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 deploy?

True or False? The development team's goal of launching new features conflicts with the operations team's goal of ensuring services stay live and usable.

A. True

B. False

Answer: True

From the Google SRE handbook: "At their core, the development teams want to launch new features and see them adopted by users. At their core, the ops teams want to make sure the service doesn't break while they are holding the pager. Because most outages are caused by some kind of change—a new configuration, a new feature launch, or a new type of user traffic—the two teams' goals are fundamentally in tension."

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

As a practical matter, the trend towards DevOps means that as the application developer you are responsible for more of the traditional ”operations” tasks (provisioning machines, deploying, etc.) while ”operations” teams are increasingly automating operational tasks to support frequent deployment, fault tolerance, and more.

Definition sourced from: https://landing.google.com/sre/book

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 emphasizes frequent small integrations (hence the name)

There are two related concepts:
* *Continuous Deployment*: Every change automatically gets put into production, and thus there are many production deployments each day.
* *Continuous Delivery*: An extension of CI in which SW is deployable throughout its lifecycle, the team prioritizes keeping SW deployable, and it is possible to automatically deploy SW on demand.

https://martinfowler.com/bliki/ContinuousDelivery.html

We will be aiming for a Continuous Delivery-like workflow in which our applications start and stay deployable throughout the development process. As with CI, this reduces the complexity (and risk) of deployment by enabling us to do so in small increments. And Continuous Delivery facilitates getting user feedback by frequently getting working SW in front of real users. Although to mitigate risk companies will often first deploy for a small subset of users.

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

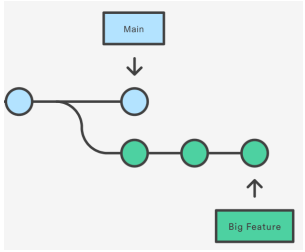
At the beginning of the course, we handed you git and said use this, but only briefly talked about why… I though this definition from GitLab captured it quite well.

- Protect our source code from irreparable loss
- Make us more comfortable making changes
- And increasingly as infrastructure for supporting collaboration, testing, and deployment

That you use version control (often abbreviated VCS) is more important than what you use. Git is not the only choice and not all companies use Git (Google most famously). That said it is very widely used, partly driven by the use and adoption of GitHub.

Git was originally developed to support the distributed development model used for the Linux kernel and its design reflects that need. As a reminder Git is a distributed VCS. Each repository "stands alone" and changes are not automatically propagated between repositories. Instead, we need to explicitly communicate changes.

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 features branches to isolate changes until integration
- The “main” branch remains deployable

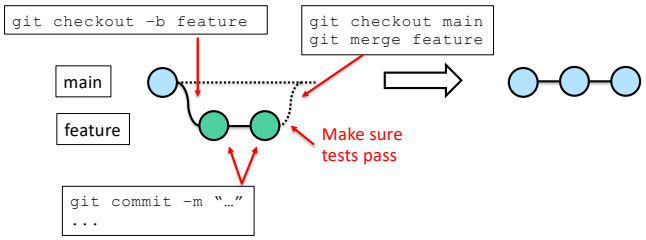
There are many Git workflows. We will adopt one particular workflow that works well for our kind of project and infrastructure. We will use of “feature” branches for all development.

A note about Git branch naming. The “main” branch used to be named “master” (and is still by default). That naming is no longer used as it is a charged term (https://www.acm.org/diversity-inclusion/words-matter) that does not reflect the relationship between branches. We will use “main” (although you will still see ”master” in some projects and documentation).

A key attribute is “short lived”. This workflow is most effective when branches only live for a brief period (as opposed to days or weeks) and most feature development can start main.

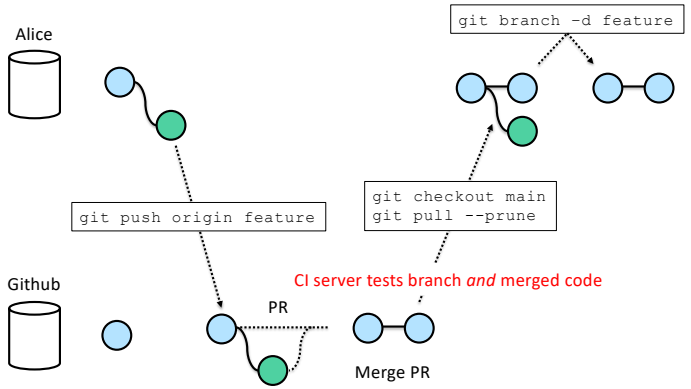
https://www.atlassian.com/git/tutorials/using-branches

Git “solo” branching workflows



But we are rarely working alone. On a team we need to make sure we stay in sync and create opportunities to get a second pair of eyes on our code (i.e. create opportunities for code review).

Git/GitHub workflow with CI



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!

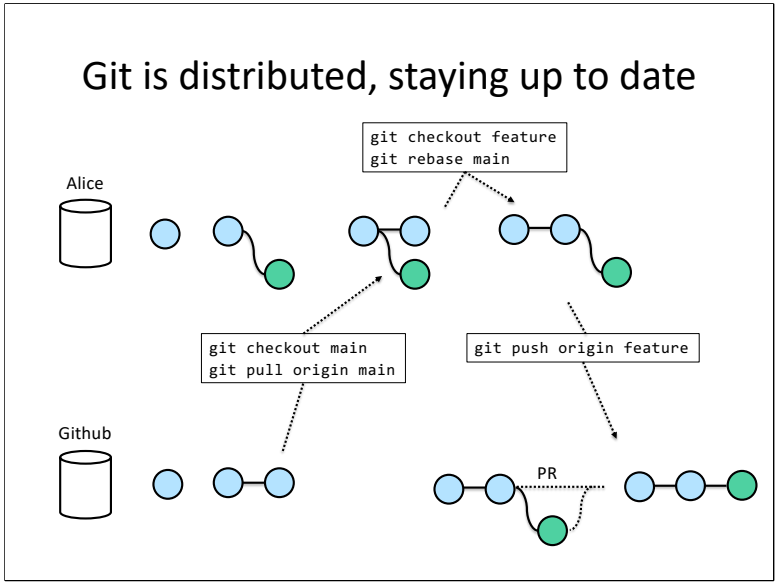
Adapted from Berkeley CS169

You try to push to a remote branch and get a “(non-fast-forward) error: failed to push some refs […]" message. What should you do?

