Deployment: Closing the loop

*Programs that are never deployed have not fulfilled their purpose. We must deploy!*

To do so we must answer:

- Is our application in a working state?
- Do we have the necessary HW/SW resources?
- How do we actually get our code, etc. onto a server accessible by others?
DevOps principles

• Involve operations in each phase of a system’s design and development
• Heavy reliance on automation versus human effort
• The application of engineering practices and tools to operations tasks
Continuous Integration (CI): Ensuring our application is in a working state

- Maintain a single repository
  *With always deployable branch*
- Automate the Build (Build is a proper noun)
  *And fix broken builds ASAP*
- The Build should be self testing
- Everyone integrates with main frequently
  *Small “deltas” facilitate integration and minimize bug surface area*
- Automate deployment
  *Practice “DevOps” culture*

Martin Fowler “Key practices of Continuous Integration”
CI, CD and more

CI rigorously tests every integration in production-like environment
• Prevent development-production mismatch
• Test multiple browsers, etc.
• “Stress test” code for performance, fault-tolerance, etc.

Then we deploy!

By deploying frequently, we make what was rare and fraught common and unremarkable!
Why version control?

“Version control (or source control or revision control) serves as a safety net to protect the source code from irreparable harm, giving the development team the freedom to experiment without fear of causing damage or creating code conflicts.”

-GitLab
Git workflow for CI

Main is always “deployable”
- Tests pass
- No incomplete features

Short-lived branch for single feature

- Branching is cheap in Git
- We will use branches to isolate changes until integration
- The “main” branch remains deployable
This is Git. It tracks collaborative work on projects through a beautiful distributed graph theory tree model.

Cool. How do we use it?

No idea. Just memorize these shell commands and type them to sync up. If you get errors, save your work elsewhere, delete the project, and download a fresh copy.
Git “solo” branching workflows

```
git checkout -b feature

main
feature

```

```
git commit -m "..."
...

```

```
git checkout main
git merge feature

Make sure
tests pass

```

```
git checkout main
```
**Git/GitHub workflow with CI**

**Alice**
- `git push origin feature` [Graph: push to remote]

**Github**
- CI server tests branch and merged code [Graph: CI test]
- Merge PR and delete feature branch [Graph: merge]

**Branching**
- `git branch -d feature` [Graph: delete branch]

**Pulling**
- `git checkout main`
- `git pull --prune`
Git is distributed, staying up to date

```sh
git checkout main
git pull origin main

# Alice

git checkout feature
git rebase main

git push origin feature

# Github

git checkout main
git pull origin main

git push origin feature

PR
```
Trying out git: Visualizing branches

First type "git fetch". Then type "git rebase origin/master".

- git checkout master
- git pull origin main
  Fetched 3 commits on master.

Fast-forwarded to origin/master.

- git checkout my-branch
- git merge master

Local Repository
Current Branch: my-branch

https://onlywei.github.io/explain-git-with-d3
git branch vocabulary

First type "git fetch". Then type "git rebase origin/master".
$ git fetch
Fetched 3 commits on master.

Local Repository
Current Branch: my-branch
branches in remote repository
remote-tracking branch
"local" tracking branch
local branch

https://onlywei.github.io/explain-git-with-d3
The golden rule of rebase (and any re-writing of history)

- Never modify public history (commits)
  If anyone else could see this feature branch (e.g., you pushed to GitHub), don’t use rebase, --force, or any command that alters history
- When in doubt it is OK to just merge
Conflicts happen: Merge commits

On branch feature
Unmerged paths: (use "git add/rm ..." as appropriate to mark resolution)
both modified: App.js

Git identifies the conflicts:

here is some content not affected by the conflict
<<<<<<<<< HEAD
this is conflicted text from feature branch
=======
this is conflicted text from main
>>>>>>> main

Fix all conflicts then add updated files and commit to complete the merge
Create our own merge conflict

https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts
Student advice: Branch-per-feature

- “Aggressive branch-per-feature minimized merge conflicts”
- “With this many people you NEED branch-per-feature to avoid stepping on each other”

Our goal is to work efficiently as a project team. *Practice now the processes you will need in your project!*

Adapted from Berkeley CS169
Client-Server

HTTP & URI

HTML, JSON, ...

3-tier Architecture

Client (e.g. browser)  Internet  Site

Web Server (e.g. Apache, nginx)  csci312.dev  Database (e.g. SQLite, PostgreSQL)

App. Server (e.g. NodeJS)

Presentation Tier

Logic Tier

Persistence Tier

Routing & Controllers (e.g. NextJS, Express)

Models (e.g. knex, objection)

MVC

*aaS (e.g., PaaS) “factor out” the common needs
High-level project workflow

GitHub Actions

GitHub

Alice

Bob

Push & PR

“Customer” Feedback

csci312.dev