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?

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 [i.e., on call]. 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."

This is an instance of a broader issue. Often the "safest" course of action is no action at all. Accomplishing something requires some risk. Our goal is to create processes and teams that will take on and successfully manage those risks.

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

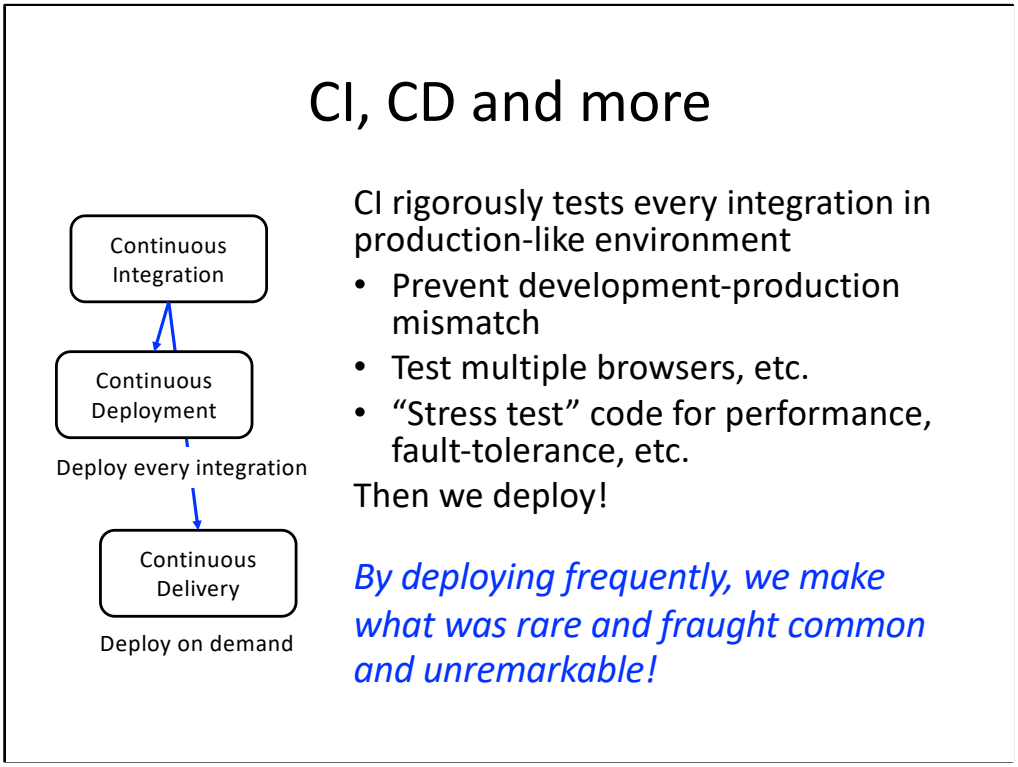
Continuous Integration (CI): Ensuring our application is in a working state

- 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"

One of those practices is Continuous Integration (CI). CI emphasizes frequent integrations (hence the name). The main ideas are… <clicks>

There are lot of different terminology in this space, not all of it consistently or clearly defined. Terms may not translate. "Everyone integrates frequently" is sometimes described as "trunk-based development". We contrast this approach with workflows with long-lived branches (maybe tied to a specific release, or a major feature), perhaps with rules for who can merge code between branches.



We rigorously and automatically test each integration. We will use GitHub actions for this purpose. At a minimum it prevents the dreaded "but it worked on my machine". The integration testing typically goes beyond the testing each developer is doing on their own code to include multiple browsers, stress tests, etc. Once that integration is complete, we are ready to deploy!

