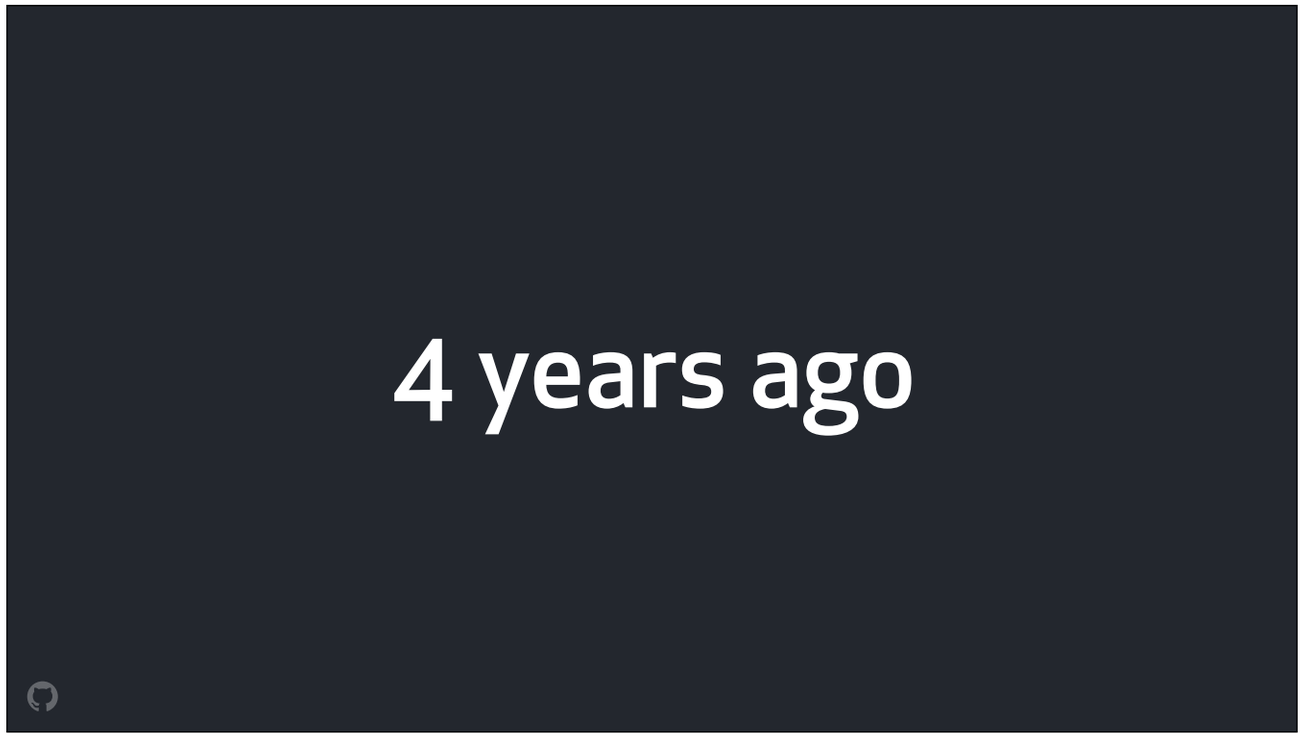
Kubernetes at GitHub

Jesse Newland
@jnewland
Principal Site Reliability Engineer

Good morning everyone! I'm Jesse Newland, a Principal Site Reliability Engineer at GitHub. I'm super excited to be able to share a glimpse into GitHub's Kubernetes infrastructure with you today.

I'd like to start with a quick story ... a story about a software project that never got off the ground.

