECKOUT BUTTON** to jump to any point in history.

## The Bookshelf Analogy

Every commit is like a book on a shelf:

- Each book is a complete snapshot of your work at that moment
- Want to see how things looked last month? Pick up that book
- **Checkout** = checking out a book from the library

## 3. The Problem with Linear History

Real work isn't linear:

- Scientists test multiple hypotheses
- Analysts try different formulas
- Writers explore alternate endings

**You need to experiment without breaking what works.**

## Real Example 1: The Dashboard Analyst

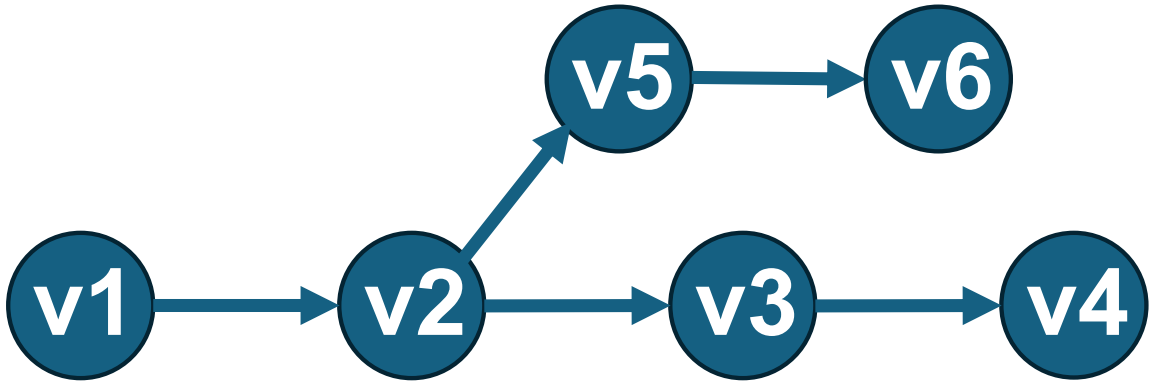
You're building a biomedical research dashboard:

- You have a **better formula** to try
- Want to **test it** without switching everyone
- Can't **stop working** while people test
- Still need the **current version** running

**Solution: We need branches.**

# Branches and Trees: The Problem with Linear History

2. Fixing bugs in a previously deployed version



## Merging: Bringing Ideas Together

When an experiment succeeds, you can merge it back.

Git finds the most recent common ancestor,
compares differences, and integrates the changes.

## Hashes and Integrity

Each commit has a unique *digital fingerprint* (hash).
Changing any content changes all future hashes.

## The Staging Area

Select which changes to include before committing.
Like adding items to a cart before checkout.

## Tags: Marking Milestones

Want to mark special versions?
Tags are bookmarks in history.

## Git as a Cognitive Model

**Snapshots** – freeze moments in time

**Branches** – explore ideas in parallel

**Merges** – bring ideas together coherently

# End

“Every version tells a story.”

Git just helps you remember the whole plot.