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 (small) 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

In our project, 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

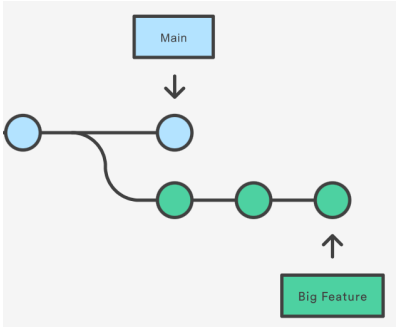
At the beginning of the course, we handed you git and said use this, but only briefly talked about why… I thought this definition from GitLab nicely captures the motivations. Version control systems (VCS):

- Protects our source code from irreparable loss
- Make us more comfortable making changes
- And increasingly serves as infrastructure for collaboration, testing, and deployment. For example, we will use git to facilitate deployment of our application.

That you use a 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 branches to isolate changes until integration
- The “main” branch remains deployable

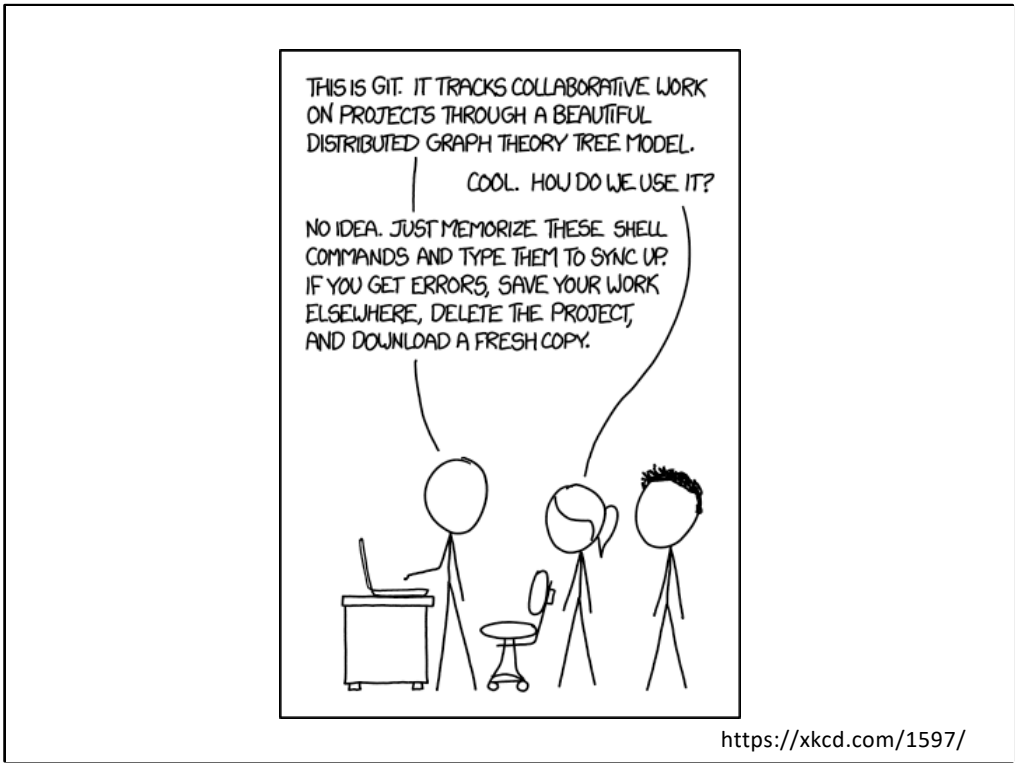
There are many Git workflows. We will adopt a “feature branch” workflow that suits our kind of project(s) and infrastructure. In this model…

We are taking of advantage that in Git, branching is ”cheap”, i.e., it is easy for us create a branch from the current state of the codebase that allows us to isolate changes (from each other, from the current deployable version of the application, etc.) until we are ready to integrate. By creating a new branch, the main branch remains unchanged, and thus in its deployable state, until the new feature is ready and deployable.

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).

