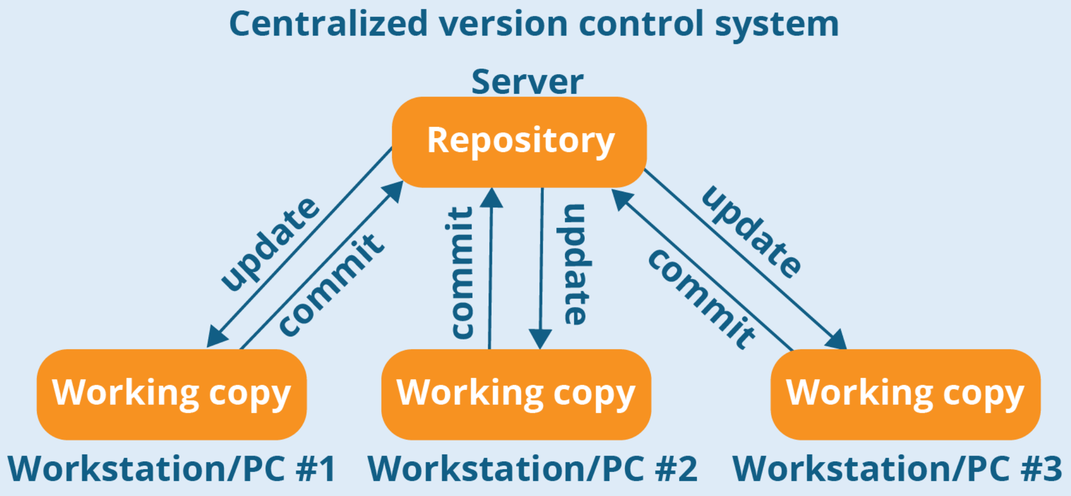
Git Workshop

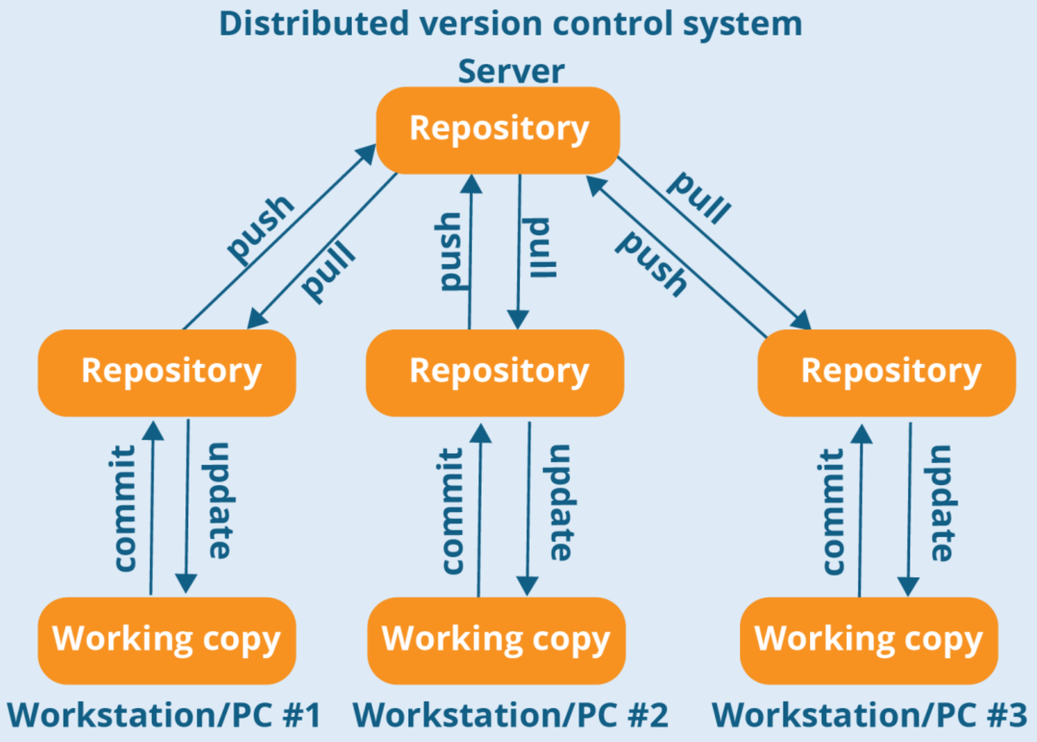
- Motivation
- Version control systems
- Git concepts
- First Git repository
- Remote/origin
- Collaborative coding

Version Control Systems

- Key tool for all coding, and much more
- Necessary for all collaborative coding
- Integral to all software engineering



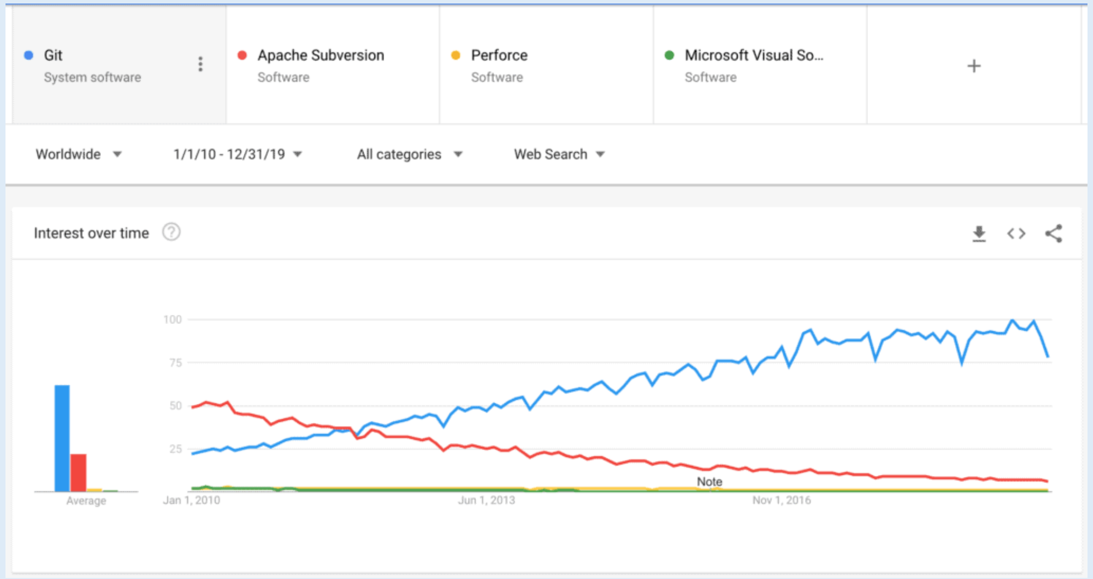
Subversion (SVN)



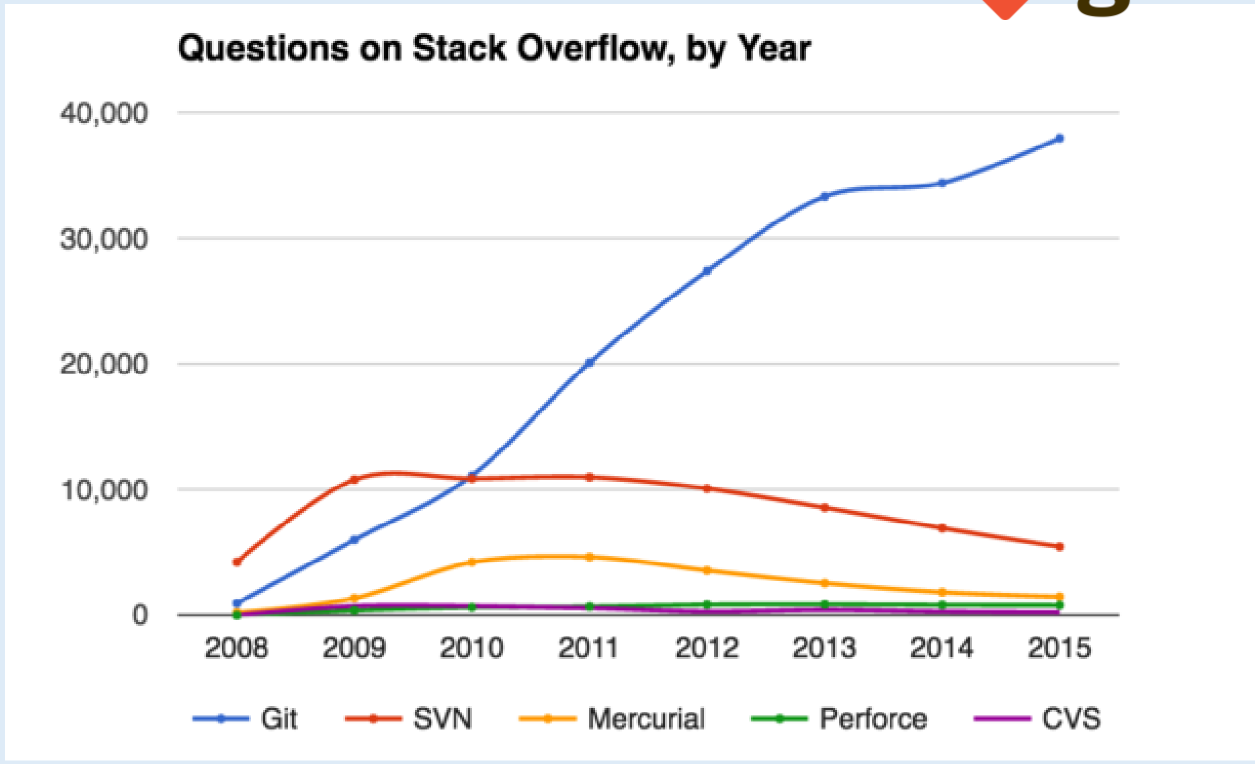
Git, Mercurial

Credit: https://www.edureka.co/blog/what-is-git/

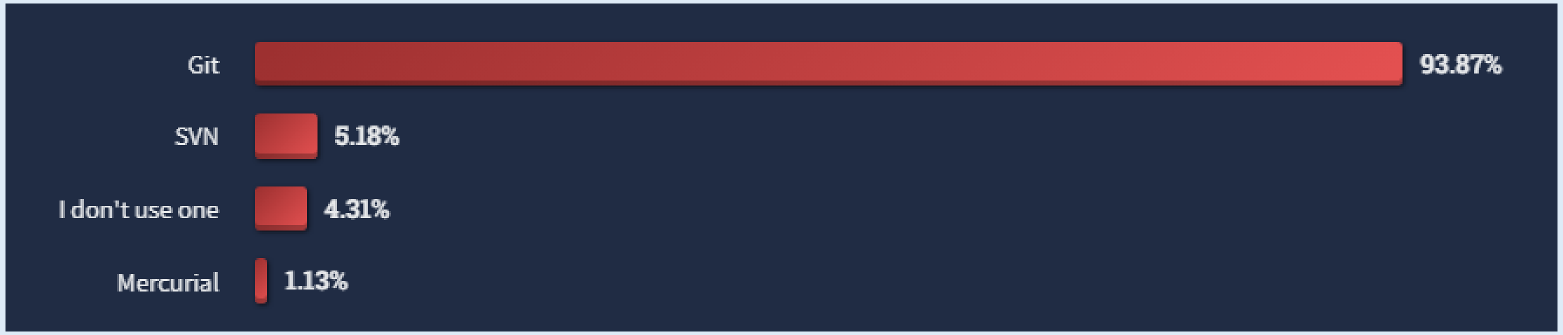
Which VCS to use?



https://sourcelevel.io/blog/7-git-best-practices-to-start-using-in-your-next-commit



https://rhodecode.com/insights/version-control-systems-2016



https://survey.stackoverflow.co/2022/#version-control-version-control-system

What is Git?

