Building Better Data Science Workflows

*Core Practices with **Git**, **GitHub**, and **Data Version Control (DVC)** for Effective Collaboration & Development*

Ryan Paul Lafler

Miguel Angel Bravo

Premier Analytics Consulting, LLC

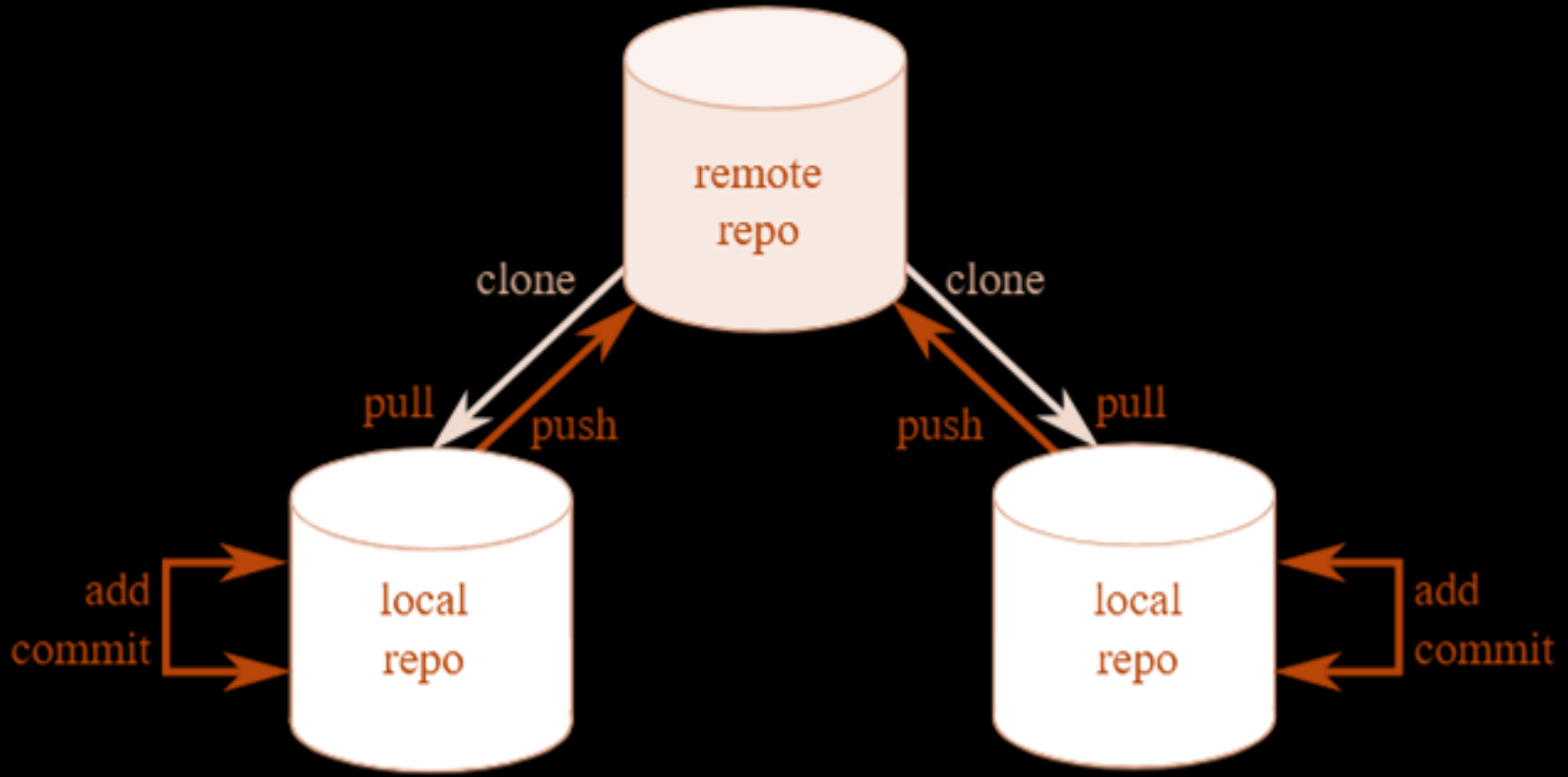
Premier Analytics Consulting, LLC

Premier Analytics Consulting, LLC

1. Tools for Managing Code, Versioning Software, & Tracking Data

Git & GitHub Fundamentals: *What*, *Why*, and *How*

- **Git** is a tool for tracking changes to code repositories in a distributed manner
- **GitHub** is a cloud-based platform for hosting remote repos for collaboration