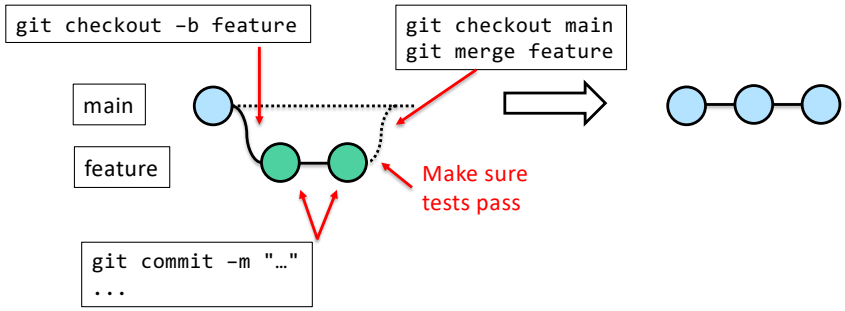
[at end]
A key attribute is “short lived”. This workflow is most effective when branches only live for a brief period (as opposed to days or weeks) and most feature development can start from main, as opposed to another feature branch.

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



To this end checkout the links on the course page with various cheat sheets of the common commands. As with many things we will only scratch the surface of git. Like any complex tool, the best way to learn is not to bite it all off at once but try to continually learn new techniques/tricks as you go and encounter new situations. It is OK to just use same handful of commands (that is what I do!)

Git “solo” branching workflows

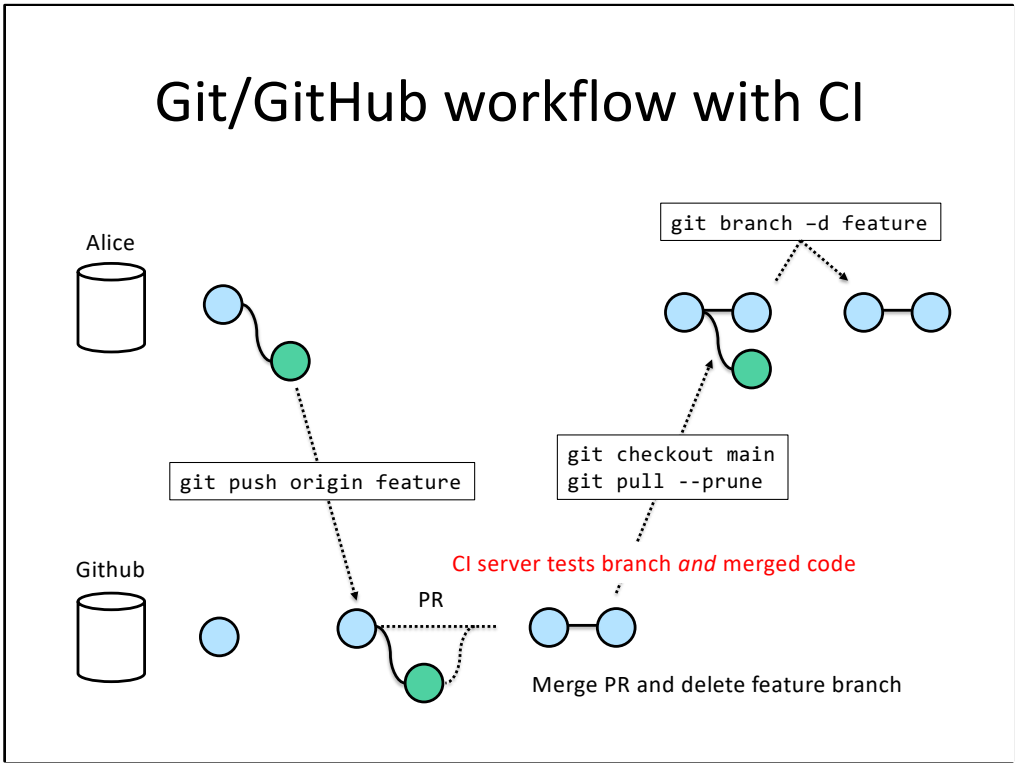


Steps:

1. Create branch with `git checkout –b <branch name>`. Any subsequent changes will be isolated from main (we could switch back to the main branch point at any time)
2. Make one or more commits on our feature branch
3. When we are ready (e.g., tests pass), merge those changes back into main.

The main branch now looks like the picture at right, i.e., it has all of the commits, including those originally made on the feature branch…

I encourage you to work this way, even if you are working totally alone (i.e., no team, no GitHub), that is the ability to maintain history and isolate changes is valuable, even if you are working by yourself. And it is that much more valuable when working with a team, implementing DevOps processes, etc.