Tracking code changes

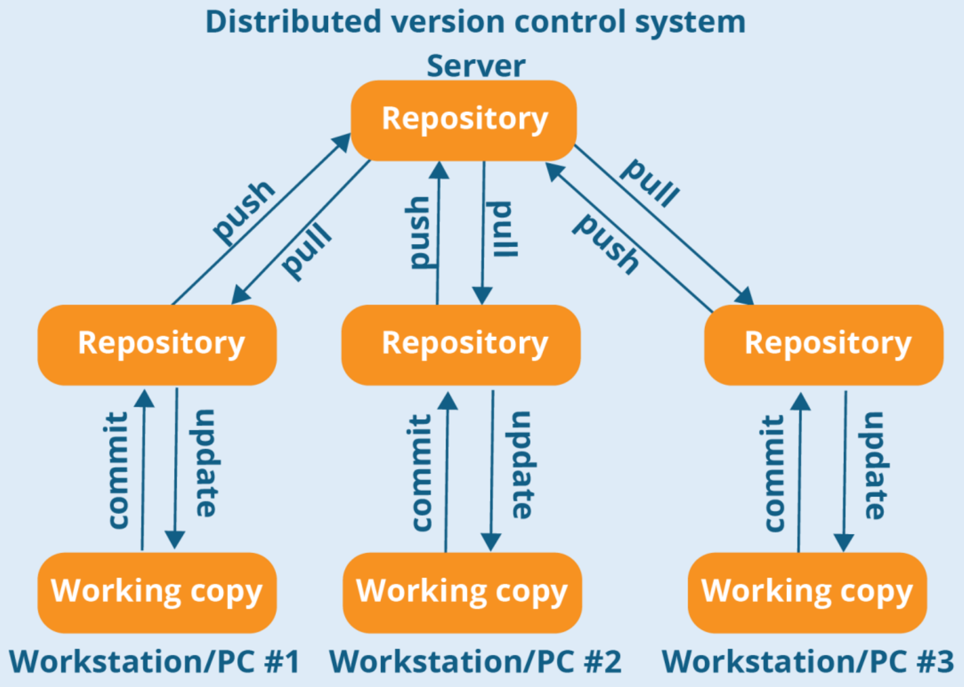
Tracking who made changes

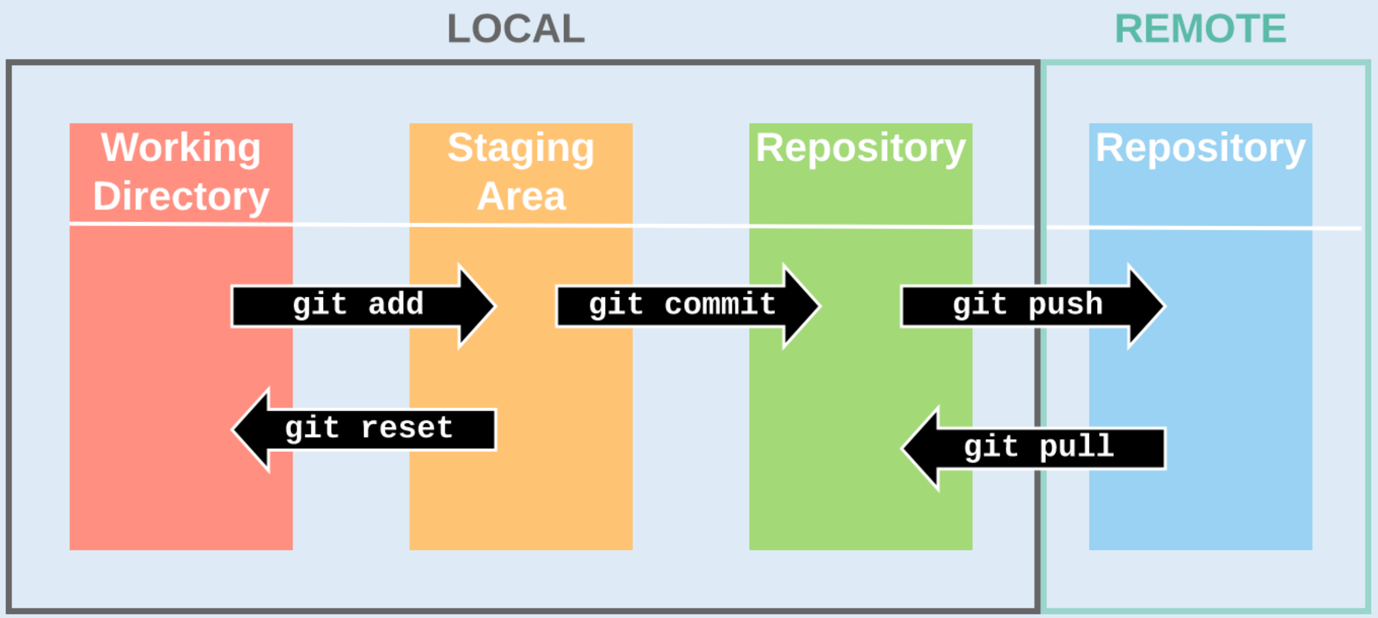
For collaboration

Invented by L. Torvalds for Linux Kernel

Not the same as GitHub or Gitlab

Distributed VCS





https://support.nesi.org.nz/hc/en-gb/articles/360001508515-Git-Reference-Sheet https://git-scm.com/ git

Intro to Git

Installing it -> follow instructions on official site for your system

Configure git on your system:

- git config --global user.name "Jacinda Arden"
- git config --global user.email j.arden@nz_is_awesome.com

Starting to use **git:**

- Go to directory your code is in
- git init
- This creates a .git/ folder with the repo
- Set remote (c.f. next slide)

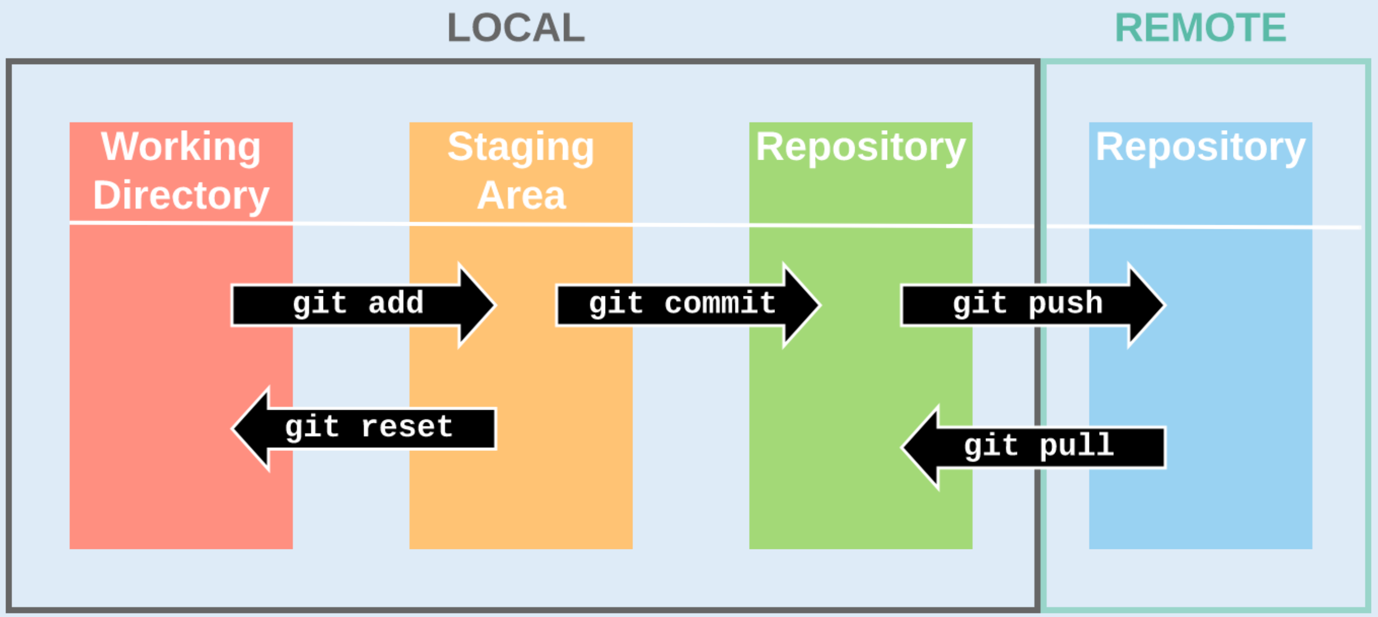
Or simply clone an existing repo:

- git clone <url>

https://git-scm.com/ git

Using Git

- Git will only track the files in the folder you add with:
  - git add <file/dir>
  - DO NOT ADD BIG (>10MB) FILES TO REPO!!
- Only added to repository after a commit:
  - git commit –m "message"
- Pushing to remote:
  - git push



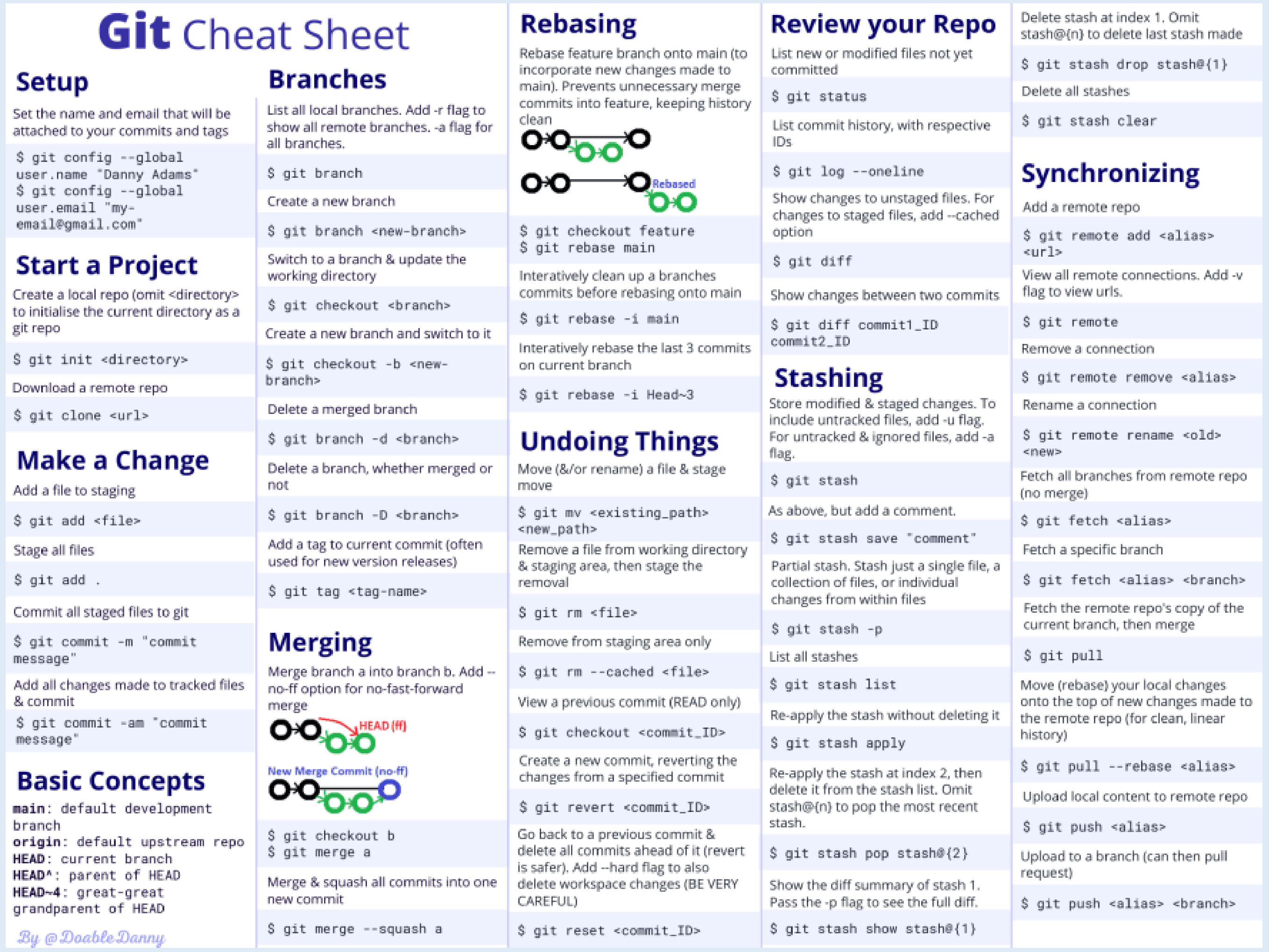
- Try not to have commitment issues.
- Commits are not 'new snapshots'. They are stored changes.
- Informative but short messages.
- Commit and push often.
- Use branching for parallel work
- Use Pull Requests (aka Merge Requests) for collaborative work.

Git Concepts

- Local and remote repos
- Clone
- Staging/tracking
- Commits
- Branches
- Merging
- Checkout
- Pull Requests
- Cherry-picking

Git Cheat Sheets

- https://dev.to/doabledanny/git-cheat-sheet-50-commands-free-pdf-and-poster-4gcn
- https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet
- https://intellipaat.com/blog/tutorial/devops-tutorial/git-cheat-sheet/



Git Cheat Sheet

Setup

Set the name and email that will be attached to your commits and tags

```
$ git config --global
user.name "Danny Adams"
$ git config --global
user.email "my-
email@gmail.com"
```

Start a Project

Create a local repo (omit <directory> to initialise the current directory as a git repo

```
$ git init <directory>
```

Download a remote repo

```
$ git clone <url>
```

Make a Change

Add a file to staging

```
$ git add <file>
```

Stage all files

```
$ git add .
```

Commit all staged files to git

```
$ git commit -m "commit
message"
```

Add all changes made to tracked files & commit

```
$ git commit -am "commit
message"
```

Basic Concepts

```
main: default development
branch
origin: default upstream repo
HEAD: current branch
HEAD^: parent of HEAD
HEAD~4: great-great
grandparent of HEAD
```

By @DoableDanny

Branches

List all local branches. Add -r flag to show all remote branches. -a flag for all branches.

```
$ git branch
```

Create a new branch

```
$ git branch <new-branch>
```

Switch to a branch & update the working directory

```
$ git checkout <branch>
```

Create a new branch and switch to it

```
$ git checkout -b <new-
branch>
```

Delete a merged branch

```
$ git branch -d <branch>
```

Delete a branch, whether merged or not

```
$ git branch -D <branch>
```

Add a tag to current commit (often used for new version releases)

```
$ git tag <tag-name>
```

Merging

Merge branch a into branch b. Add --no-ff option for no-fast-forward merge

HEAD (ff)

New Merge Commit (no-ff)

```
$ git checkout b
$ git merge a
```

Merge & squash all commits into one new commit

```
$ git merge --squash a
```

Rebasing

Rebase feature branch onto main (to incorporate new changes made to main). Prevents unnecessary merge commits into feature, keeping history clean

Rebased

```
$ git checkout feature
$ git rebase main
```

Interatively clean up a branches commits before rebasing onto main

```
$ git rebase -i main
```

Interatively rebase the last 3 commits on current branch

```
$ git rebase -i Head~3
```

Undoing Things

Move (&/or rename) a file & stage move

```
$ git mv <existing_path>
<new_path>
```

Remove a file from working directory & staging area, then stage the removal

```
$ git rm <file>
```