Git & GitHub Fundamentals: *What*, *Why*, and *How*

- Managing codebases requires both Git & GitHub:
  - Track, modify, version, & manage project histories on local & remote repos
  - Implement new features, fix bugs without impacting primary codebase
  - Implement controls on branches, review & authorize changes to code
- Git enables parallel development of code with independent local repos
- GitHub enables open-source collaboration with Pull Requests & more

Version Control, Team Development, & Repositories

- Version control keeps a history of changes & enables rollbacks to previous (stable) versions in organized manner

- **Main Advantage:** Version control prevents merge conflicts with multiple developers

    - Git is **distributed version control** where every developer has their own local repo

Version Control, Team Development, & Repositories

- Repository (repo) is a storage location for code & version history
  - **Local Repo:** Stored on each developer's system
  - **Remote Repo**: Stored in available & central location (i.e., cloud, server)
- GitHub enhances repos by allowing **public** & **private** repos

But What About Data? Where Git Needs Support…

- **Git is designed for code (in text / Markdown files)**
  - Struggles with binary file formats → inefficient when handling large file because it replicates entire file (does not just do *deltas*)
  - Every version of binary file is stored completely as a new file
- **Data Version Control (DVC) rests on top of Git → allows file versioning**
  - Large data stored in external (separate) data storage from repo
  - Stores hash pointers in Git, actual data is kept in separate storage center
  - Data is not duplicated → different versions managed in DVC cache

