# Cherry-Picking

ADVANCED GIT

**Amanda Crawford-Adamo**
Software and Data Engineer

# What is cherry-pick?

Applies the changes from a specific commit to another branch

### Cherry-Pick Single Commit

```
git cherry-pick <commit-hash>
```

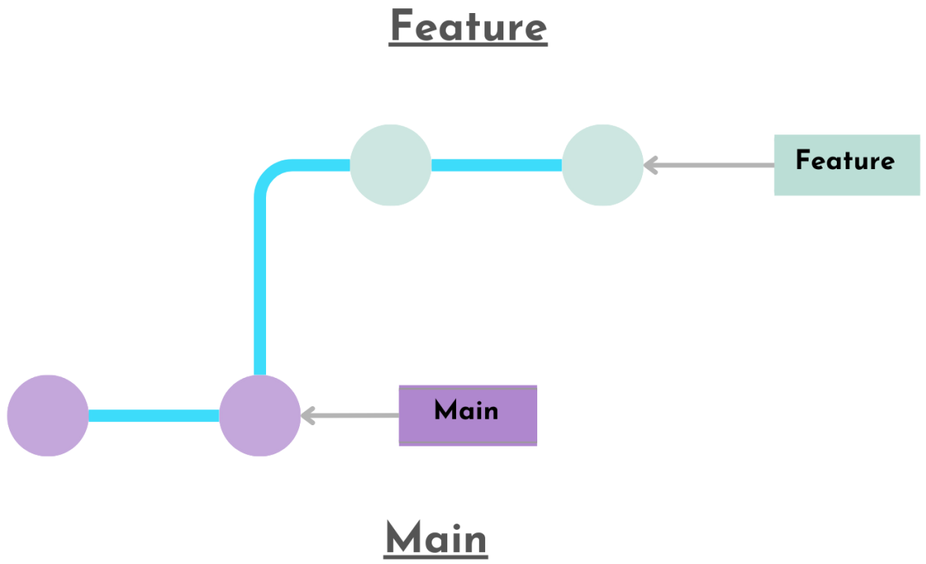
### Cherry-Pick Multiple Commits

```
git cherry-pick <hash1> <hash2> ..
```

### Purpose

- Apply specific bug fixes across branches
- Roll back features to stable versions
- Selectively apply experimental changes
- Recover lost commits

# Cherry-Pick example




Main

```
Data Team

$ git log --oneline main
jkl2345 (HEAD -> main) Update data schema
mno6789 Initial pipeline setup
```

Feature

```
Data Team

$ git log --oneline feature
abc1234 (feature) Add advanced data validation
def5678 Implement error logging
ghi9101 Update data schema
mno6789 Initial pipeline setup
```

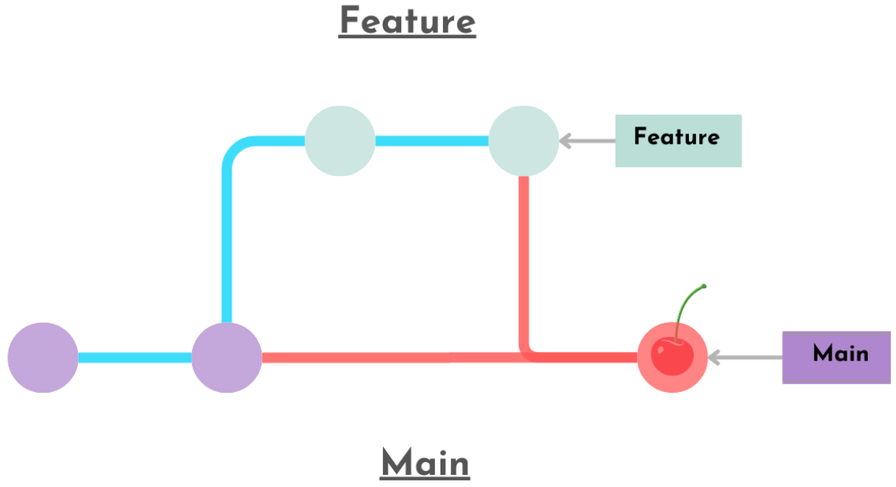
# Cherry-Pick process

1. Checkout main branch

```
git checkout main
```

2. Run cherry-pick command to bring in commit `def456` changes from feature branch.

```
git cherry-pick def456
```




Data Team

```
$ git log --oneline main
pqr0123 (HEAD -> main) Implement error logging
jkl2345 Update data schema
mno6789 Initial pipeline setup
```