Remove from staging area only

```
$ git rm --cached <file>
```

View a previous commit (READ only)

```
$ git checkout <commit_ID>
```

Create a new commit, reverting the changes from a specified commit

```
$ git revert <commit_ID>
```

Go back to a previous commit & delete all commits ahead of it (revert is safer). Add --hard flag to also delete workspace changes (BE VERY CAREFUL)

```
$ git reset <commit_ID>
```

Review your Repo

List new or modified files not yet committed

```
$ git status
```

List commit history, with respective IDs

```
$ git log --oneline
```

Show changes to unstaged files. For changes to staged files, add --cached option

```
$ git diff
```

Show changes between two commits

```
$ git diff commit1_ID
commit2_ID
```

Stashing

Store modified & staged changes. To include untracked files, add -u flag. For untracked & ignored files, add -a flag.

```
$ git stash
```

As above, but add a comment.

```
$ git stash save "comment"
```

Partial stash. Stash just a single file, a collection of files, or individual changes from within files

```
$ git stash -p
```

List all stashes

```
$ git stash list
```

Re-apply the stash without deleting it

```
$ git stash apply
```

Re-apply the stash at index 2, then delete it from the stash list. Omit stash@{n} to pop the most recent stash.

```
$ git stash pop stash@{2}
```

Show the diff summary of stash 1. Pass the -p flag to see the full diff.

```
$ git stash show stash@{1}
```

Delete stash at index 1. Omit stash@{n} to delete last stash made

```
$ git stash drop stash@{1}
```

Delete all stashes

```
$ git stash clear
```

Synchronizing

Add a remote repo

```
$ git remote add <alias>
<url>
```

View all remote connections. Add -v flag to view urls.

```
$ git remote
```

Remove a connection

```
$ git remote remove <alias>
```

Rename a connection

```
$ git remote rename <old>
<new>
```

Fetch all branches from remote repo (no merge)

```
$ git fetch <alias>
```

Fetch a specific branch

```
$ git fetch <alias> <branch>
```

Fetch the remote repo's copy of the current branch, then merge

```
$ git pull
```

Move (rebase) your local changes onto the top of new changes made to the remote repo (for clean, linear history)

```
$ git pull --rebase <alias>
```

Upload local content to remote repo

```
$ git push <alias>
```

Upload to a branch (can then pull request)

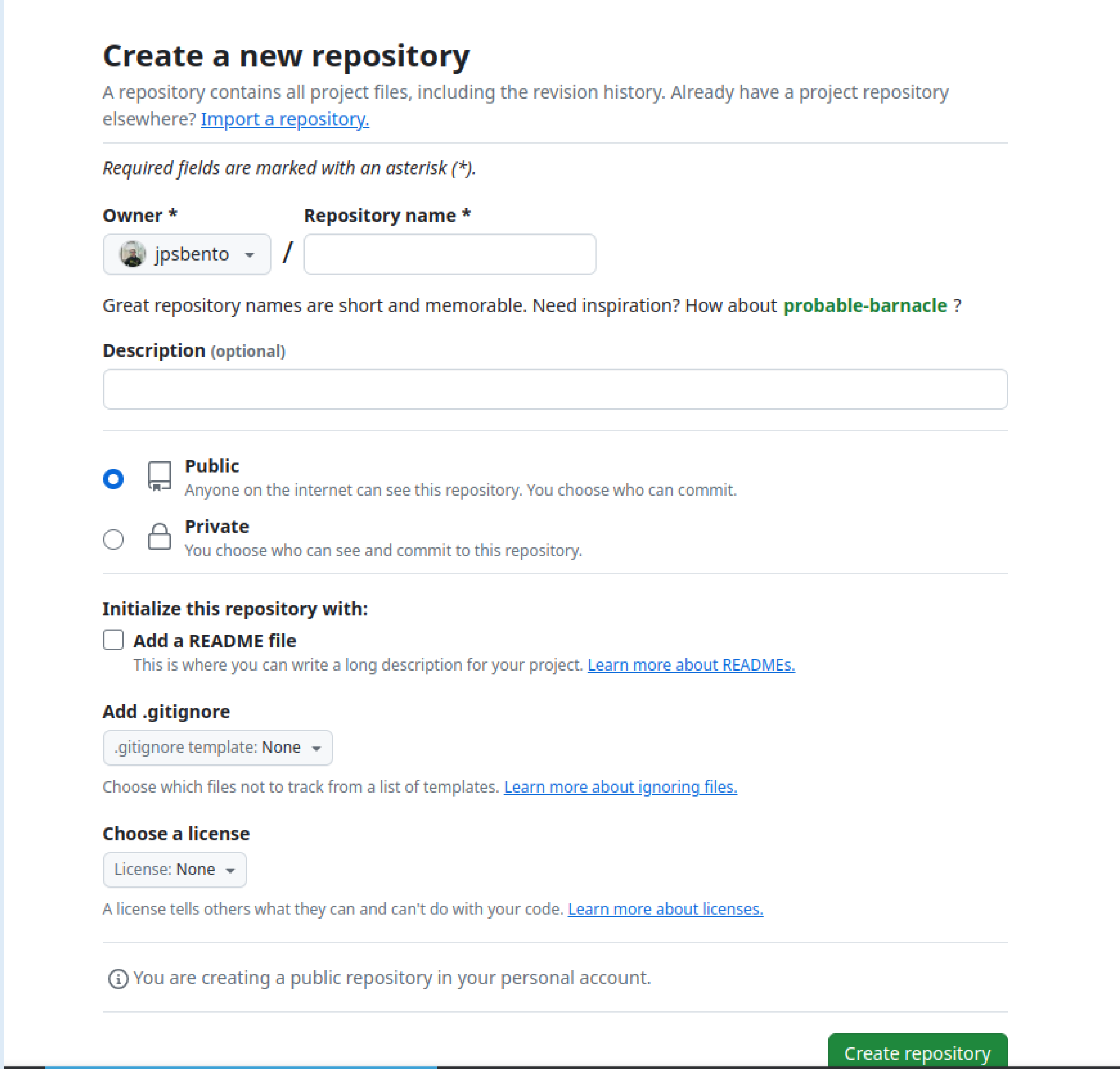
```
$ git push <alias> <branch>
```

Typical workflow for individuals

- Sarah has some code that they want version controlled. For this we assume GitHub setup already in place.
- They create a GitHub repository called 'some_code' (origin).
- Then, once created, they clone it locally. A some_code/ folder is created, with a local repo and a .git/ folder inside. The GitHub repo is set as 'origin' already.
- They copy all the existing code to this new directory, and start committing files, pushing to origin and generally working on it.
- A few weeks later, Sarah makes a breaking change to the code. It no longer works and it's not obvious why. Luckily, they can diff the current state with an earlier version that worked and can either identify the breaking change or revert back to that earlier version.
- Success! This code can even be shared with others easily in the future.

Task 1: Create a new Github repository

- Create a new repo
- Clone it locally:
  - git clone git@github.com:<username>/<repo>.git
- Make a new file, add some content
- Git add, commit, push
- Inspect github
- Branch, commit and merge



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

Owner * jpsbento / **Repository name ***

Great repository names are short and memorable. Need inspiration? How about **probable-barnacle** ?

Description (optional)

● **Public** Anyone on the internet can see this repository. You choose who can commit.

○ **Private** You choose who can see and commit to this repository.

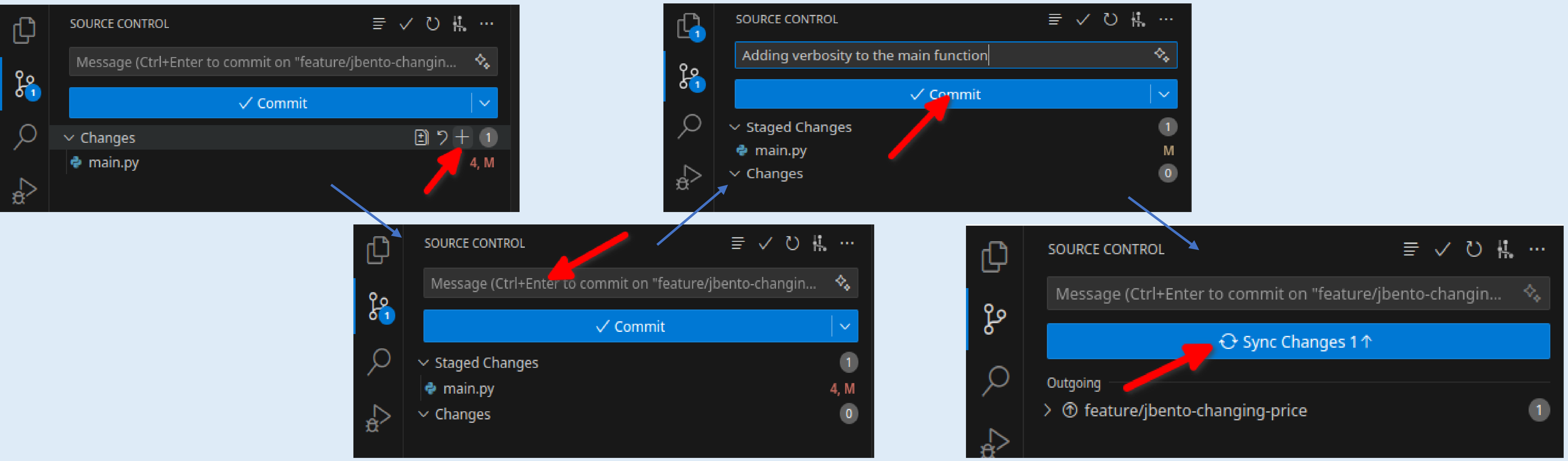
Initialize this repository with:

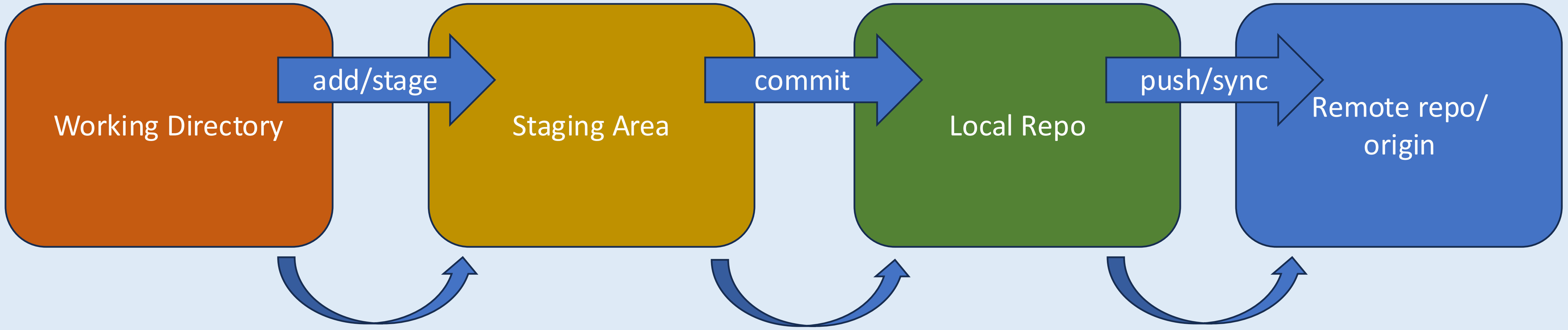
☐ **Add a README file**
This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore
.gitignore template: None
Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license
License: None
A license tells others what they can and can't do with your code. Learn more about licenses.

ⓘ You are creating a public repository in your personal account.