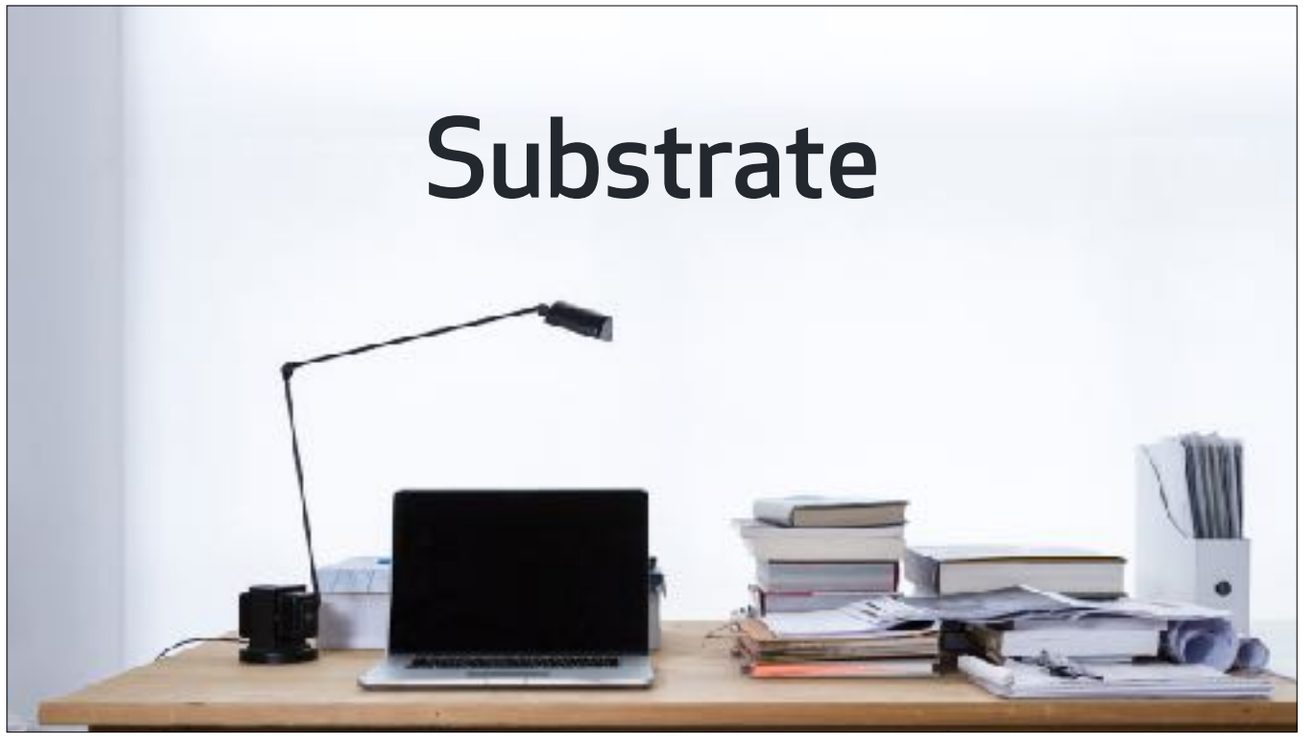


Around 4 years ago - in the fall of 2013 - the GitHub Ops team all met in San Francisco. We'd just finished a big datacenter migration, which had bought us a **ton** of **time** in our **race** to keep up with **traffic, user growth, and DDoS attacks.**

And as a team, we were excited about this newfound time; we were full of ideas about moving up a layer of abstraction. So at this team meeting, we started talking seriously about building a platform.

A handful of engineers on the team had experience building and running platform as a service offerings, and had a design in mind that used containers instead of VMs. We sketched out some of the rough concepts, and split up some follow up tasks.

The team all went back to their home offices, and started researching the problem space. We picked a…



…name and started to get serious about the work. But as we learned more about state of container tech at the time, we started…

to get a better sense of the amount of work ahead of us. There was a lot to do. We started to realize that it was gonna take us quite a bit of time to deliver something that solved the problems in our sights

And as we started comparing notes with others in the org, we got feedback from a few senior engineers that was initially surprising

They encouraged us to wait. To do nothing.

Let someone else build this, they said. The design's not bad, in fact, it's good! But it's **worthless** without **execution**. And we're not in the right spot to ship something like this at the moment.