# Resolving cherry-pick conflicts

## Conflict resolution steps

1. Manually edit conflicting files
2. Add resolved files to staging using `git add <resolved-files>`
3. Continue with the cherry-pick process by running `--continue` flag

```
git cherry-pick --continue
```

## Stopping a cherry-pick

To stop a cherry-pick operation, use the `--abort` flag

```
git cherry-pick --abort
```

# When to use cherry-pick

## Use cases

1. Applying hotfixes
2. Testing isolated features

## Cautions

1. Can create duplicate commits
2. May complicate project history if overused
3. Larger changes: consider merging or rebasing

# Let's practice!

ADVANCED GIT

# Bisect

ADVANCED GIT

**Amanda Crawford-Adamo**
Software and Data Engineer

datacamp

# What is git bisect?

A tool that uses binary search to find the commit that introduced a bug

### Git Bisect Command

```
git bisect
```

### Purpose

1. Find the bad commit fast
2. Essential for data debugging
3. Speeds up root cause analysis

# Bisect - start

1. Initiate git bisect session

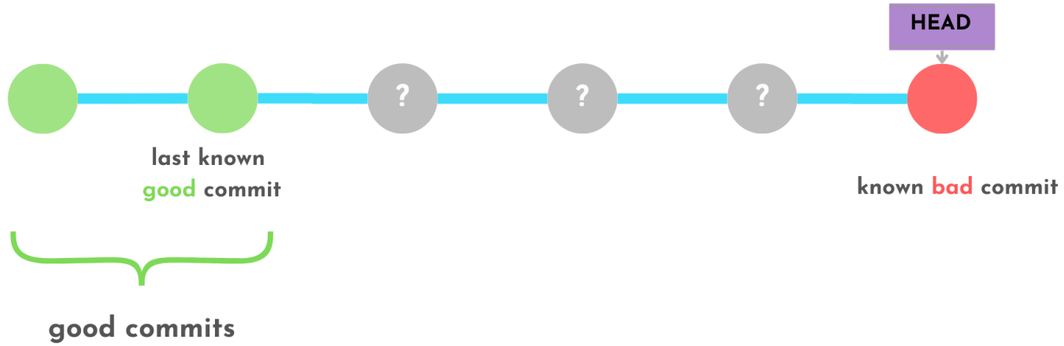
```
git bisect start
```

2. Initialize the current state as a bad state

```
git bisect bad
```

3. Mark the last known good state

```
git bisect good <commit-hash>
```



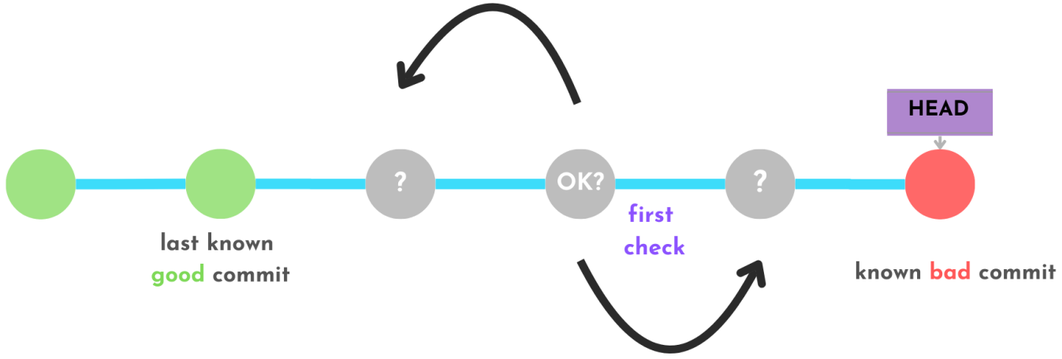
# Bisect - search

1. Marks the commit state as a bad commit

```
git bisect bad
```

2. Marks the commit state as a good commit

```
git bisect good
```

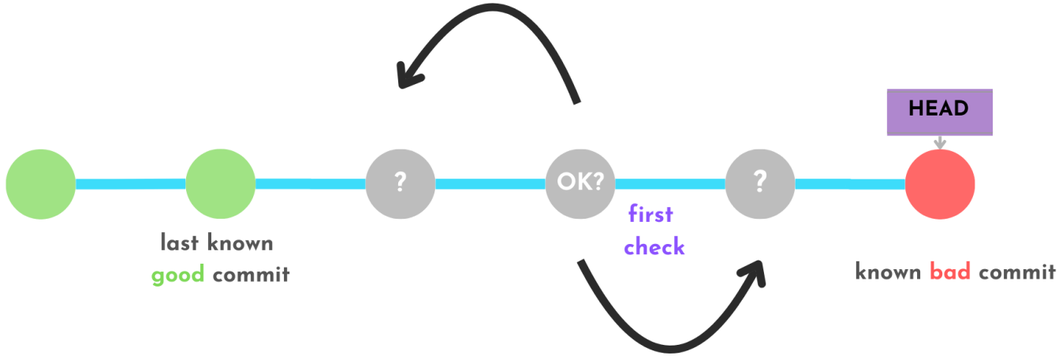


# Bisect - automated search

Checks if the commit version is good or bad by running an automated test script.

```
git bisect run <script_name>
```

- The script must return 0 if the tests passed.
- If the tests fail, it should return a non-zero number.



# Bisect - result

### Git Bisect Output Example

```
$ git log
b1a534f is the first bad commit
commit b1a534f89l2c3d4e5f6g7h8i9j0k1l2m3n4o5p
Author: Jane Doe <jane@example.com>