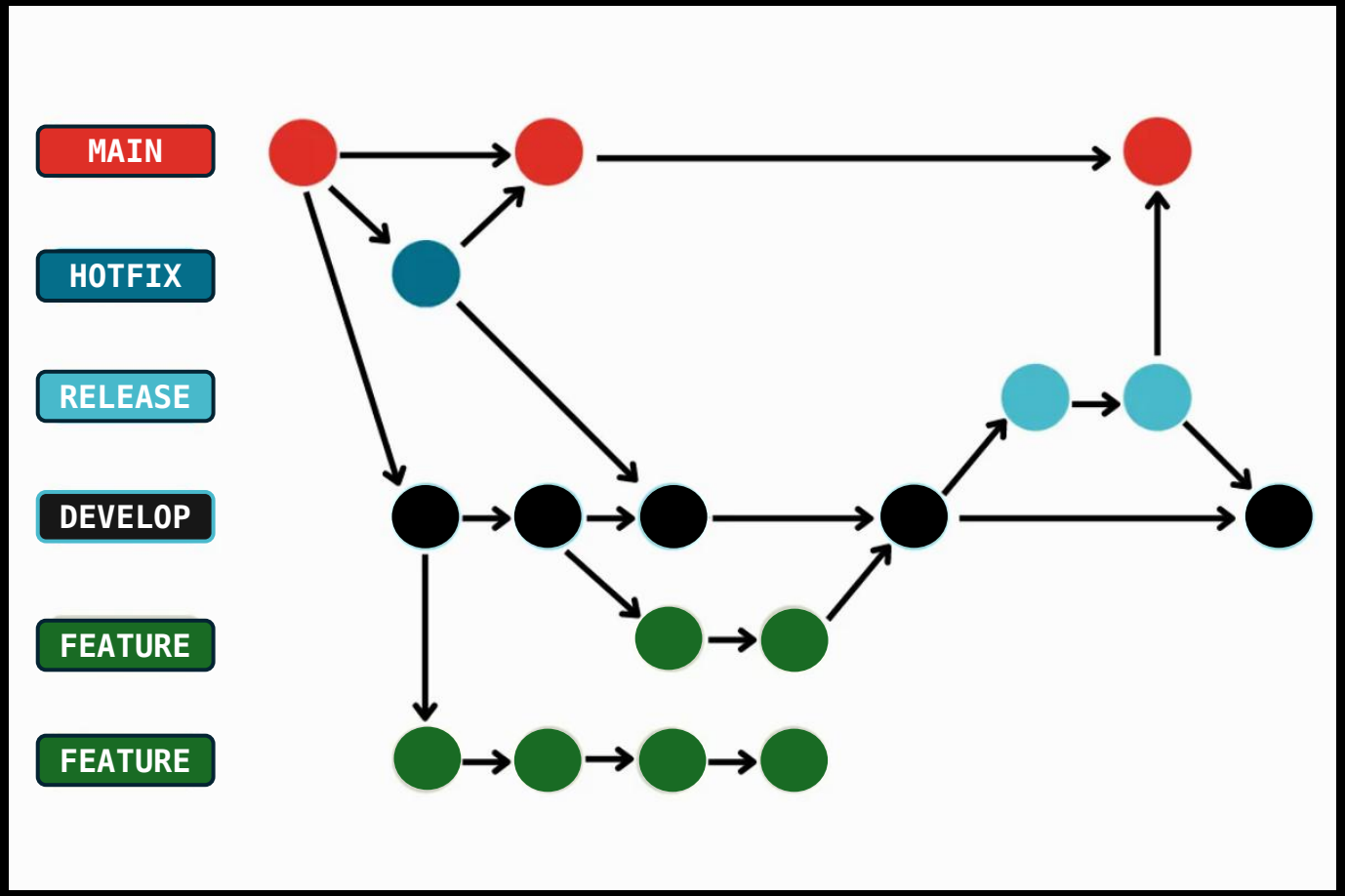
Premier Analytics Consulting, LLC

2. Growing Healthy Development Workflows by Supporting Strong Branches

git

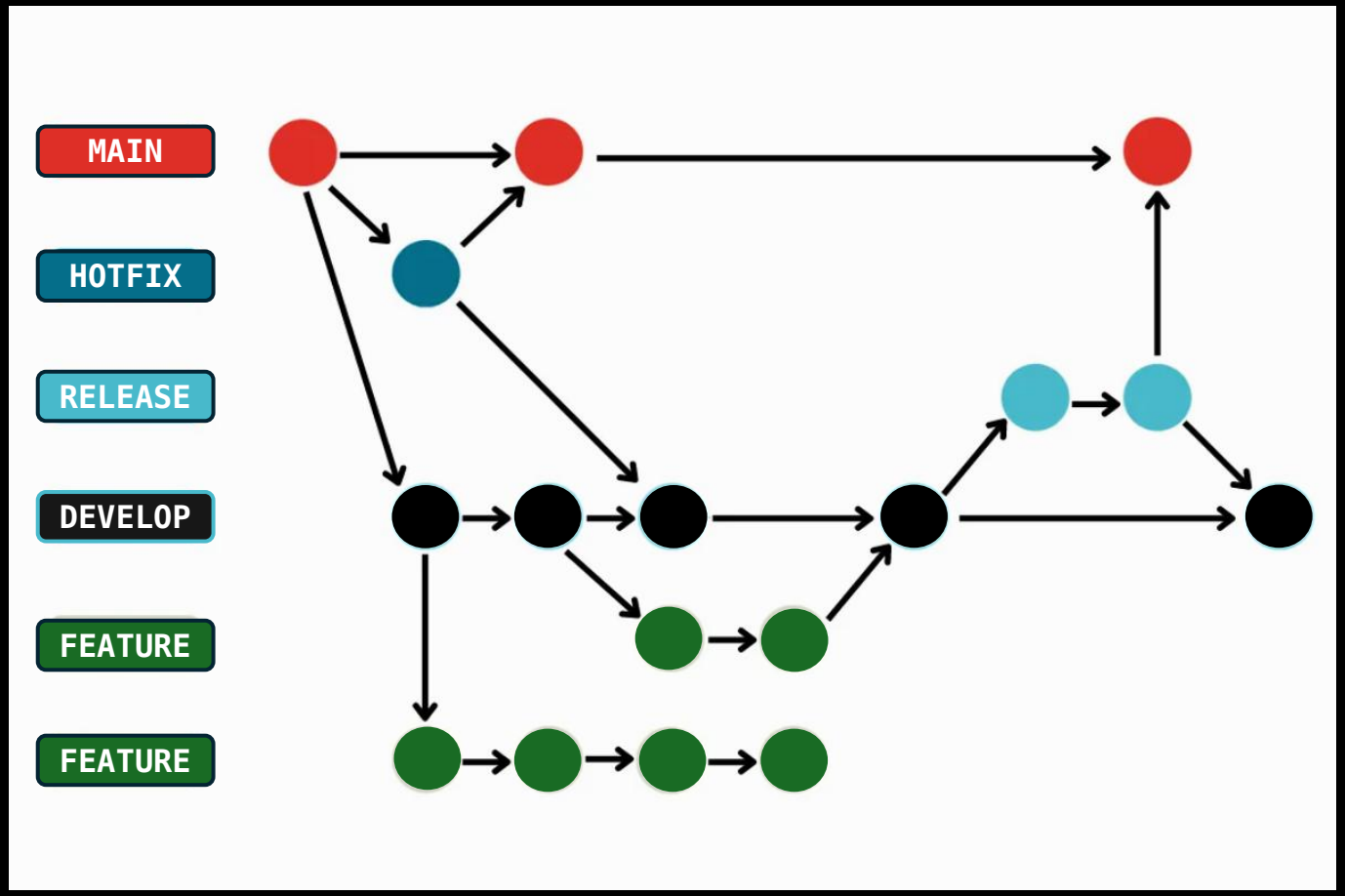
DVC

Branching in Git → Independent Workflows



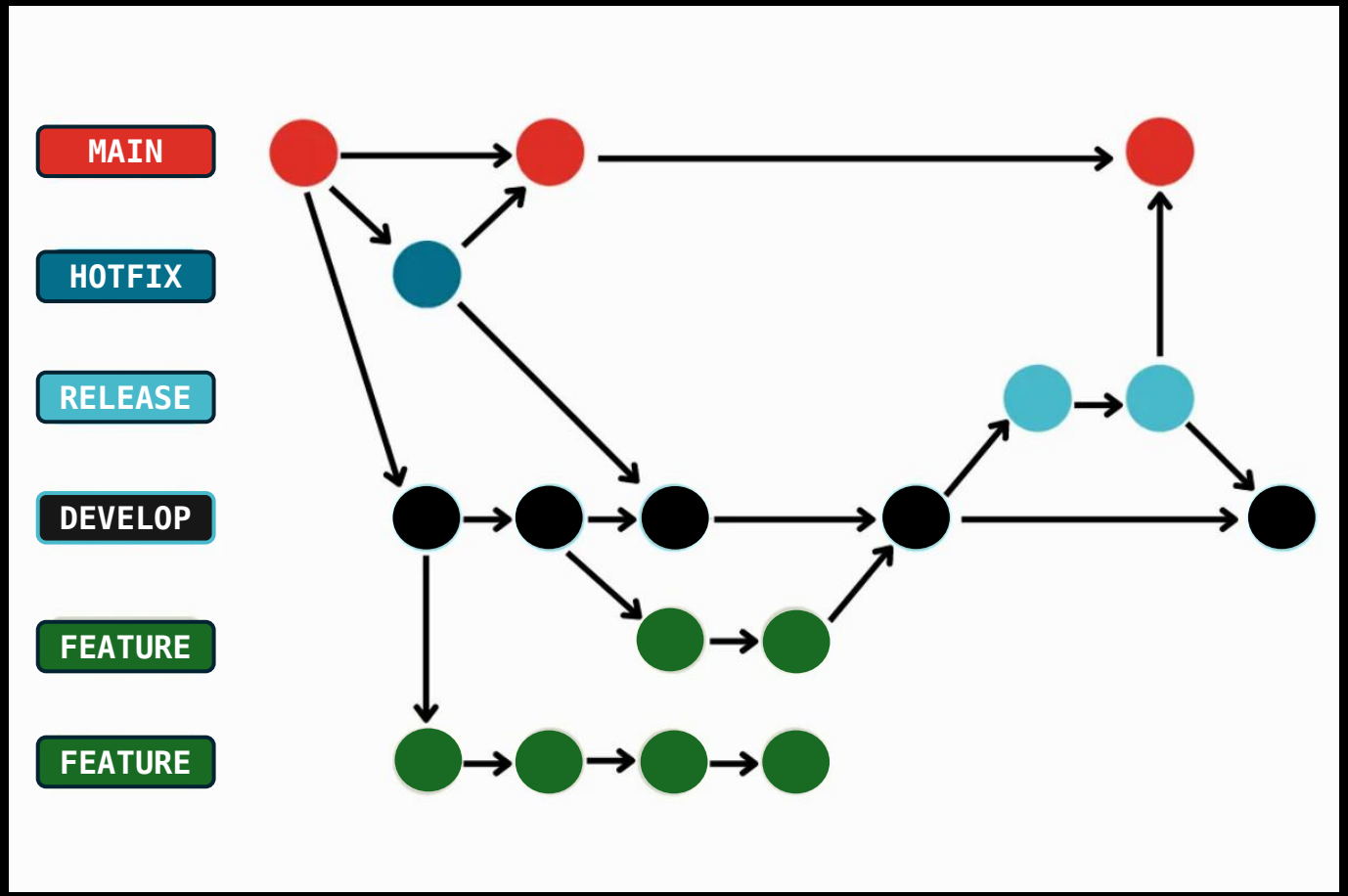
- Branches are pointers to specific commits
- Every repo has `main` branch → new branches spawn from other branches
- Each branch moves forward independently as new commits are made to it

Branching in Git → About the Branch Workflow



- **`main`** represents primary (stable) codebase
- **`hotfix`** is created from **`main`** → used for quick fixes; merges into **`main`** and **`develop`**
- **`release`** is created from **`develop`** → used for final testing; merges into **`main`** and **`develop`**