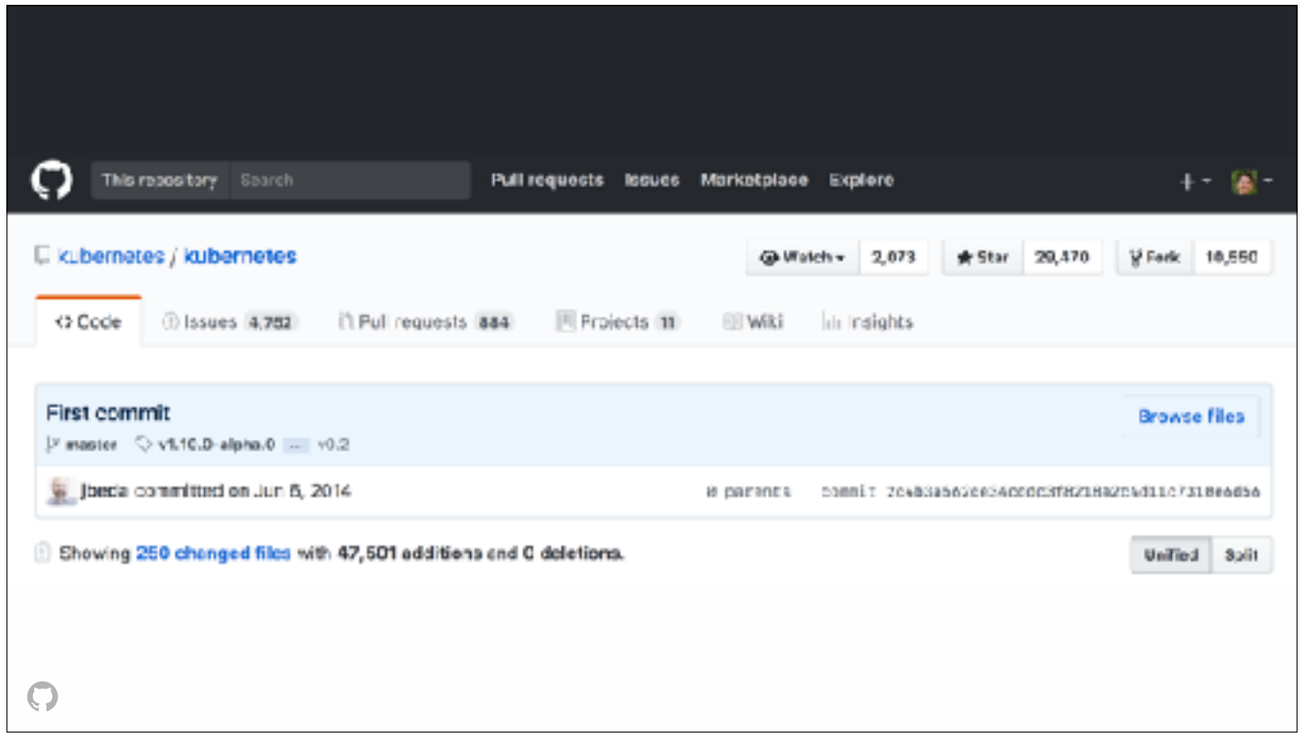
This project needs a team that has more experience with containerized workloads, and the resources necessary to refine that tech a bit further. This project would be better served by an organization with more experience managing long running projects. We want to see this happen, they said, but we're not the right group of folks to do it, especially now it right now. Instead, we need to focus on leveling up our organization, as well as our tech for storing Git repositories.