SOURCE CONTROL
Message (Ctrl+Enter to commit on "feature/jbento-changin...
✓ Commit
Changes
main.py
4, M
SOURCE CONTROL
Adding verbosity to the main function
✓ Commit
Staged Changes
main.py
M
Changes
0
SOURCE CONTROL
Message (Ctrl+Enter to commit on "feature/jbento-changin...
✓ Commit
Staged Changes
main.py
4, M
Changes
0
SOURCE CONTROL
Message (Ctrl+Enter to commit on "feature/jbento-changin...
Sync Changes 1
Outgoing
feature/jbento-changing-price
1



git add <file(s)/folder>

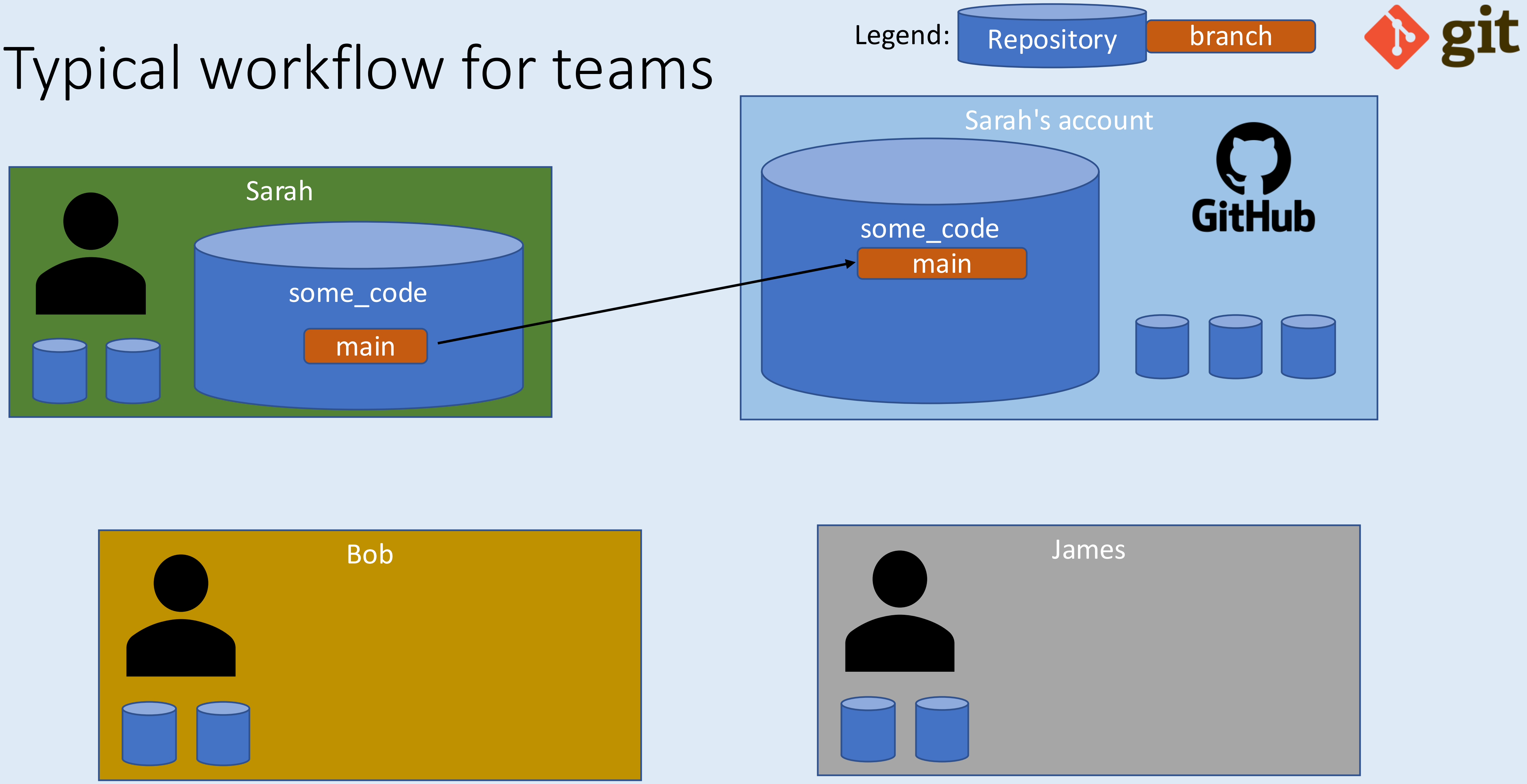
git commit -m <message>

git push

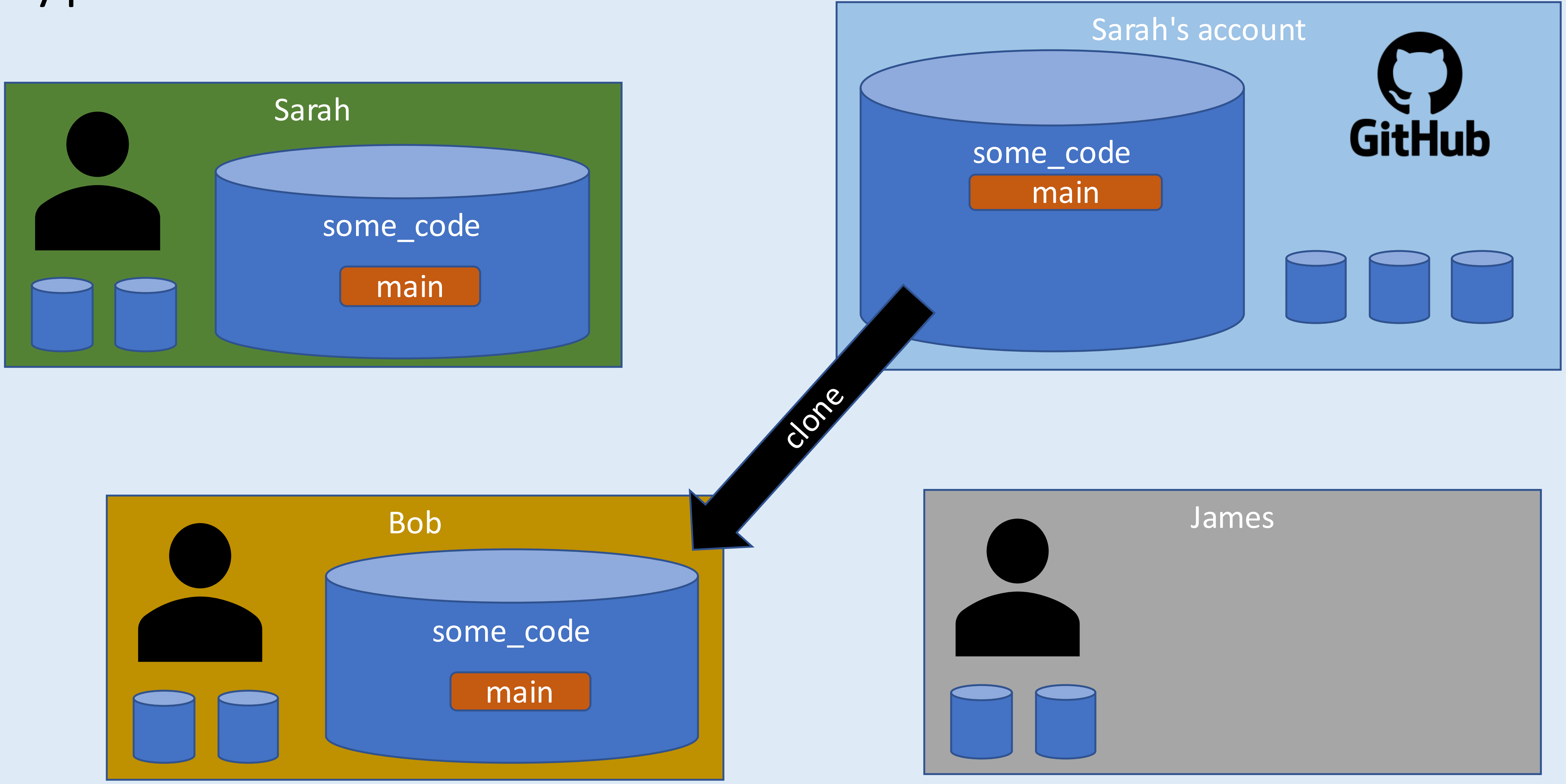
Typical workflow for teams

- Sarah has some code that they want Bob and James to make contributions to. Sarah shares the GitHub repo with them.
- Bob clones it locally and creates a branch called 'bob_working'. Bob continues to code on that branch, committing and pushing to GitHub.
- Separately, James clones the repo, creates a new 'feature/james-awesome-stuff' branch and starts working on a new feature there.
- Bob is done with their work and wants it to be part of the 'main' branch (the code in production). They create a PR and ask Sarah to review it.
- Sarah approves and merges the PR. Bob's code is now part of the 'main' branch.
- A few days later, James wants to do the same. They create a PR and ask Sarah to approve, with optional review by Bob. Immediately, GitHub detects that, when merged, this would cause a code conflict with Bob's new code, which Bob also comments on.
- James pulls the 'main' branch from GitHub to their local repo and merges it to the 'feature/james-awesome-stuff' branch. Git detects the same merge conflict and James is able to resolve it, commiting and pushing the changes. The conflict is no longer present on both the local repo and at GitHub.
- Sarah approves and merges the PR and James' code is now on the 'main' branch too, ready for production.