Branching in Git → About the Branch Workflow



- **`feature`** branches always originate from develop
- Each **`feature`** branch developed separately & merged into **`develop`**
- **`feature`** can have sub-branches too to prevent merge conflicts
  - Called **Task Branching**

Premier Analytics Consulting, LLC

3. Enhancing Your Git & GitHub Workflows with Tips and Tricks

#1 Manage New Features, Releases, Hot Fixes, & Bug Fixes with Branches and Sub-Branches

Tip #1

- Use branches and sub-branches to describe:
    1. **General item under development** → *Feature*, *Bugfix*, *Hotfix*, *Release*.
    2. **More specific (focused) work** → *login-page, ML-model, db-fix*.
- Less conflicts between developers → work is divided into manageable functions
- Can merge sub-branches together into parent branch
- Promotes parallel development & organizes large features into smaller parts
- Reduces potential merge conflicts → developers on assigned function branch
- Branches protect primary codebase (`main`)

Tip #1 | Example

We need to fix feature on login page of app not scaling as expected across different devices.

1. Must create a new branch from **`develop`** branch
2. Change relevant CSS & JS files
3. Add files to tracking
4. Commit changes to local sub-branch
5. Switch back to **`develop`** branch
6. Merge sub-branch with **`develop`** branch

Tip #1 | Example

```
git checkout -b bugfix/login-scaling develop

git add *

git commit -m “fix: resolved login page scaling on diff devices”

git checkout develop

git merge --no-ff bugfix/login-scaling
```

Creates new branch from develop; adds files to tracking.

Commits bugfix to local repo; switches back to develop branch.

Merges bugfix branch into develop branch (maintains merge history).

#2 Commit with Purpose for Meaningful Changes Using Prefixed-Messages

Tip #2

- Don’t wait until features are fully developed to commit
  - **Counterpoint:** Don’t commit every small change for tracking
- Commit on **local repo** and **stash** those commits *only* on the current branch
- Never make commits directly on `main`, `development`, or `release` branches
  - Only writes commits to branch (or sub-branch) currently worked on
- Keep commits atomic → each commit should serve a single purpose

Tip #2

- Use Meaningful Messages with Simple Prefixes:
  - **`WIP`**: Work in Progress
  - **`feat`**: New feature
  - **`fix`**: Bug fix
  - **`hotfix`**: Immediate (emergency) fix
  - **`refactor`**: Restructure / optimize code chunks
- Makes searching, squashing, & dropping commits easier when using:

```
git rebase -i
```

Tip #2 | Example

We need to issue `WIP` and `fix` commits to fix the responsive scaling issue with out login page. View the commit history containing `WIP` and `fix` messages.

1. Switch to **`bugfix/login-scaling`** branch from **`develop`** branch
2. Track changes to file fixing responsive issue
3. Create meaningful commits
4. View commit history

Tip #2 | Example

```
git checkout -b bugfix/login-scaling develop

git add login.css login.html

git commit -m “WIP: adjust flexbox layout for better scaling”

git add login.css

git commit -m “WIP: refine media queries for mobile viewing”

git log --oneline
```

Generates a series of `WIP` commits after meaningful changes are implemented on local repo. Nothing is pushed to remote repo yet. When meaningful change is done on code, commit is made locally.

#3 Squash `WIP` Commits Together *Before* Pushing to Remote Repo → Maintain a Cleaner Version History

Tip #3

- `WIP` commit messages can make version tracking difficult to manage
  - `WIP` does not add meaningful info to version history
- Since commits are only made on local repo, **squash** WIP commits together to maintain a clean (and linear) version history
- Ensures every commit pushed to remote repo reflects a completed, functional change
- **Use WIP commits locally, but squash before sharing to remote repo**

Tip #3 | Example

We need to squash our multiple `WIP` commits into a single, completed `fix` commit and then *safely* push to Git and GitHub.

1. Retrieve the previous `WIP` commits in Git
2. Interactively edit previous `WIP` commits and *squash* together
3. Save changes to the commit history and exit
4. Create new message at top of rebased commit history, then save, & exit
5. Safely push rebased commit history to remote repo

Tip #3 | Example

```
git rebase -i HEAD~2
```

```
pick abc1234 WIP: refine media queries for mobile viewing

pick def5678 WIP: adjust flexbox layout for better scaling
```

```
pick abc1234 WIP: refine media queries for mobile viewing

squash def5678 WIP: adjust flexbox layout for better scaling
```

Squash previous `WIP` with newest commit. **`HEAD~2`** retrieves the 2 most recent commits → manually type **`squash`** to condense commits together.