We took that feedback to heart, and agreed to call the project off

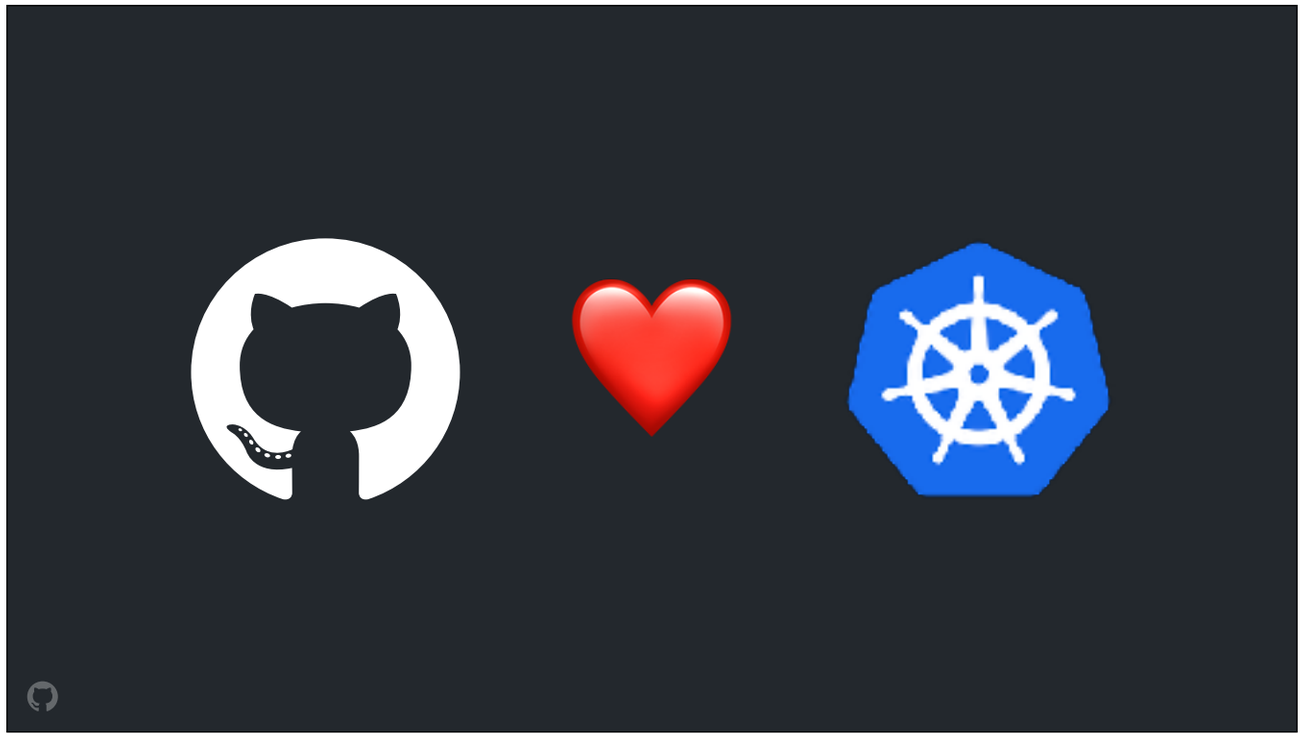
I was disappointed, as were a lot of others on the team. But in retrospect, I think this was some of the best technical advice I've ever had the privilege to receive.

There were a lot of other things GitHub needed to work on: we **really weren't** in a good place to execute on a project of this scope at the time. I'm glad that we focused our time and attention where we did: we needed to grow in other ways.

Now, I didn't tell you this story to make a false claim to any idea of a platform or anything - in fact, quite the opposite! I told you this story to setup a sincere expression of gratitude to all of those that ...



were making different decisions just a few months later, deciding to dedicate the time, experience, resources, and emotional labor necessary to kick off the Kubernetes project.