Date:   Thu Mar 14 14:30:00 2024 -0500

Update data transformation logic
```

Exits the git bisection process and return to our current HEAD

```
git bisect reset
```

# When to use git bisect

## Use cases

1. Find regressions in data workflows
2. Use test scripts for faster debugging

## Tips

- Automate testing with `git bisect run <test-script>`
- Use descriptive commit messages to aid the process

# Let's practice!

ADVANCED GIT

# Git Filter Repo

ADVANCED GIT

**Amanda Crawford-Adamo**
Software and Data Engineer

datacamp

# What is git filter-repo?

### Git Filter-Repo Command

```
git filter-repo
```

A tool for rewriting Git repository history quickly and safely.

- Rename files or directories
- Operates on all branches simultaneously

### Purposes

1. Remove sensitive data (e.g., passwords, tokens)
2. Clean up unnecessary files
3. Restructure repositories
4. Reduce repository size

# Filter-Repo process

1. Install git filter-repo using pip

```
pip install git-filter-repo
```

2. Remove secrets.txt from every commit

```
git filter-repo --path secrets.txt --invert-paths
```

### Filter-Repo Related Filters

`--path` : specifies which paths to operate on

`--invert-paths` : operate on all paths except the ones specified in `--path`

# Filter-Repo result

## Output

```
Parsed 150 commits
New history written in 0.10 seconds; now repacking/cleaning...
Repacking your repo and cleaning out old unneeded objects
```

## Key Implications

- All branches and commits were updated
- All commit hashes were changed
- A force push is needed after this step
- Team members will need to clone repo again

# When to use filter-repo

## Use cases

- Removing sensitive data (e.g., passwords)
- Cleaning up bloated repositories
- Renaming or reorganizing files across all commits

## Tips

- Always back up your repository before using filter-repo
- Coordinate with collaborators before pushing rewritten history