A. Use "--force" argument to force Git to complete the push
B. You must still have merge conflicts. Manually fix those conflicts then push.
C. There have been intervening commits to remote branch. Pull then push again.

Answer: C

(A) will rewrite shared history; (B) is not necessarily true; (C) this error is created when there are changes to the remote branch that haven't been fetched to the local repository.



In the ideal case, your feature branch will merge with main cleanly without any conflicts (basically, the changes in the feature can be applied directly without overwriting anything that has happened to main since the feature branched off).

From the picture, what does rebase do? Put my changes on "after" previous changes, thus making it appear I branched after the most recent changes. Note that this is optional and can be fraught (stay tuned).

Another "friendly" behavior is to squash your feature commits to just one (because you have been making lots of commits as you go right!). Doing so makes it easier to review.

What happens when you need to make changes to your PR. Commit and push to remote feature branch, will be added to PR.

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 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

Assume we have tried to merge main into our feature branch in anticipation of pushing our feature branch to GitHub (and creating a pull request). But we get conflicts. Git will put us in a "half-way" state where we can fix the merge conflicts. What do we mean by fixing the conflicts? You have to choose between the code on your branch and the code on the branch your merging, or some combination thereof. Git marks these with the angle brackets and the equals signs (it may also include code from the nearest common ancestor). Let's assume you wanted the code form your branch. You would delete the angles and equals, and keep just the code you wanted, e.g.

```
this is conflicted text from feature branch
```

Often though you will need to integrate the two, e.g. you will edit the above section to be:

```
this is conflicted text from feature branch and main
```

One you have resolved all these conflicts (and made sure your tests pass!), you can complete the merge by adding all the files you modified (git add …) and committing (git commit …). This create the merge conflict that shows git how to combine these branches. As a result, when you create your pull request Git will know how to automatically merge your feature branch onto master.

Check out for more details: https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts

The operational work involved in supporting a service should realistically scale how as the service grows by 10X?

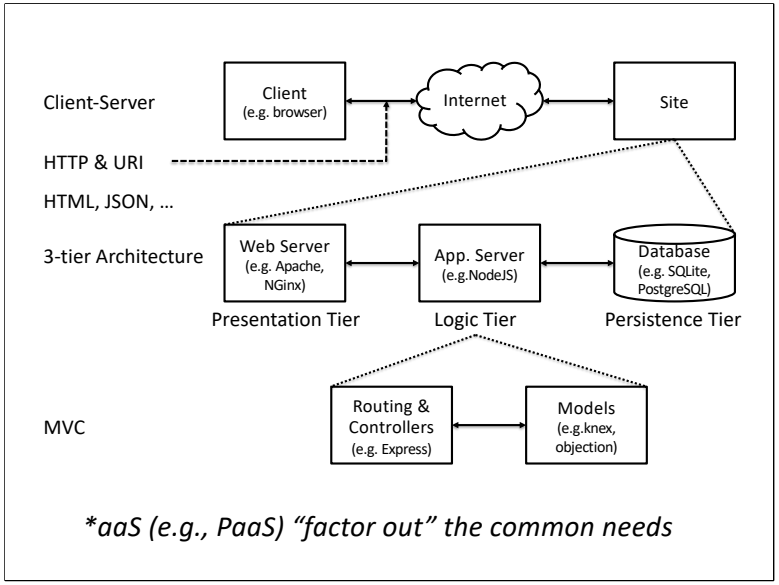
A. *O(1)*: Just one-time efforts to add resources
B. Sublinear: There will be additional work required as a function of service size
C. *O(n)*: The effort will have to grow linearly with demand
D. Greater than *O(n)*: Increasing scale means increasing complexity

Answer: A

Again, from the Google SRE handbook: "An ideally managed and designed service can grow by at least one order of magnitude with zero additional work, other than some one-time efforts to add resources." To do so, one needs highly automatic systems.

Automation (and engineering practices) are what enables that constant effort scaling. Automation goes beyond just provisioning resources, it is techniques like automatically rolling out changes to a small fraction of users, detecting errors (through monitoring) and then automatically rolling back the changes!

Although DevOps wasn't a thing (and thus not a job), the role of site reliability engineer (SRE) is the closest to DevOps as a job. Popularized by Google, SREs are engineers who focus on running products and "create systems to accomplish the work that would otherwise be performed, often manually, by sysadmins."



As described previously the 3-tier architecture is a design pattern. PaaS factor out the common elements of that architecture. For example, the Heroku PaaS provides the "presentation tier" and the "persistence tier" and the portions of the "logic tier" that wrap around your specific application.

This semester we are going to try working with fly.io as our deployment platform. Fly uses Docker to package up build process for our application (getting all dependencies installed) and offers a command line tool for actually deploying our application to their system and managing the required resources.

*aaS and the cloud has eliminated all physicality from the process but also change the dynamic from provisioning (and decommissioning) HW infrequently to doing so frequently, thus forcing automation even by otherwise small-scale users.