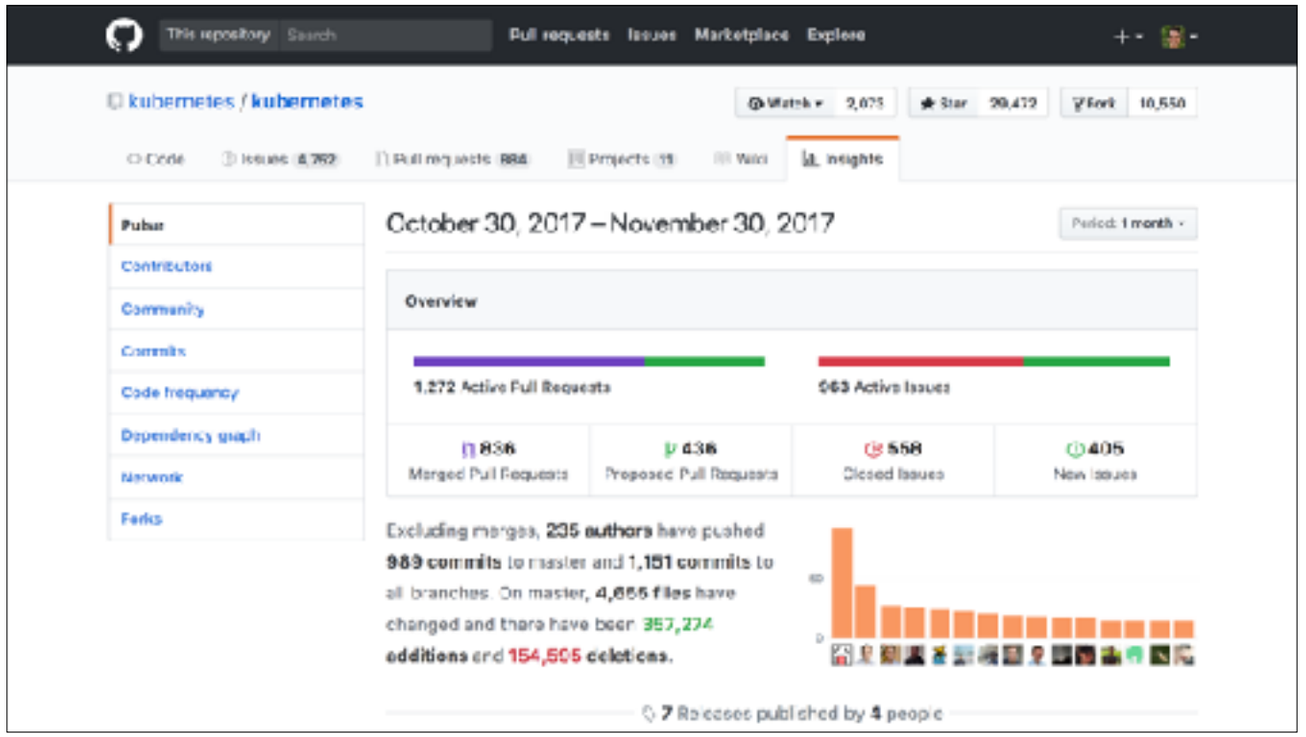


We're incredibly grateful for Kubernetes at GitHub, and want to extend our gratitude to every single member of the community that has formed around it.

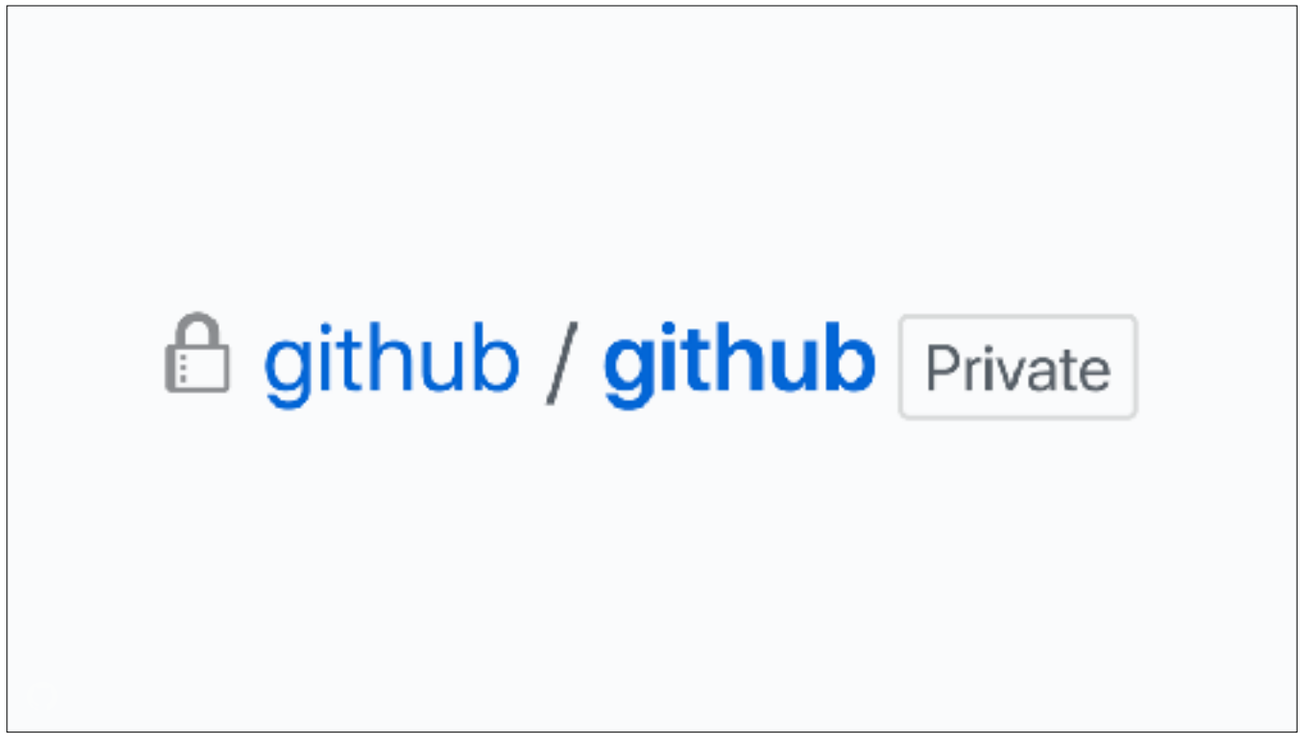
GitHub is standardizing on Kubernetes as our service runtime across our entire engineering organization. We hope to do that over the course of the next few years.