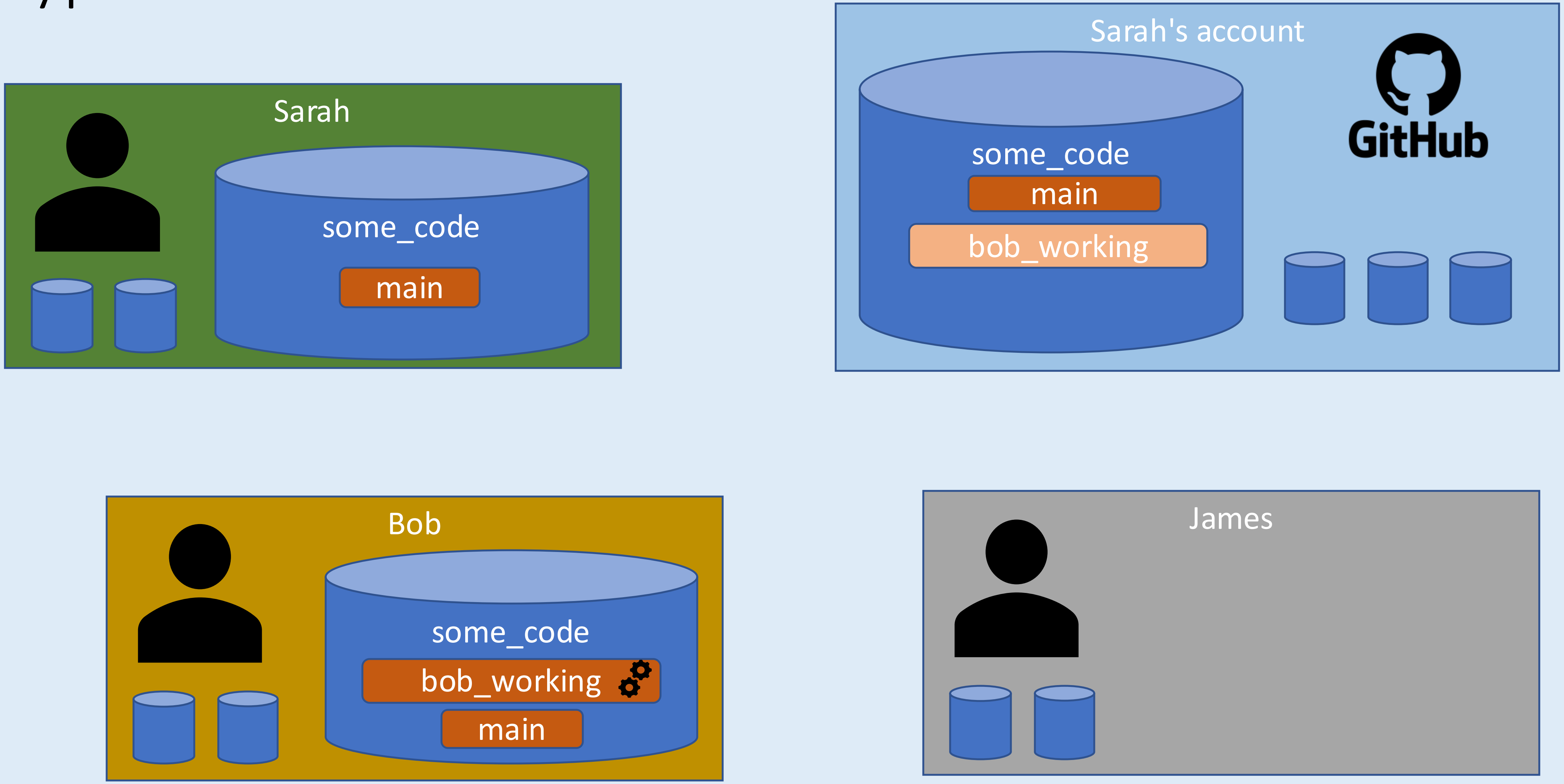
Typical workflow for teams



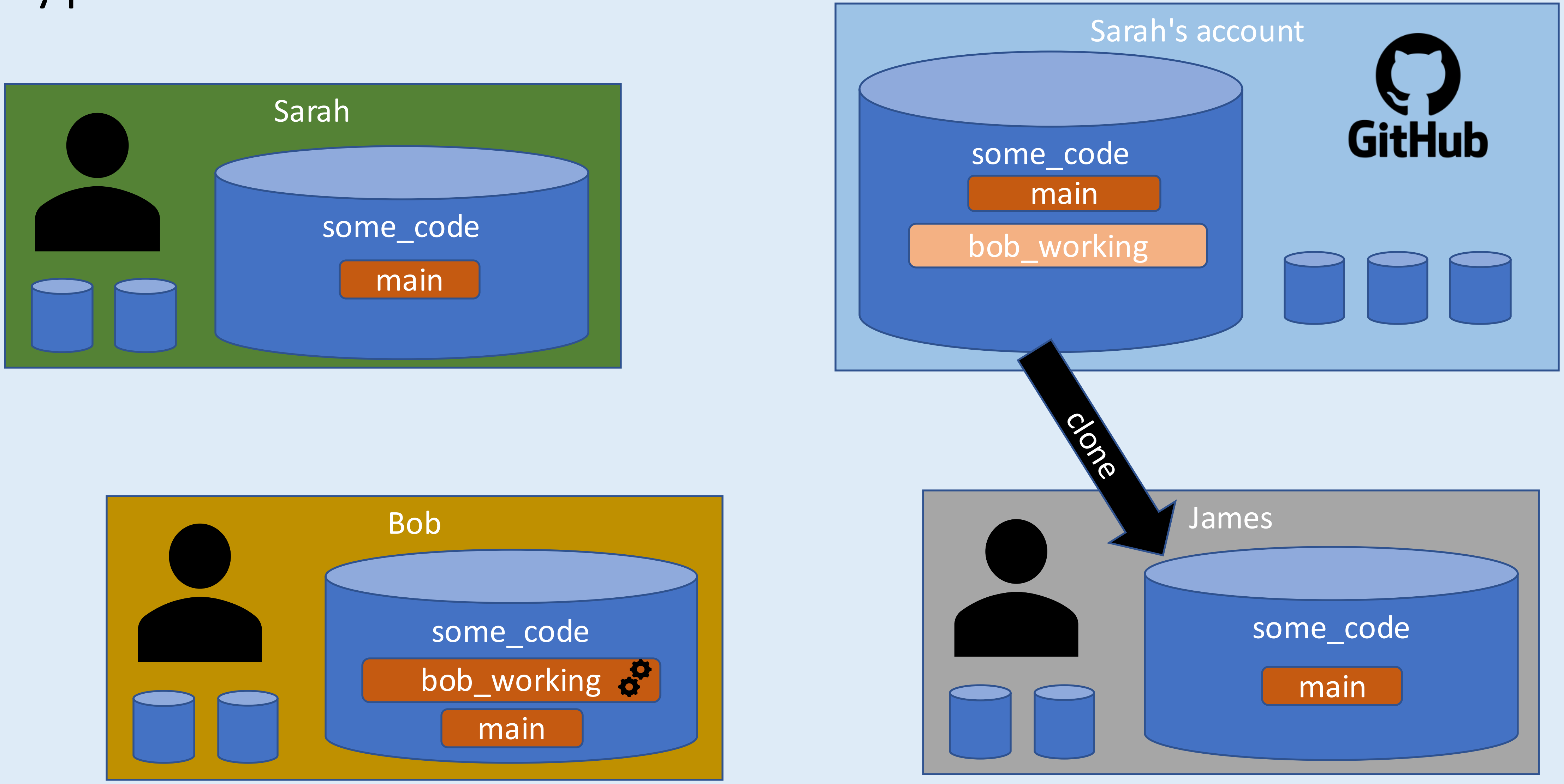
Typical workflow for teams



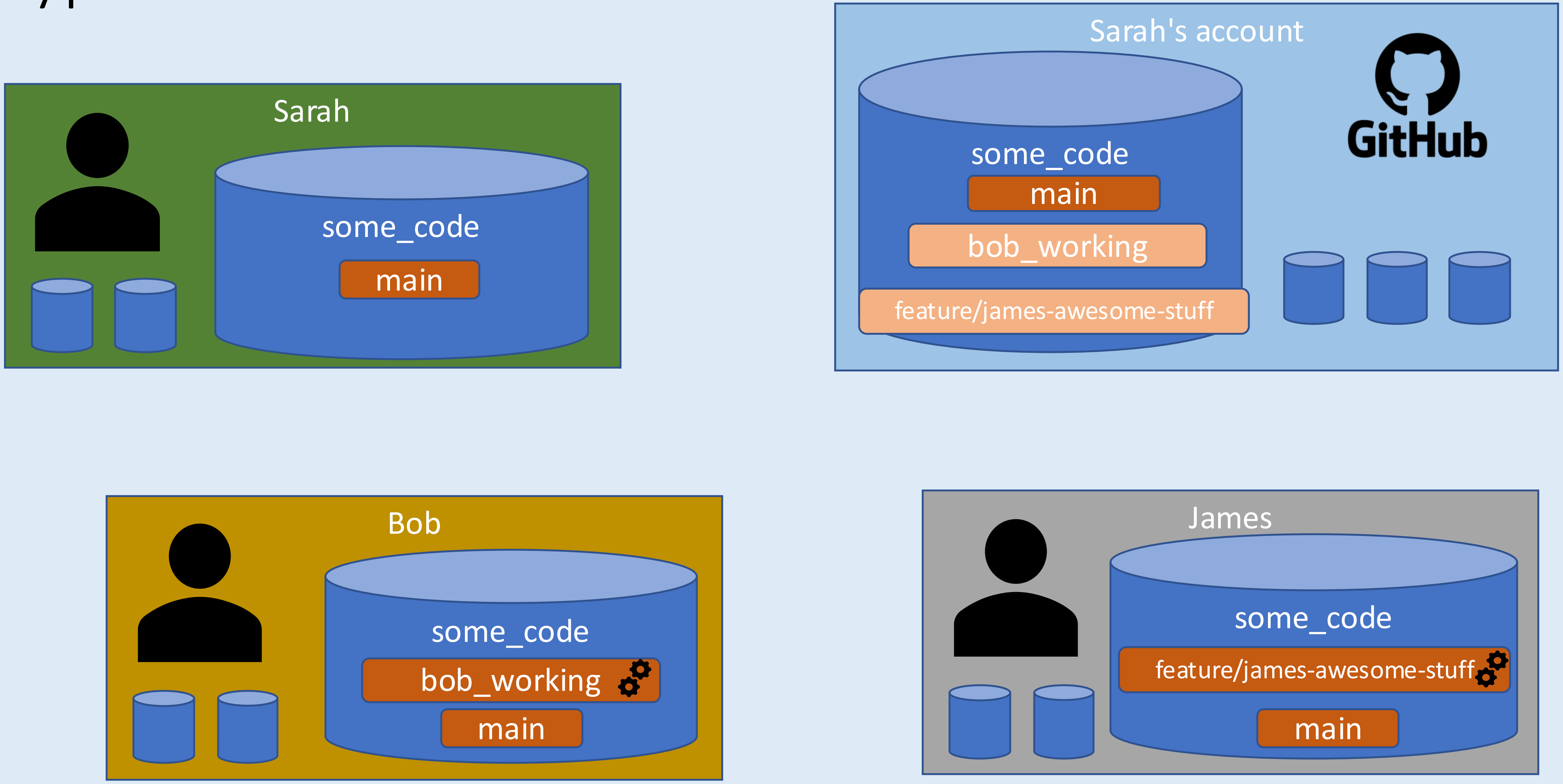
Typical workflow for teams



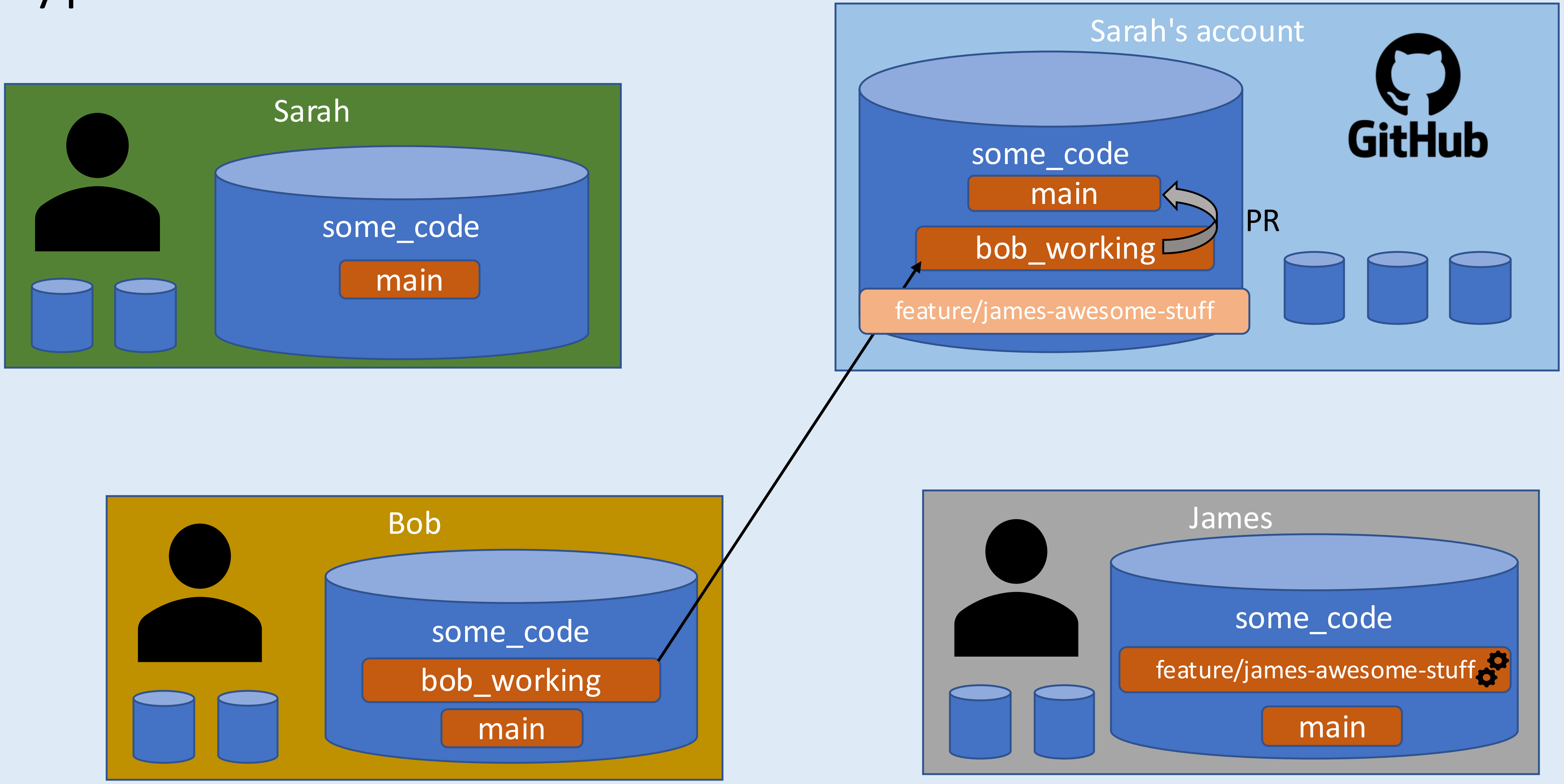
Typical workflow for teams



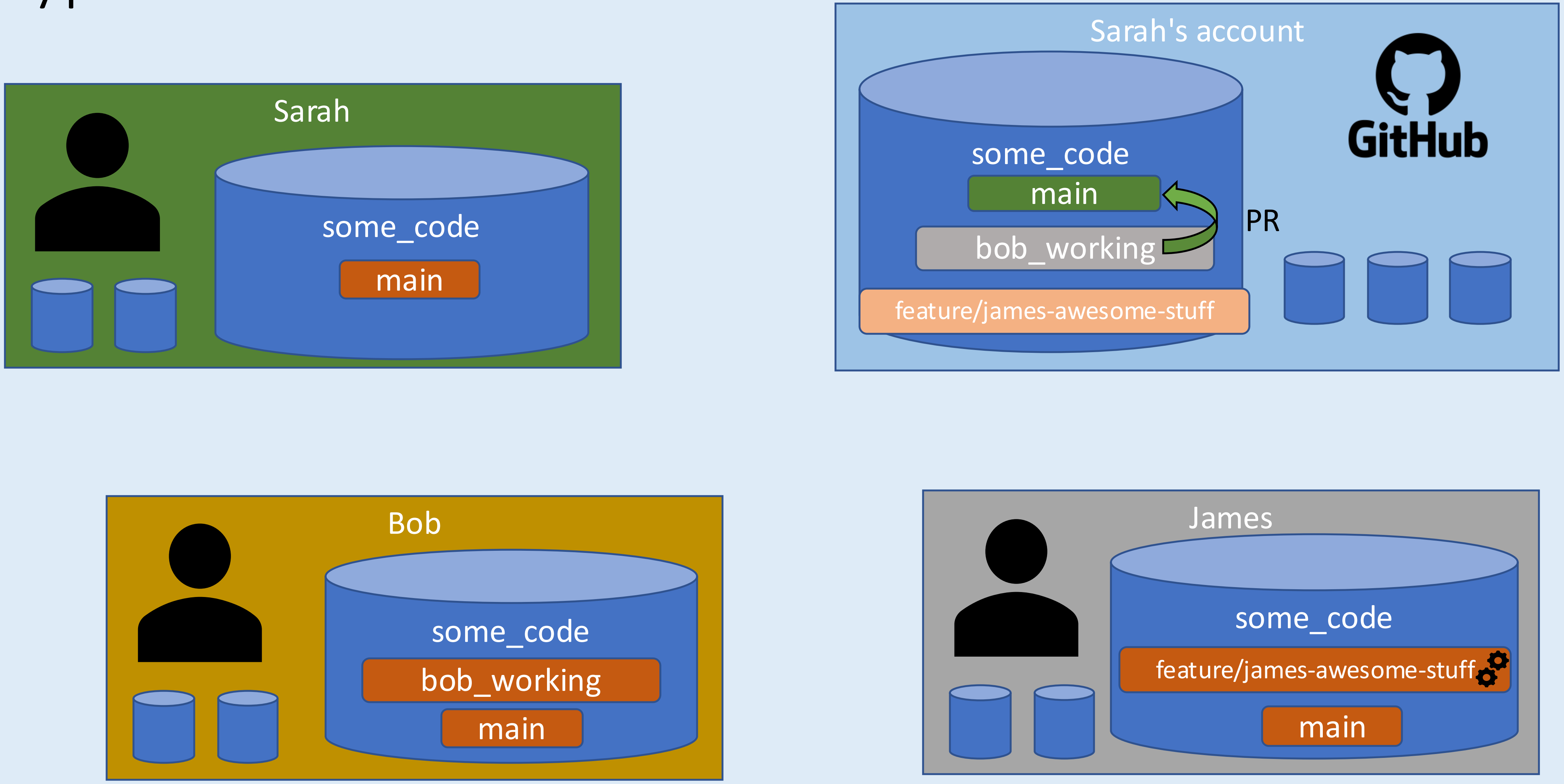
Typical workflow for teams



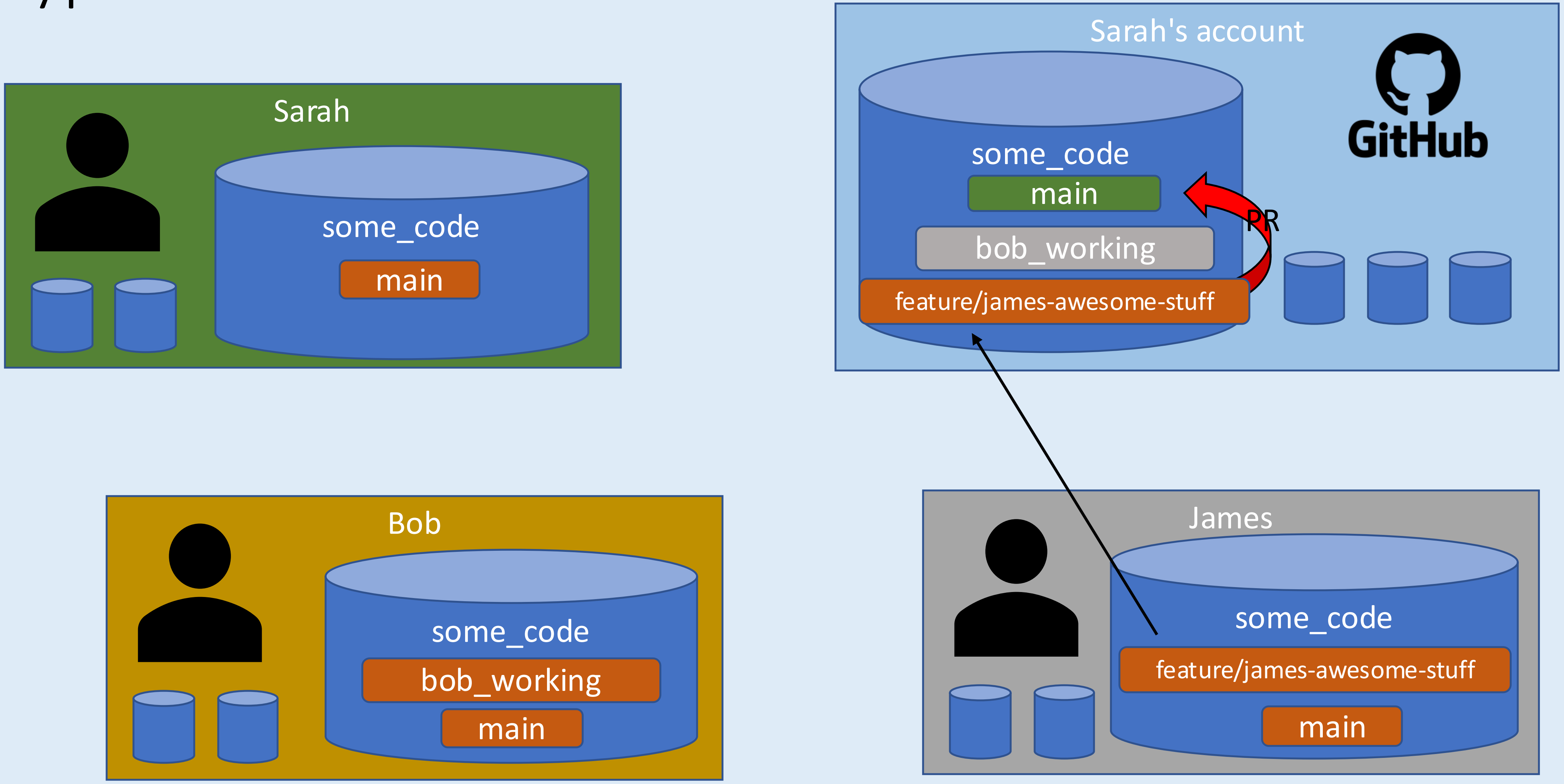