Tip #3 | Example

After saving and closing commit history window, a **new interactive window** opens & prompts user to type in new commit representing final feature of all squashed WIP commits. Then safely push history to remote repo.

```
# Type at top of interactive window

fix: corrected issue with responsiveness on mobile devices
```

```
# Safely push re-written commit history to remote repo

git push --force-with-lease origin bugfix/login-scaling
```

#4 (Alternative) Stash WIP Changes on Local Repo and then Create and Push a Final Commit to Remote Repo

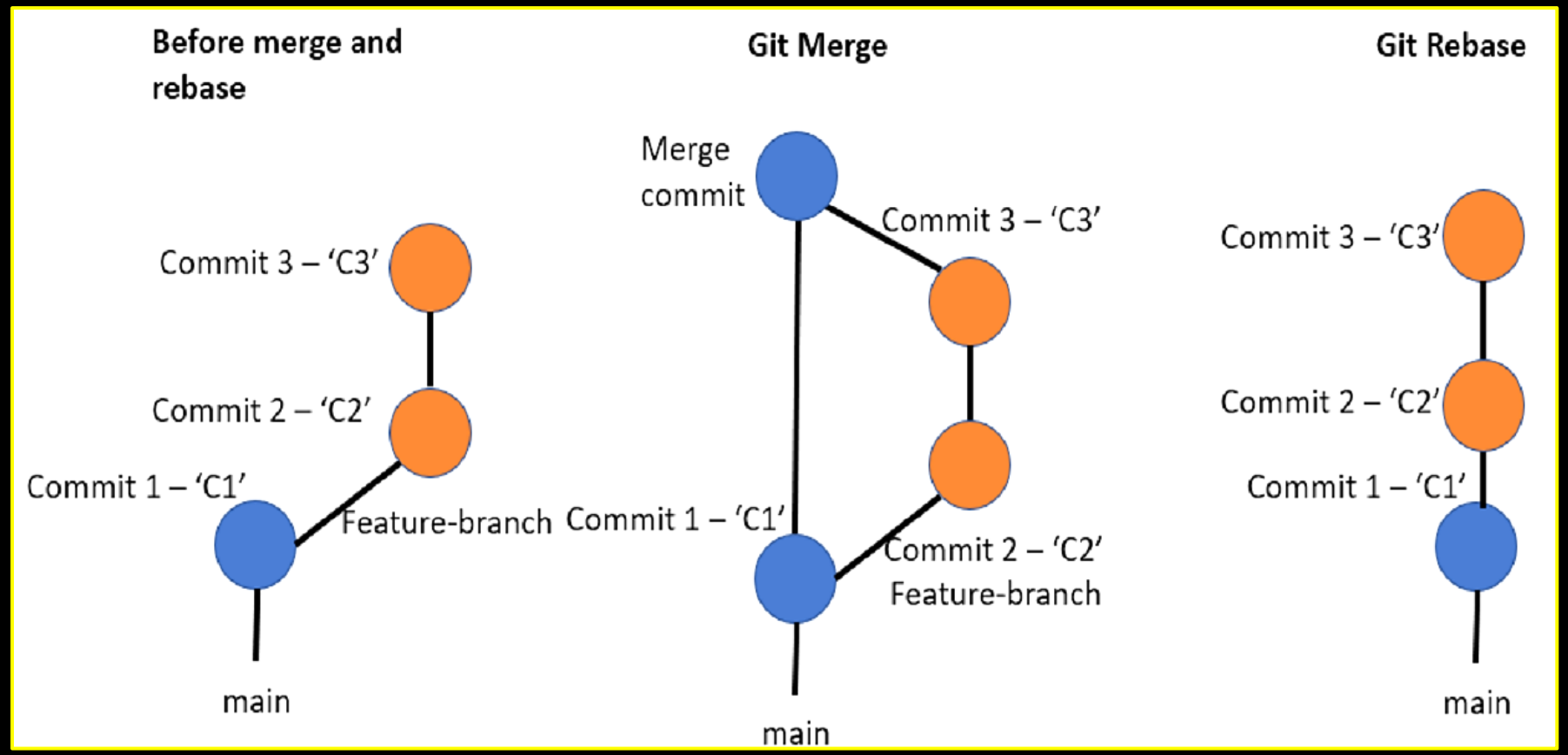
Tip #4

- Instead of committing to local repo, let's save our work in temporary storage so we do not interfere with version history for `WIP` commits
- Helps avoid messy rebase situations → no need to squash commits before pushing to remote repo
- Allows experimentation of work on local system without affect commit history
- Clean git history → no more `WIP` cluttering log (or needing to be squashed)
- Stashes can be deleted, removed, and applied to one final commit
    - Can safely push a final commit to remote repo and reducing merge conflicts