20% of services run on Kubernetes

We're making decent progress towards this goal: right now, 20% of the services we run are deployed on Kubernetes clusters inside our datacenters…



…including the service that serves github.com and api.github.com: the service you use when you open an issue or review a pull request on the Kubernetes repo. I'm excited to tell you more about the Kubernetes infrastructure that powers this one, small, part of the Kubernetes community today.



The Ruby on Rails application that serves GitHub's web UI is also developed on GitHub, and lives at github dot com slash github slash github

That's a bit of a mouthful, so it's become know inside the organization colloquially as

GitHub dot com,
the website

..which I find pretty amusing.

But to our Kubernetes clusters, it has another name

```
$ kubectl get ns github-production
NAME                STATUS    AGE
github-production   Active    168d
```

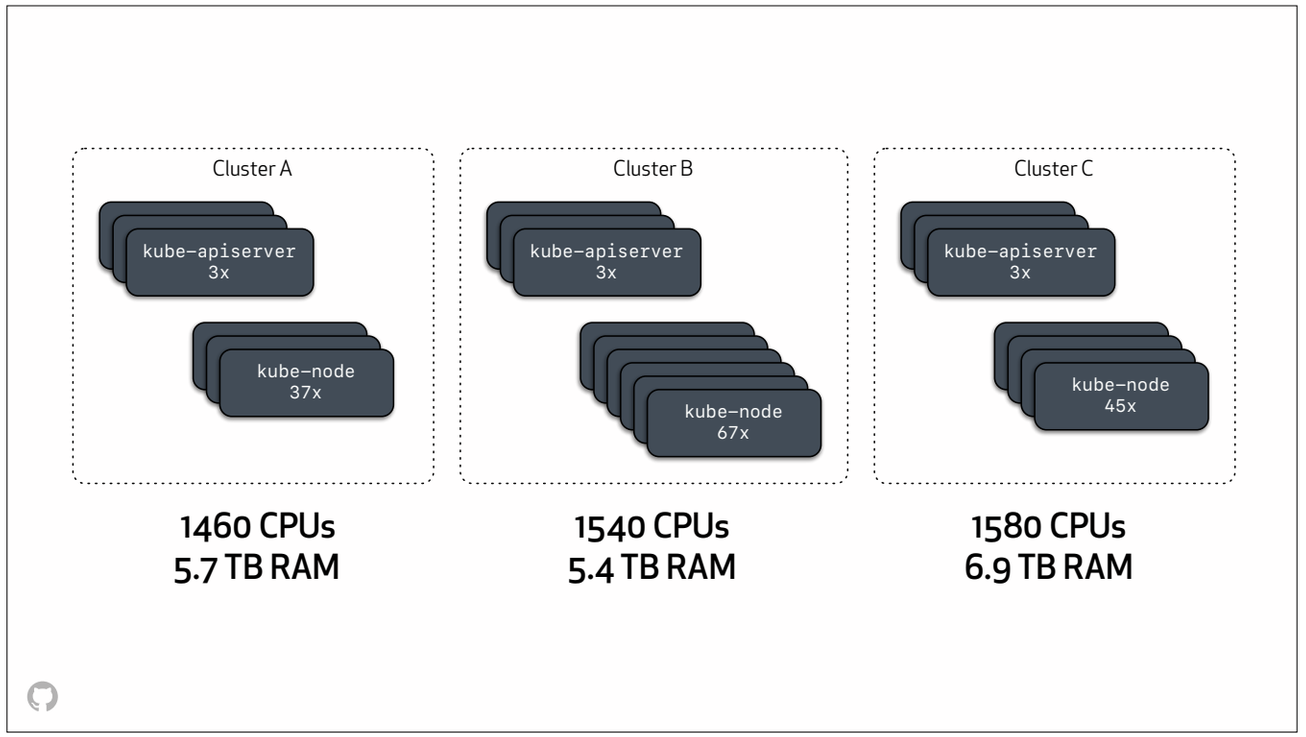
It's the github-production Namespace.

```
$ kubectl get ns
NAME                STATUS    AGE
github-production   Active    168d
kube-system         Active    169d
```