Typical workflow for teams



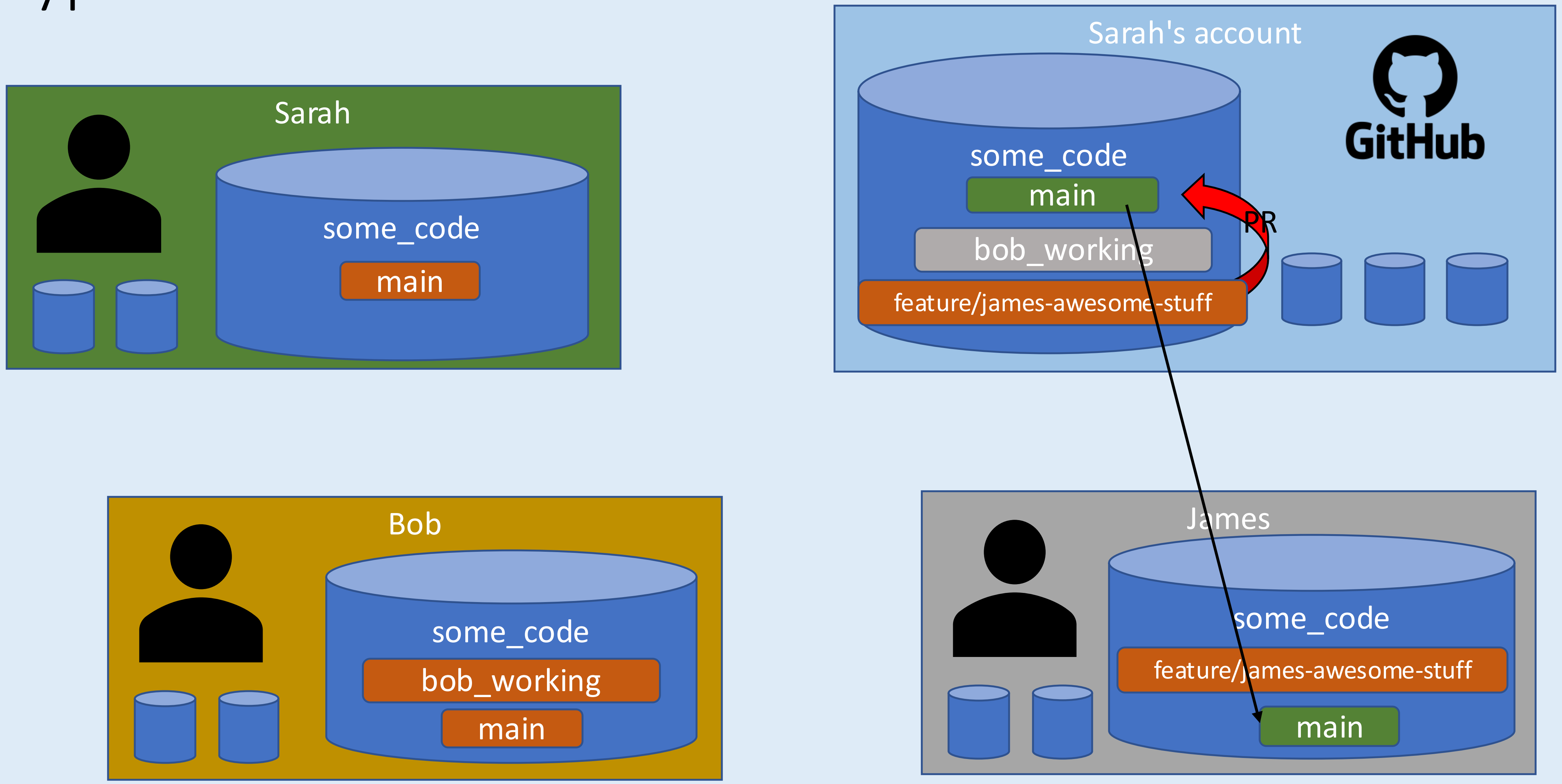
Typical workflow for teams



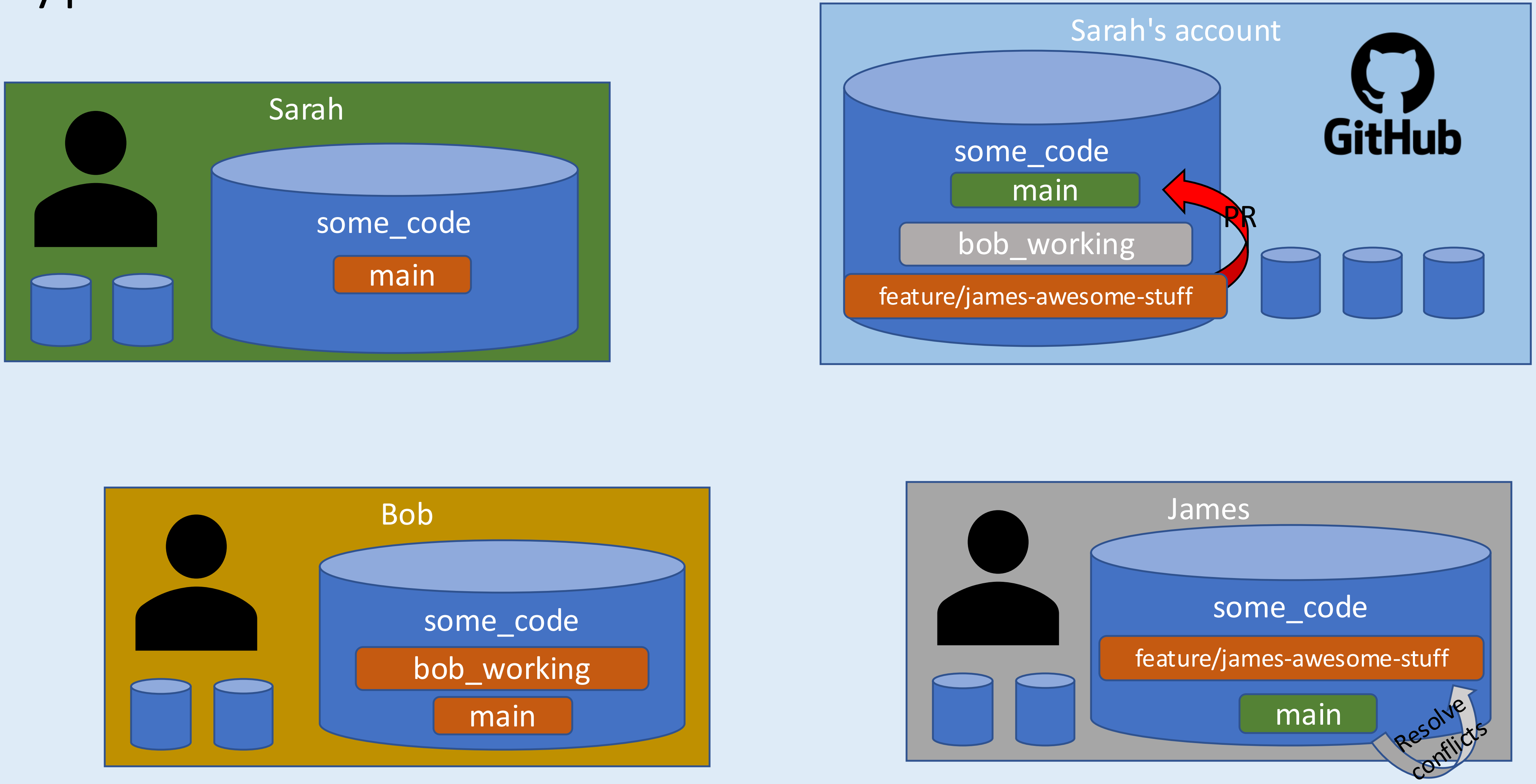
Typical workflow for teams



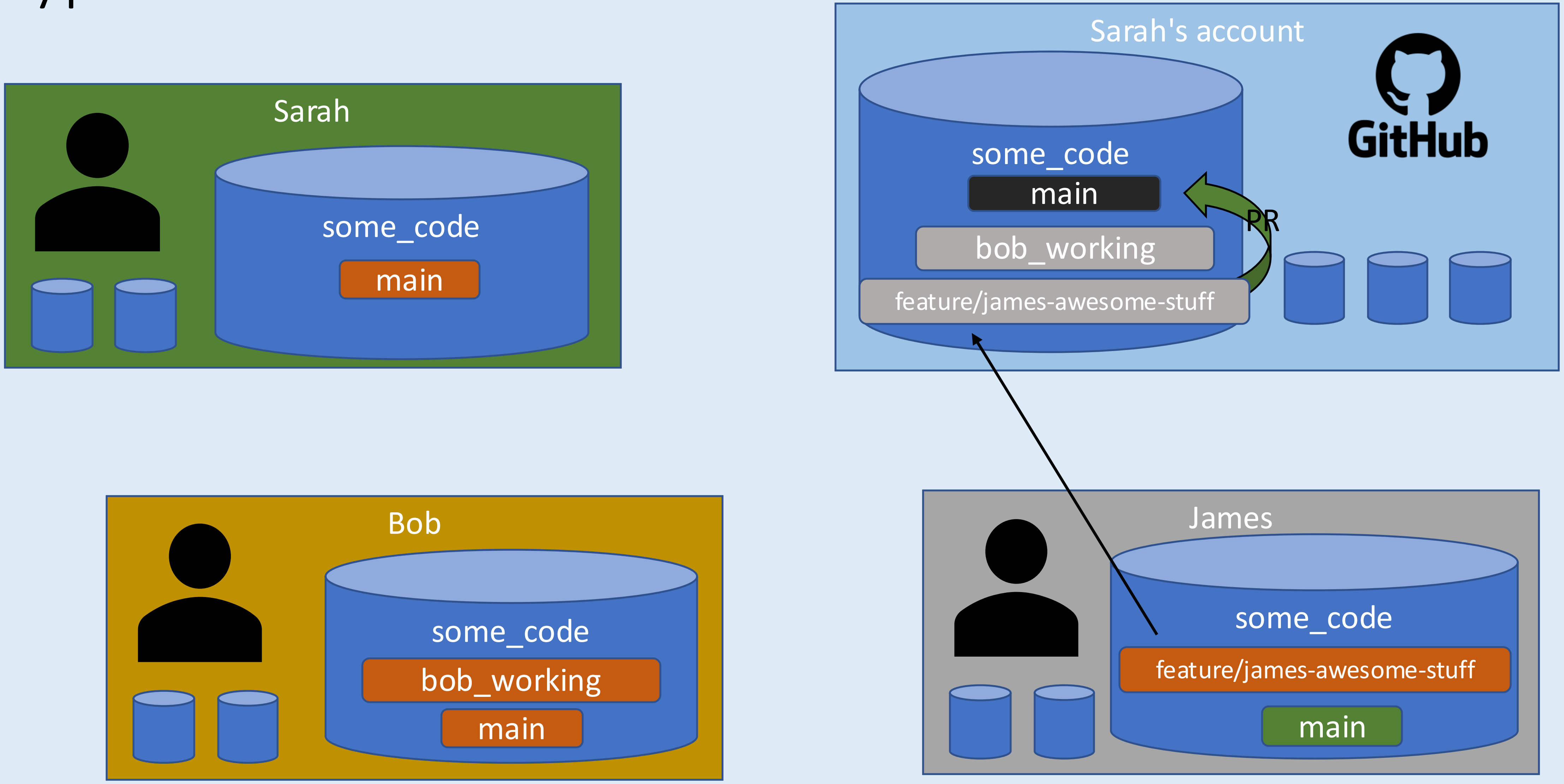
Typical workflow for teams



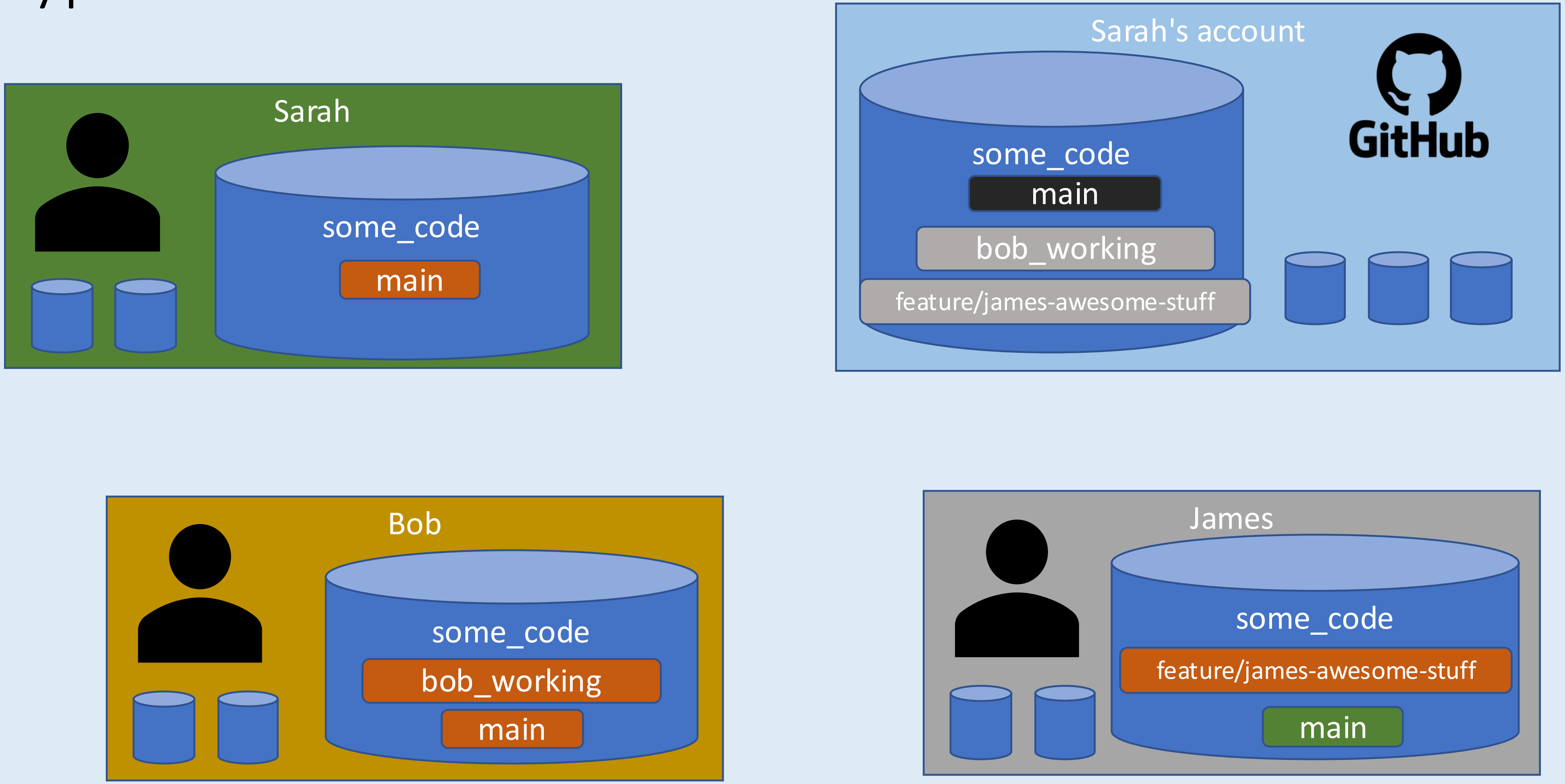
Typical workflow for teams



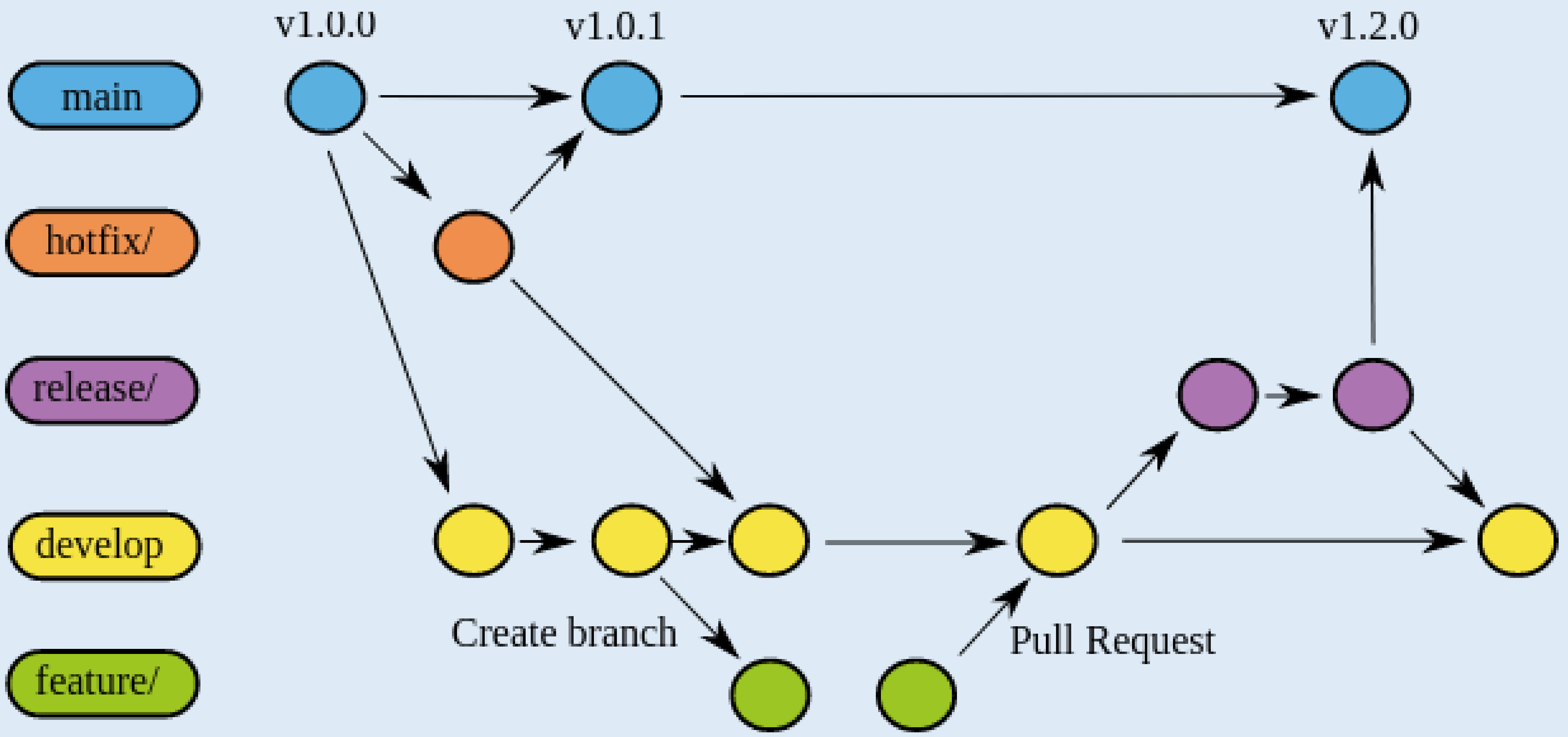
Typical workflow for teams