Let's add CI into the mix… Now we push our feature branch to GitHub and use its tools, specifically a pull request or PR, to implement the merge.

We use the PR to get other set(s) of eyes, both automated and human, on our changes. When we create the PR, which is exactly what its name suggests, a request to merge one branch into another, we can configure GitHub to test the code and especially test the eventual result of the merge. Often, we incorporate a similar human check, that is a team member(s) must review our code before merge. If everything looks good, we perform the merge on GH. Note that because git is distributed, merging the PR on GH doesn't change our local repository. We must pull the change. Adding the `--prune` option to pull, cleans up references to remote branches that no longer exist on the remote repository (e.g., the feature branch we deleted after the merge) and the `branch –d` deletes the local copy of now irrelevant feature branch.

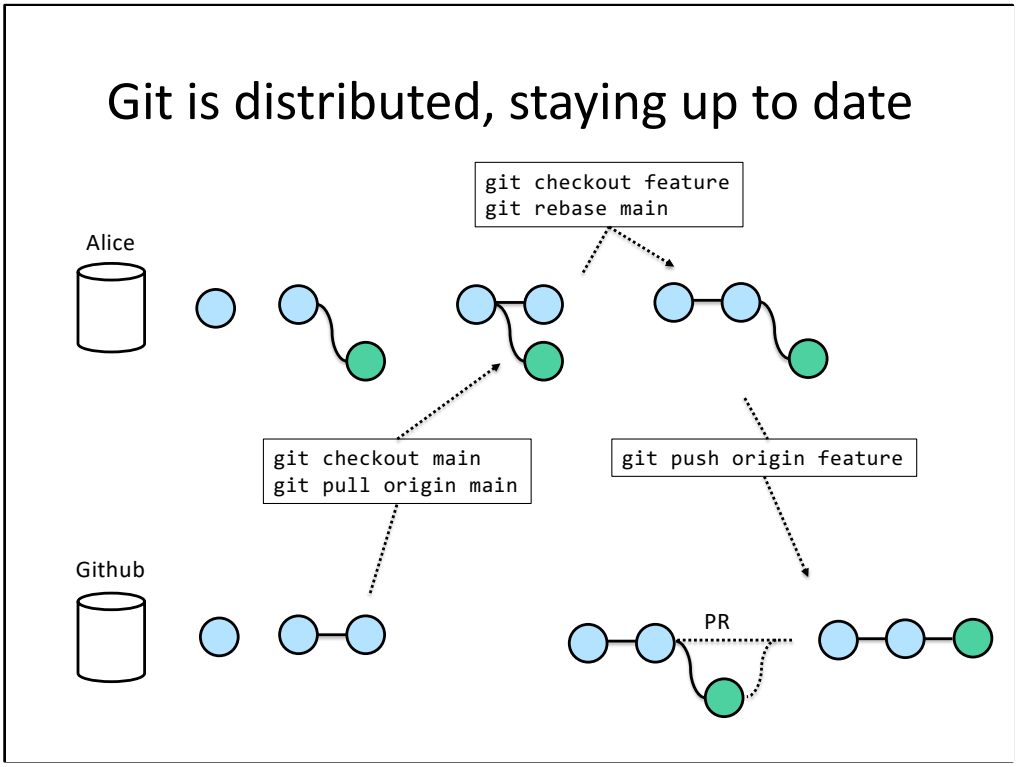
What happens when you need to make changes to your PR after you have created it? You can make additional commits and push to the remote feature branch (i.e., push the feature branch in GH). New commits will automatically be added to PR.

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 NO!!
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.