When we started the process of migrating github dot com the website to Kubernetes, we hadn't yet settled on a design for isolating workloads running on the same cluster - and RBAC was still in alpha - so we reserved a few clusters for the exclusive purpose of running this workload and a few of its supporting services. We're getting more confident with our isolation design over time, and are excited to revisit this soon so we can more densely pack our workloads together.

We also didn't have a lot of experience **operating** clusters at the beginning of this project. We knew how to **break** one, though! We'd recently broken a cluster pretty badly while trying to roll out a apiserver config change, and broken another one when we tried an upgrade.

To ensure we could hit our targets for the reliability of github.com, we decided to run its workload on multiple clusters in each site. We didn't end up using any of the existing federation tools to do this, but instead use **our** deployment's tooling's support for deploying to multiple "partitions" in parallel. This multi-cluster approach has been pretty useful recently, as it's allowed us to replace entire clusters without interrupting the regular flow of traffic or work.

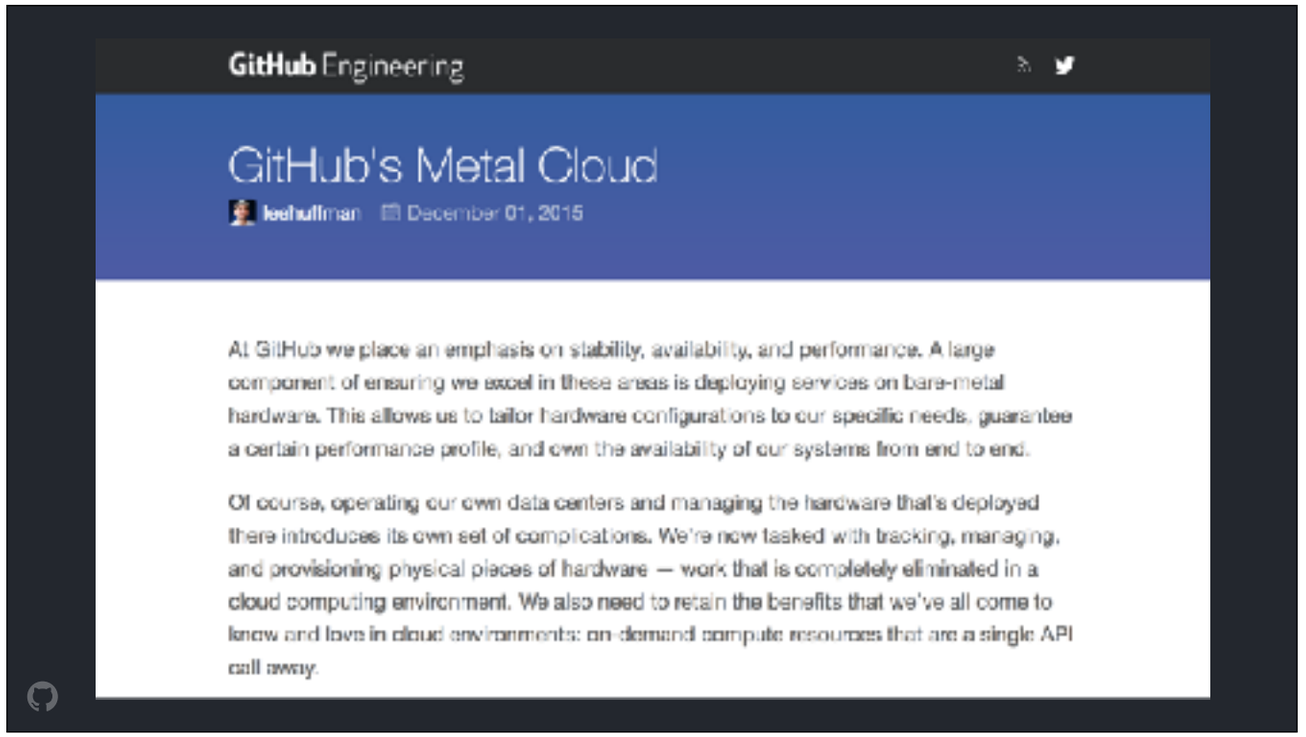


The three clusters running the dot com workload in our primary site provide around 1500 CPUs and 5TB worth of RAM each. We're currently running around 7 of different types of nodes in these clusters, so they vary pretty widely in terms of node count. Running the same workload on a wide variety of node types has helped our datacenter teams evaluate the relative performance and efficiency of a bunch of different types of hardware

We've recently been preferring these tiny little things, in a configuration that gives us 40 CPU threads, 128G of RAM, and two local SSDs.