Typical workflow for teams



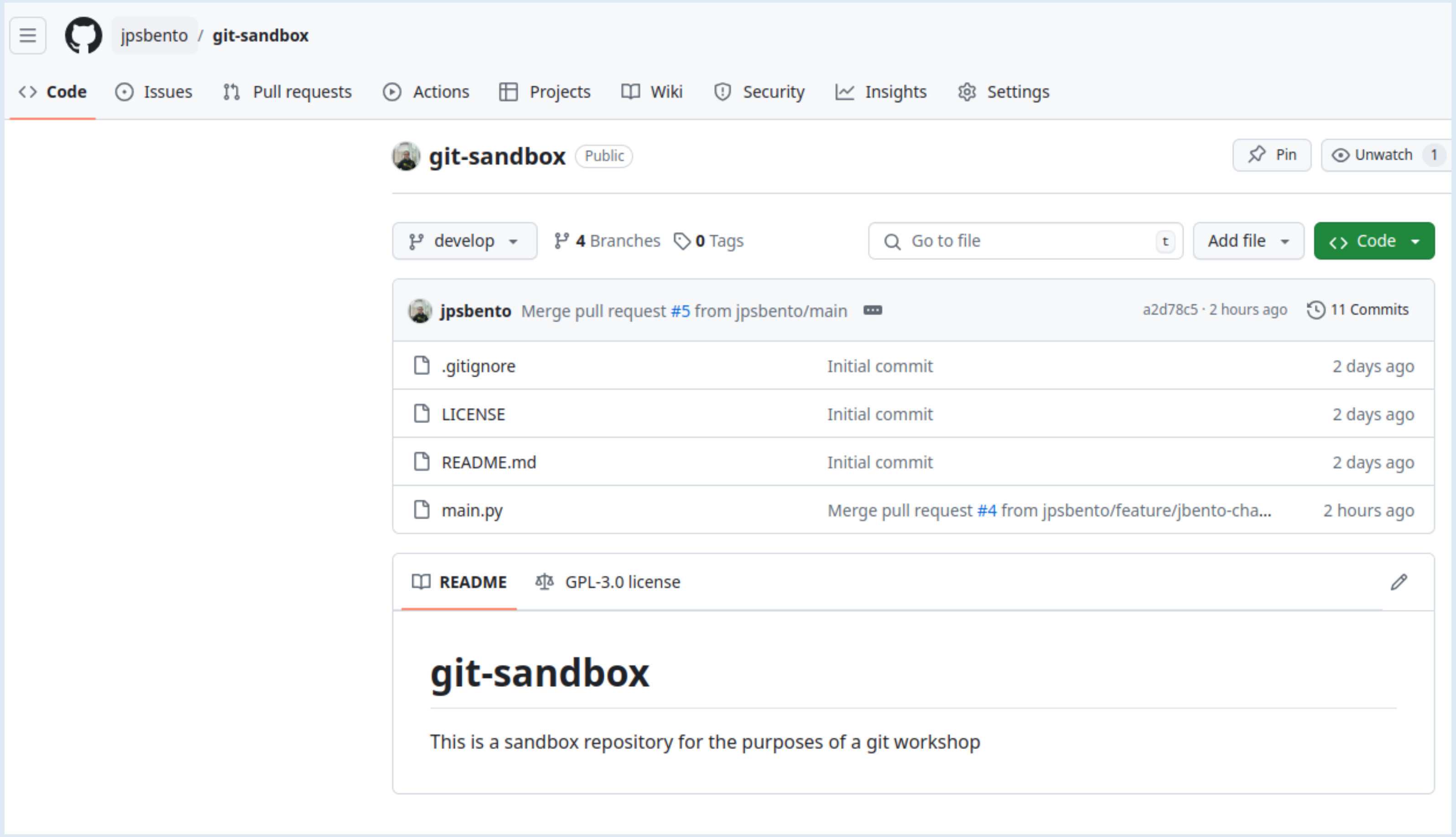
GitFlow



https://docs.ultrazohm.com/general/project_structure/contribution_workflow.html

Task 2: Contribute to an existing repo
https://github.com/jpsbento/git-sandbox

- git clone git@github.com:jpsbento/git-sandbox.git



jpsbento Merge pull request #5 from jpsbento/main a2d78c5 · 2 hours ago 11 Commits