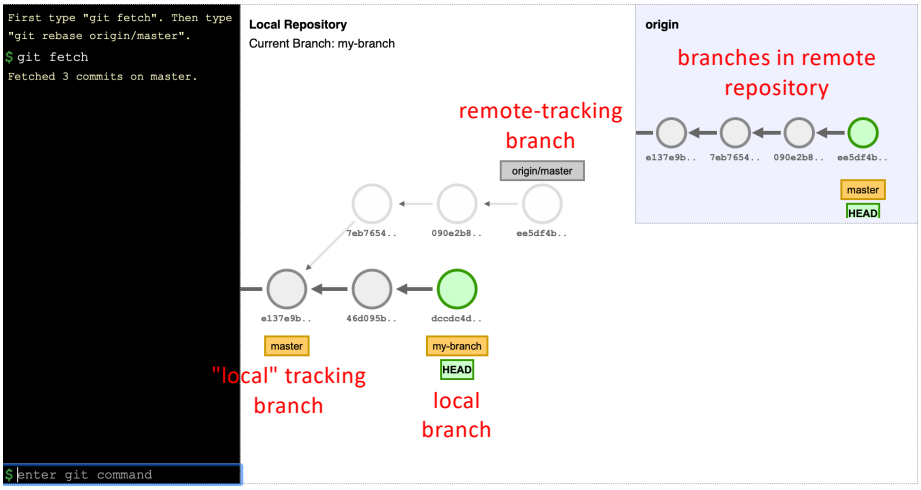
Let's imagine that since you created your branch, your teammate has integrated a new feature of their own. Now main on GH has commits that you don't have. Our first step is to get any new commits to main, bringing our local branches up to date with GH. We then need to combine the two sets of changes.

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 and automatically without overwriting anything that has happened to main since the feature branched off). What if not? We will need to resolve any conflicts manually, that is tell git what the result of the combined branches should look like. We will talk more about this in a moment.

Here we use rebase instead of the merge command we saw previously. From the picture, what does rebase do? Apply Alice's changes "after" previous changes, thus making it appear the branch was created _after_ the most recent changes. Some teams will prefer this approach as it makes the history appear more linear than it was practice (when you look at the sequence of commits). Rebasing is optional and can be fraught (stay tuned). We can also just use merge in this situation.

At this point we are back to our previous situation and are ready to complete the PR process.

git branch vocabulary



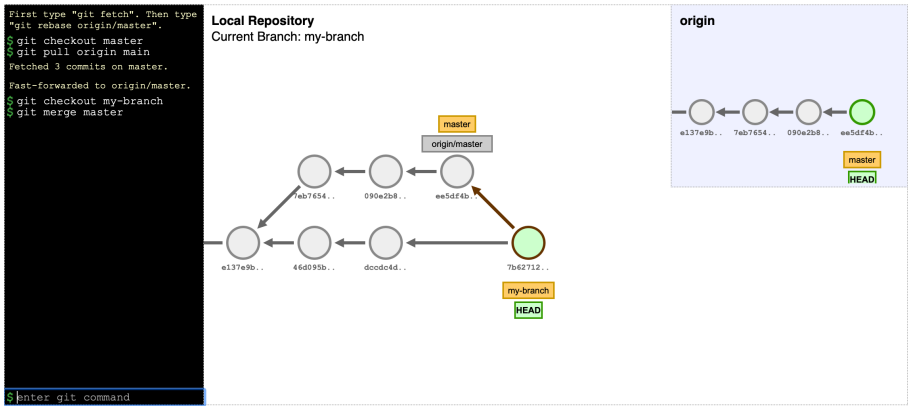
https://onlywei.github.io/explain-git-with-d3

"Remote-tracking branches [like origin/master] are references to the state of remote branches. They're local references that you can't move; Git moves them for you whenever you do fetch (explicit communication), to make sure they accurately represent the state of the remote repository."

"Checking out a local branch from a remote-tracking branch automatically creates what is called a "tracking branch" (and the branch it tracks is called an "upstream branch"). Tracking branches are local branches that have a direct relationship to a remote branch. If you're on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in." Unlike remote-tracking branches, these are branches that can change.