#5 `Git Merge` vs. `Git Rebase`

→ Preserving all History vs. Rewriting History to be More “Clean”

Tip #5



- **`git merge`**: Preserves all merge commits & histories between public branches (protects collaboration by creating merge commits)
- **`git rebase`**: Re-writes commit history to be linear → good when working on **private** feature branches (bad idea when working on public branches)

Tip #5

- `git rebase` is good when:
  - Cleaning up commits before sharing work
  - Maintaining a linear, easily-searchable commit history
  - Incorporating latest changes from **develop** branch into **local** feature branch
- `git merge` is good when:
  - Merging feature branch into **develop** branch → preserves all histories
  - Working on a team-shared branch in remote repo
- Rebasing on a shared branch **rewrites history for everyone: CAUTION!**

Tip #5 | Solution

- Always create independent, local feature branches for individual work → never work directly on shared branches
  - Commit **locally** to feature branch and use **rebase** to handle `WIP` messages
  - Retrieve latest updates from remote **`develop`** before rebasing local (private) feature branch into **`develop`**
- **Optimal workflow** → Fetch latest updates from remote **`develop`** branch, rebase local feature branch *on top of* local **`develop`** branch
  - When ready to merge local **`develop`** with remote **`develop`** branches, use **`git merge`** to preserve histories & then push changes to remote repo

Tip #5 | Example

```
git checkout -b bugfix/login-scaling develop

git add login.css login.html

git commit -m “fix: corrected responsive issue on devices”
```

Go to local independent branch spawned from develop branch.

Adjust code, add, and track adjusted files to the commit.

Commit a meaningful change → with prefix & message to local repo branch.

Tip #5 | Example

```
git checkout bugfix/login-scaling

git fetch origin

git rebase develop
```

Switch to local independent branch.

Fetch latest updates (only differences) from remote repo with shared branches.

Take differences from shared **`develop`** branch & add them on top of local branch **`bugfix/login-scaling`** using **`rebase`**

All local commits from feature branch applied on top of latest **`develop`** branch.

Tip #5 | Example

```
git checkout develop

git merge bugfix/login-scaling

git push origin develop
```

Switch to updated local **`develop`** branch (synced with remote branch).

Merge to save all histories between shared **`develop`** and local feature branches.

This successfully merges local feature branch into updated **`develop`** branch.

Safely sends updated **`develop`** branch to remote repo with bugfix applied.

Premier Analytics Consulting, LLC

4. Tracking Data Histories: Managing Data Stores in Different Storage Mediums

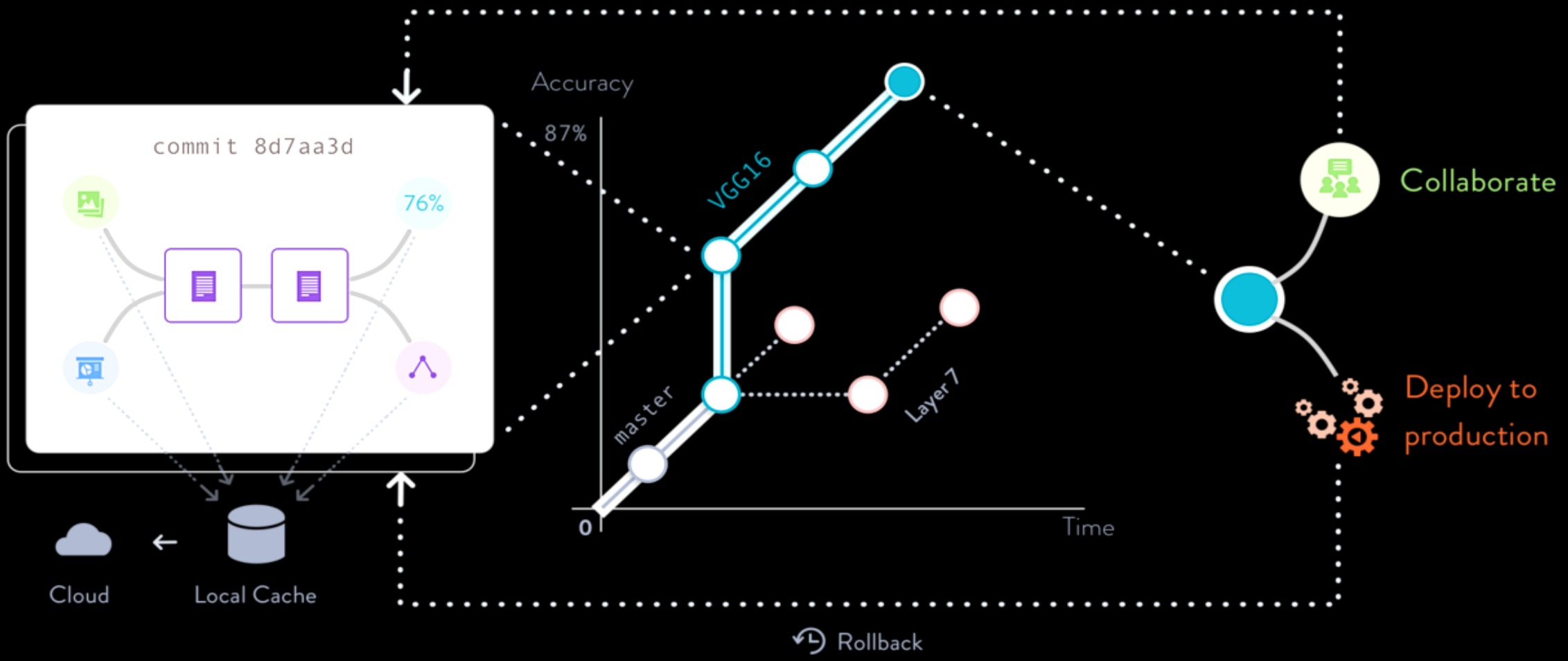
The Problem with Data Tracking & Management