# Let's practice!

ADVANCED GIT

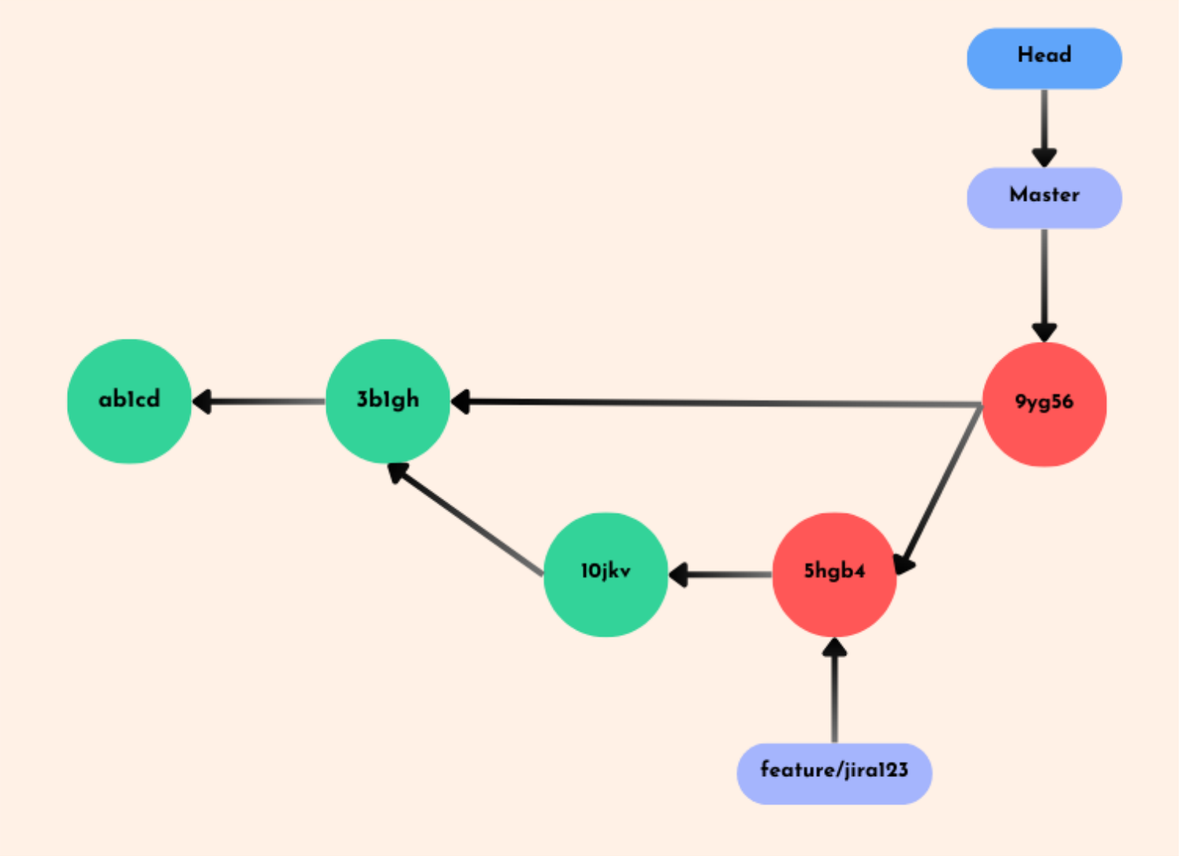
# Git reflog

ADVANCED GIT

**Amanda Crawford-Adamo**
Software and Data Engineer

# What is Git Reflog?

1. Local record of **ALL** reference updates in our repository
2. Reflog is stored in on our local system under the `.git/logs/refs/heads/` directory
3. Records changes to branch tips and HEAD position
4. Acts a safeguard for our Git operations
5. Helps recover accidental changes or deletions



# Git Reflog versus Git Log

| Feature | Git Reflog | Git Log |
|---|---|---|
| Purpose | Shows reference updates and rewrites in local repo | Shows commit history only |
| Scope | Local repository only | Local and remote repositories |
| Content | All ref updates (commits, resets, merges, etc.) | Only commits |
| Persistence | Temporary (usually 90 days) | Permanent (part of repository history) |
| Use Case | Recovering lost commits, understanding recent actions | Viewing project history, understanding feature development |