- These branches are typically created with `git checkout -t remote-tracking branch`, e.g., `git checkout -t origin/feature`. That command is typically used to start working on a branch created by someone else on the remote repository.
- When you use `git push -u` that is also creating one of these links
- If you look closely as this example, this state resulted from `git fetch`. That command is used to update the remote-tracking branches alone. It didn't change any local branches. `git merge`/`git rebase` and `git pull` (which combined `git fetch` and `git merge`/`git rebase`) is needed to update the local branches.

Trying out git: Visualizing branches



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

Notice that since our last sync with the remote origin repository we have added new commits on the `my-branch` feature branch and someone else has also pushed new commits to origin,

git checkout master
git pull origin master

git checkout my-branch
git merge master

This is an example where we performed a merge of commits to main (master) pushed to main since we created our feature branch. Unlike our previous example where we used `rebase` to make it appear as though we created out branch after the new commits, here we use merge. Notice that creates a “merge commit” that shows how those to independent histories should be combined.

I encourage you to play with this tool to explore different Git scenarios.

https://onlywei.github.io/explain-git-with-d3/#fetchrebase

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

Assume we have tried to merge main into our feature branch (or rebase our branch with main) to bring it up-to-date 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 must 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 from 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 commit 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. That is the merge is not

complete until you create that new commit. A common mistake is to start the merge and forget that you are in the halfway state and continue on with your feature work. While git is fine with that, it will be confusing to anyone trying to follow the history, as the merge commit is now both a resolution to the merge conflicts _and_ new code.

You would follow a similar process if rebasing, i.e., creating a commit with all merge conflicts resolved, but then you also need to run git rebase --continue

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

Create a merge conflict.

```
mkdir git-merge-test
cd git-merge-test
git init .
echo "this is some content to mess with" > merge.txt
git add merge.txt
git commit -am "we are commiting the inital content”

git checkout -b feature
echo "totally different content to merge later" > merge.t
xt
git commit -am
"edited the content of merge.txt to cause a conflict”

git checkout main
echo "content to append" >> merge.txt
git commit -am"appended content to merge.txt”

$ git merge feature
```

Hint: you probably want the default commit message for the merge. You can use git commit –no-edit to use the message without any changes. If you just use git commit, you’ll probably end up in vim. To get out of vim and save the current commit message, type ”:wq” and hit “enter”.

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

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

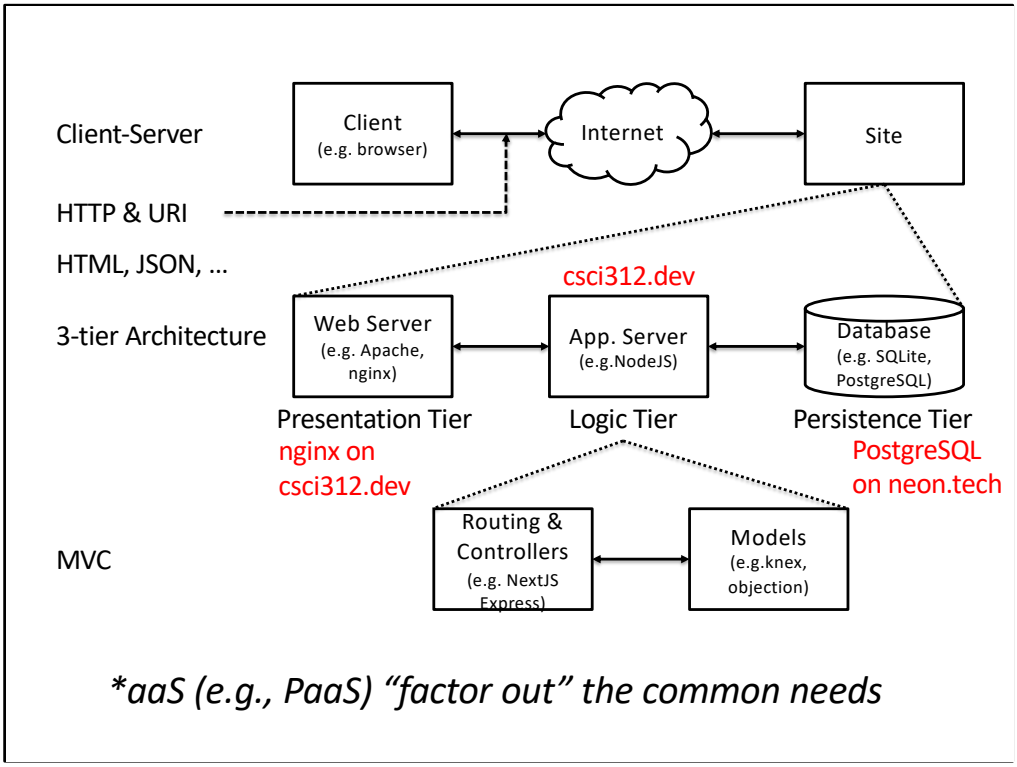
A. *O(1)*: Just one-time efforts to add resources
B. Sublinear: There will be additional work required as a function of service size
C. *O(n)*: The effort will have to grow linearly with demand
D. Greater than *O(n)*: Increasing scale means increasing complexity