.gitignore	Initial commit	2 days ago
LICENSE	Initial commit	2 days ago
README.md	Initial commit	2 days ago
main.py	Merge pull request #4 from jpsbento/feature/jbento-cha...	2 hours ago

README GPL-3.0 license

git-sandbox

This is a sandbox repository for the purposes of a git workshop

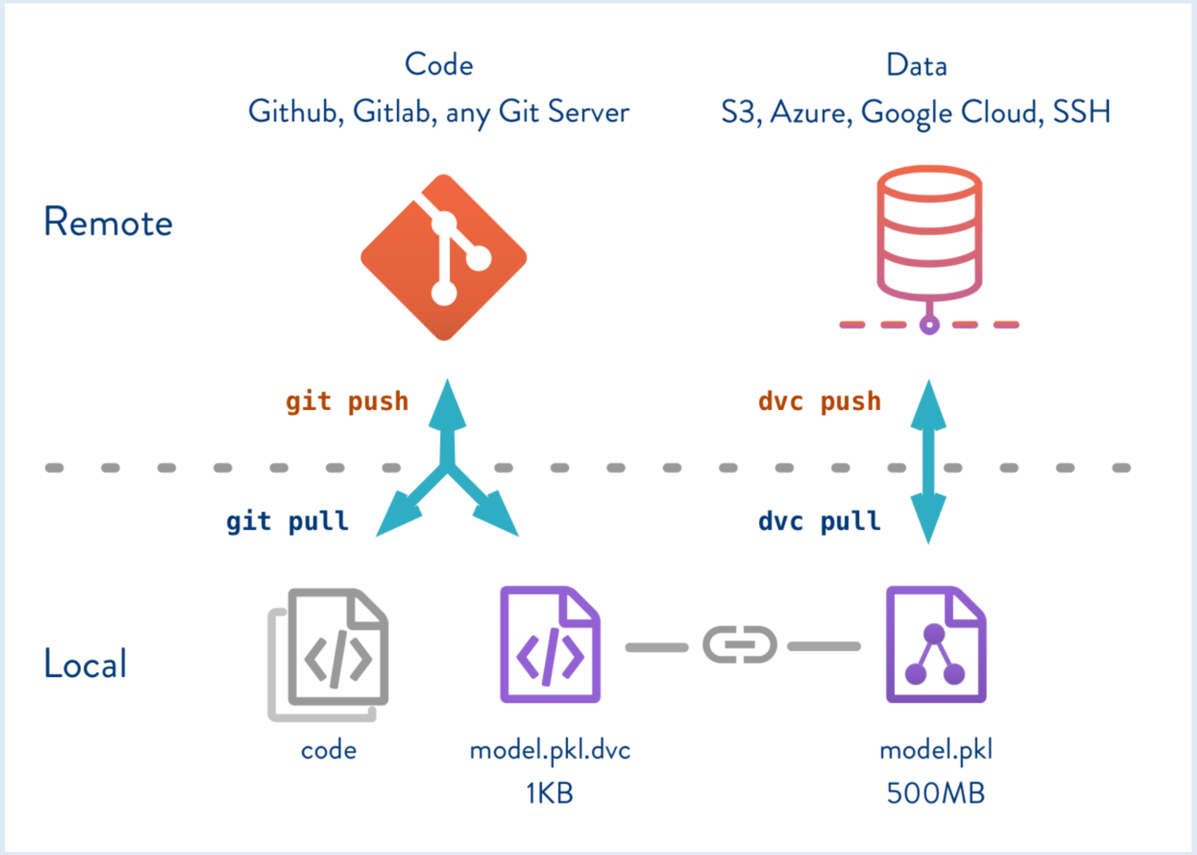
- Clone this repo locally
- Scenarios:
  - A: Multiple developers work on same branch (merge conflicts)
  - B: Multiple developers on separate branches and merging with PRs
  - C: Cherry picking

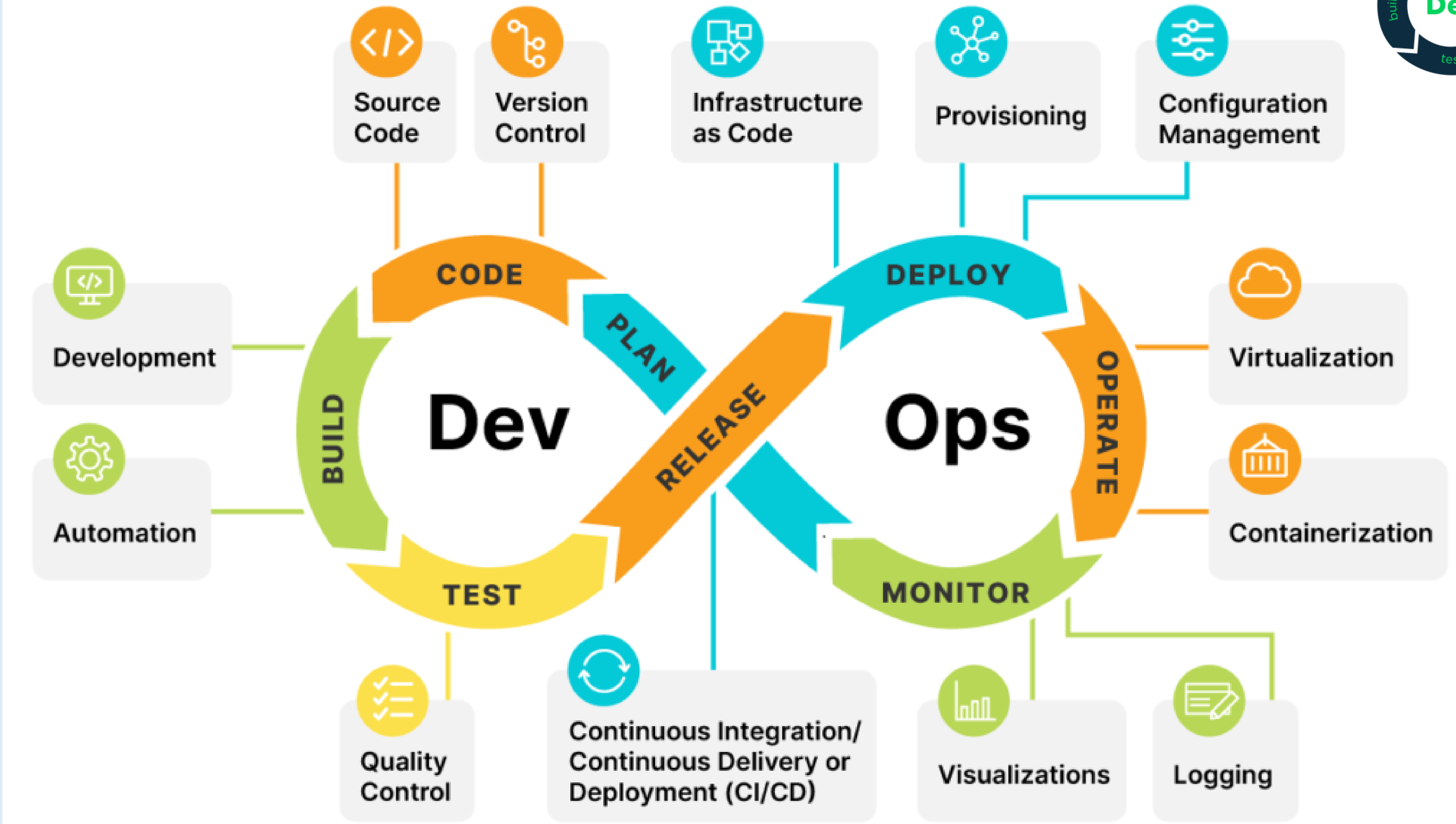
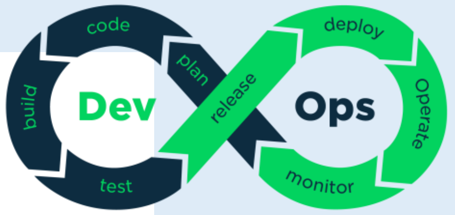
Git Summary

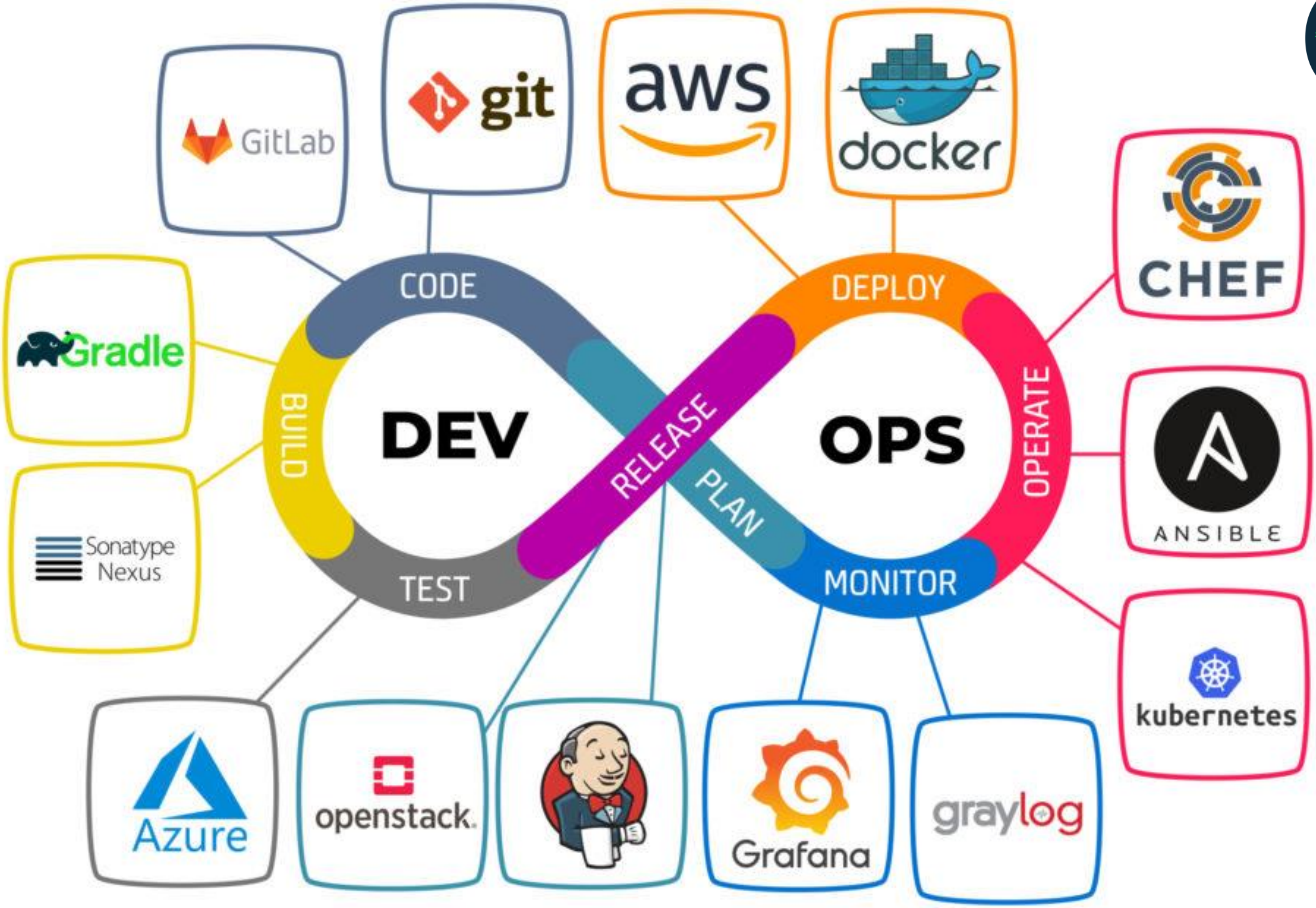
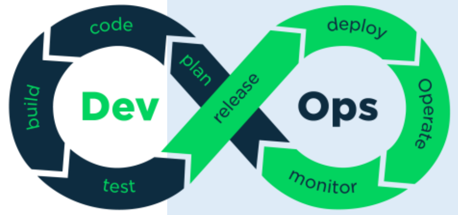
- It's a distributed VCS, great for collaborative work.
- It's the industry standard, and it's for everyone.
- It is easy to use, once you get used to it. So get used to it!
- Integrates with all systems, IDEs and works everywhere.
- Has one major caveat: It doesn't work well with large files or binary files.
    - DON'T COMMIT A LARGE FILE. Even if you remove it later, the repo will still have the original version of it.
    - Purging a file completely from the repository history is VERY HARD.
    - For large file VCS, use either Git-LFS, MLFlow or DVC.

Data Version Control (dvc.org)

- Efficient Handling of Large Files
- Reproduce Test Results and Original Data
- Works alongside Git
- Storage Flexibility
- Some basic DevOps knowledge required
- Increased Complexity




Source Code
Version Control
Infrastructure as Code
Provisioning
Configuration Management
Development
Automation
Virtualization
Containerization
CODE
PLAN
BUILD
Dev
TEST
RELEASE
DEPLOY
Ops
OPERATE
MONITOR
Quality Control
Continuous Integration/ Continuous Delivery or Deployment (CI/CD)
Visualizations
Logging

code
plan
build
Dev
test
release
deploy
Ops
Operate
monitor


GitLab
git
aws
docker
CHEF
Gradle
CODE
DEPLOY
BUILD
DEV
RELEASE
PLAN
OPS
OPERATE
ANSIBLE
Sonatype Nexus
TEST
MONITOR
kubernetes
Azure
openstack
Grafana
graylog

code
plan
deploy
build
Dev
release
Ops
Operate
test
monitor