# Reflog Commands

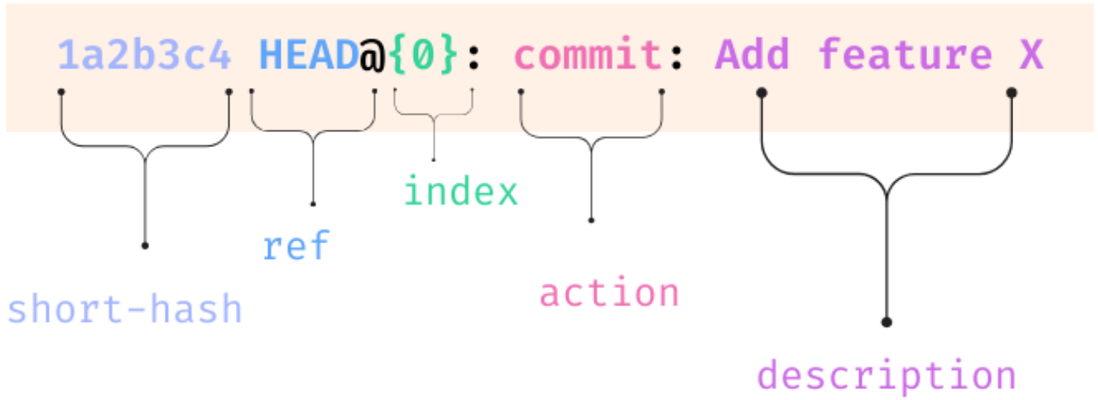
- Displays the log data and HEAD activity

```
git reflog
git reflog show
```

- Clean up old log or unreachable entries

```
git reflog expire
```

# Reflog Output Structure




| Component | Description |
|---|---|
| short-hash | Abbreviated commit hash |
| ref | Usually HEAD, but can be branch names |
| index | Position in the reflog (0 is the most recent) |
| action | Type of action (commit, reset, merge, etc) |
| descriptions | Description about the action |

```
# View HEAD history
git reflog show

0a2e358 HEAD@{0}: reset: moving to HEAD~2
0254ea7 HEAD@{1}: checkout: moving from 2.2 to main
c10f740 HEAD@{2}: checkout: moving from main to 2.2
```

[1] https://hackernoon.com/time-to-rewrite-your-git-history-effectively-with-git-reflog

# Filtering: Since and Until

- **since** = show entries from this point in time

```
git reflog --since "time-qualifer"
```

- **until** = show entries up to this point in time

```
git reflog --until "time-qualifer"
```

## Usage

```
git reflog --since="1 week ago"
git reflog --until="yesterday"
git reflog --until="2024-01-01"
```

# Recovering Deleted Branches

## Scenario

- Created a branch called `etl-feature`
- Committed ETL feature changes to this branch
- We accidentally deleted this branch
- All code changes were lost

**How can we restore the `etl-feature` branch?**

## Solution

1. Identify the **hash** of the commit at the tip of deleted branch using **git reflog**
2. Use git **checkout** to move HEAD to the commit hash of the deleted branch
3. Create a **new branch** using the commit HEAD is currently pointed

```
git reflog
git checkout <hash>
git switch -c <branch-name>
```

[1] https://stackoverflow.com/questions/3640764/can-i-recover-a-branch-after-its-deletion-in-git

# Git Reset

- Moves the HEAD to the specific commit object.
- Depending on the reset type, the working and staging area is updated.

| Reset Type | Command | Effect on Working Directory | Effect on Staging Area |
|---|---|---|---|
| Soft | `git reset --soft <commit>` | No changes | Changes remain staged |
| Mixed (Default) | `git reset --mixed <commit>` | No changes | Changes are unstaged |
| Hard | `git reset --hard <commit>` | Changes are discarded | Changes are discarded |

# Finding A Loss Commit

## Scenario

- Tests started failing after commits and rebases
- Unknown commit caused the failure
- Need to revert to a passing state

**How do we revert back to the previous ETL script changes?**

## Solution

1. Find the deleted commit hash using Git Reflog
2. Use git reset to revert to the commit with the passing tests

```
git reflog
git reset --soft HEAD@{1}
```

# Best Practices

- Powerful for recovering lost code and commits
- Use descriptive commit messages
- Push to remote regularly
- Be cautious with force pushes

*Reflog is our local time machine when we make mistakes*

# Let's practice!

ADVANCED GIT