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 also 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 is an approach not a task (it is about integrating operations tasks into development) and thus not necessarily a distinct job, the role of site reliability engineer (SRE) is close 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."



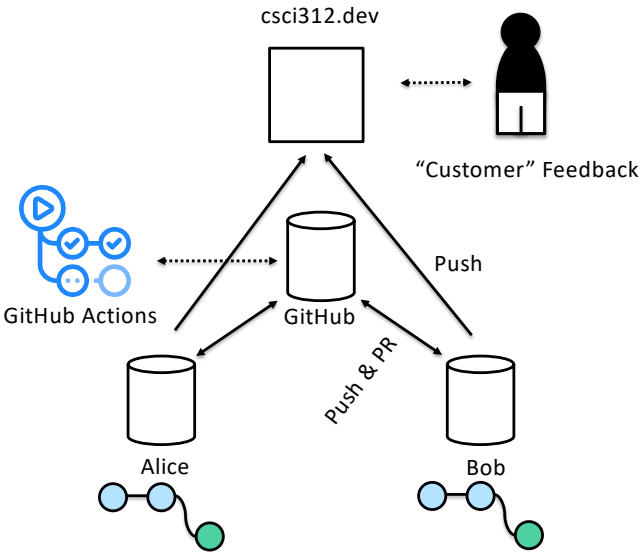
As described previously the 3-tier architecture is a design pattern for building web applications. Recall that a design pattern is a "template" for the aspects of a solution that are the same every time. We can often factor out those common elements into a library or service. Platform-as-a-Service (PaaS) offerings factor out the common elements of that architecture into a service that helps us achieve the scaling we just described. For example, the Heroku PaaS provides the "presentation tier", the "persistence tier" and the portions of the "logic tier" that wrap around your specific application.

This semester we are going to be using a cloud-based in-house PaaS, csci312.dev, and commercial databases-as-a-service offerings to deploy our applications. Like Heroku, with csci312.dev, we push our code to the service via git. The service then builds and automatically deploys our application. Why did we roll our own? It wasn't by choice … we would like to be able to use commercial services such as Heroku, fly.io, etc.. Those services often have free tiers that out more than adequate for our needs, except they require a credit card for identity verification (I didn't want anything in class to depend on a credit card) and often have limits on the size of teams that are prohibitive for us (30 developers looks like large company, with attendant large-company pricing). Fortunately for any personal projects where those constraints are less of a concern, you have many more options!

Automation is not just the province of large applications/companies. *aaS and the

cloud has eliminated most or all physicality from the operations process but also change the dynamic from provisioning (and decommissioning) resources, e.g., servers, infrequently to doing so frequently, thus forcing automation even by otherwise small-scale users.

High-level project workflow



Often projects will be setup to deploy directly from the shared repository, e.g. GitHub. We will not do so, although we could if we wanted to. In our approach a team member will deploy from their local repository, but they should only deploy the main branch as currently exists on GitHub. We model test this out, albeit solo, in our practical today. You will extend and then deploy an application, the color picker, to csci312.dev using the same processes as you will use for the project.