- **Have you ever had any of the following problems with data management?**
  - Multiple versions of the same dataset (dataset_final.csv, dataset_v2_final_OK.csv, etc.) ?
  - Not knowing which dataset was used for a previous ML model ?
  - Large datasets that won't fit in GitHub or emails ?
  - Reproducibility issues: "My code works on my machine, but not on yours."
- DVC can handle *any* data file, format, & structure → versatile & powerful
  - Git only knows how to operate on Markdown & Text files efficiently

What is DVC and Why Do We Need It?

- DVC lets you track dataset versions without overloading GitHub with large files
- DVC does NOT store data in Git: it stores metadata in Git while keeping data in external storage
  - Different versions of data stored as hash objects (string identifiers)
  - Git likewise uses hash to identify commits
- Simple to implement with any local Git repository → DVC added on top

What is DVC and Why Do We Need It?



Example: Versioning Data in an ML Project

```
mkdir real_estate_project && cd real_estate_project
git init
dvc init
git commit -m "Initialized Git and DVC for real estate project"
```

```
mkdir data
echo "house_id,price,sqft,location" > data/house_prices.csv
echo "1,250000,1600,San Francisco" >> data/house_prices.csv
echo "2,320000,2000,New York" >> data/house_prices.csv
# (Imagine this file contains 10,000 rows)
```

```
dvc add data/house_prices.csv
```

Output

```
100% Add...
Creating 'data/house_prices.csv.dvc'
```

Example: Versioning Data in an ML Project

```
git add data/house_prices.csv.dvc .gitignore
git commit -m "Version 1: Initial dataset with 10,000 house sales"
dvc push
```

```
# (Appending new rows to the dataset)
echo "10001,275000,1800,Los Angeles" >> data/house_prices.csv
echo "10002,500000,2500,Chicago" >> data/house_prices.csv
# (Imagine this adds up to 15,000 rows total)
```

```
dvc add data/house_prices.csv
```

Output

```
100% Add...
Updating 'data/house_prices.csv.dvc'
```

Example: Versioning Data in an ML Project

```
dvc diff
```

Output:

```
Path                    Size        Changes
data/house_prices.csv   3.5MB       +5000 rows added
```

```
git log --oneline
```

Output:

```
9f3a2bc (HEAD -> main) Version 2: Added 5,000 new house sales records
1d2e456 Version 1: Initial dataset with 10,000 house sales
```

```
git checkout 1d2e456
dvc checkout
```

Output:

```
Restoring 'data/house_prices.csv' to previous version...
```

Real-World Applications of DVC

1. Finance (Fraud Detection)

- Banks version transaction datasets and train models with different versions to detect fraud patterns

2. Healthcare (Medical Image Diagnosis with AI)

- Hospitals track changes in labeled medical images to see how corrections and new data impact cancer detection models

3. Tech Companies (Personalized Recommendations)

- Companies like Netflix and Spotify version user behavior datasets to test how data changes affect recommendation models

Thank You for Attending!

Questions?

Ryan Paul Lafler rplafler@premier-analytics.com

Miguel Angel Bravo mabravo@premier-analytics.com