We pack tons of these things into each rack, and use node annotations to ensure that the Kubernetes scheduler automatically distributes work across racks and other failure domains

Over the past few years, GitHub's invested a ton of resources into our datacenter management processes and practices



This investment has paid off pretty well: we've built out some incredible facilities, and have shipped provisioning APIs and automation that resemble what you might expect from a public cloud providers's VM control plane

Engineers can provision nodes from chat, but instead of getting back a VM, our metal cloud deals out…

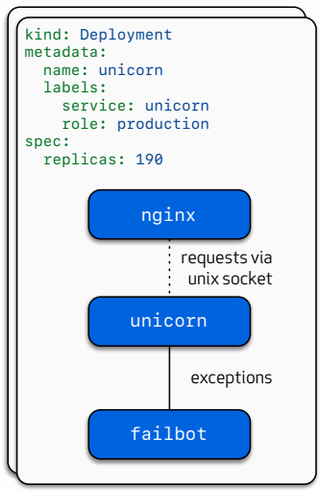
one of these, hooked up to our high performance network.

Node configuration is managed by our existing Puppet infrastructure, which we use to build the images used in the provisioning process. Configuration changes and security updates can be applied with a Pull Request to our Puppet repo, which can be deployed to individual node, a cluster, or the entire fleet.

These metal cloud workflows are pretty well tested, and gave us a great base on which build out our Kubernetes infrastructure.

```
$ kubectl -n github-production get deployment
NAME                    DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
unicorn                 190       190       190          190         168d
unicorn-api             164       164       164          164         168d
consul-service-router   2         2         2            2           168d
```

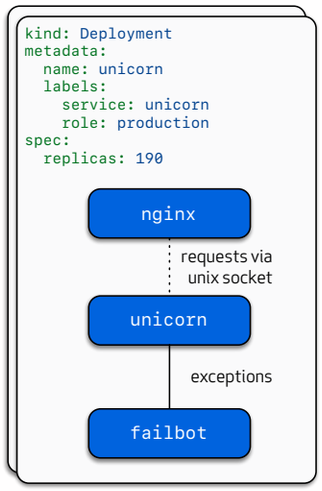
Inside the github-production namespace on each cluster, there are three primary deployments…unicorn, unicorn api, and consul service router.



unicorn

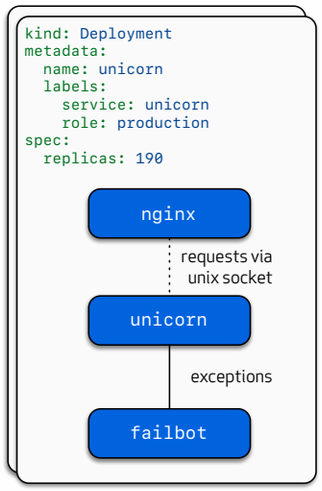
Each pod in the unicorn Deployment contains three containers:

An nginx container that accepts and buffers incoming requests…



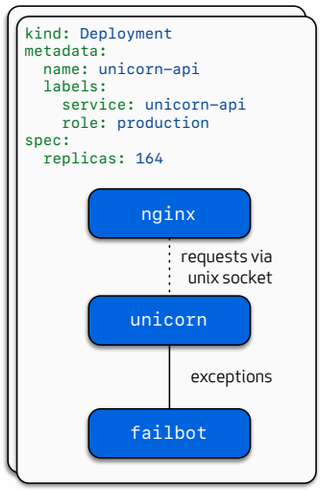
unicorn

A container that runs unicorn, a Ruby on Rails server, which processes requests sent over a domain socket from nginx, makes requests to backing services like My Ess Que El or Spokes, our git storage tier, and then renders HTML…



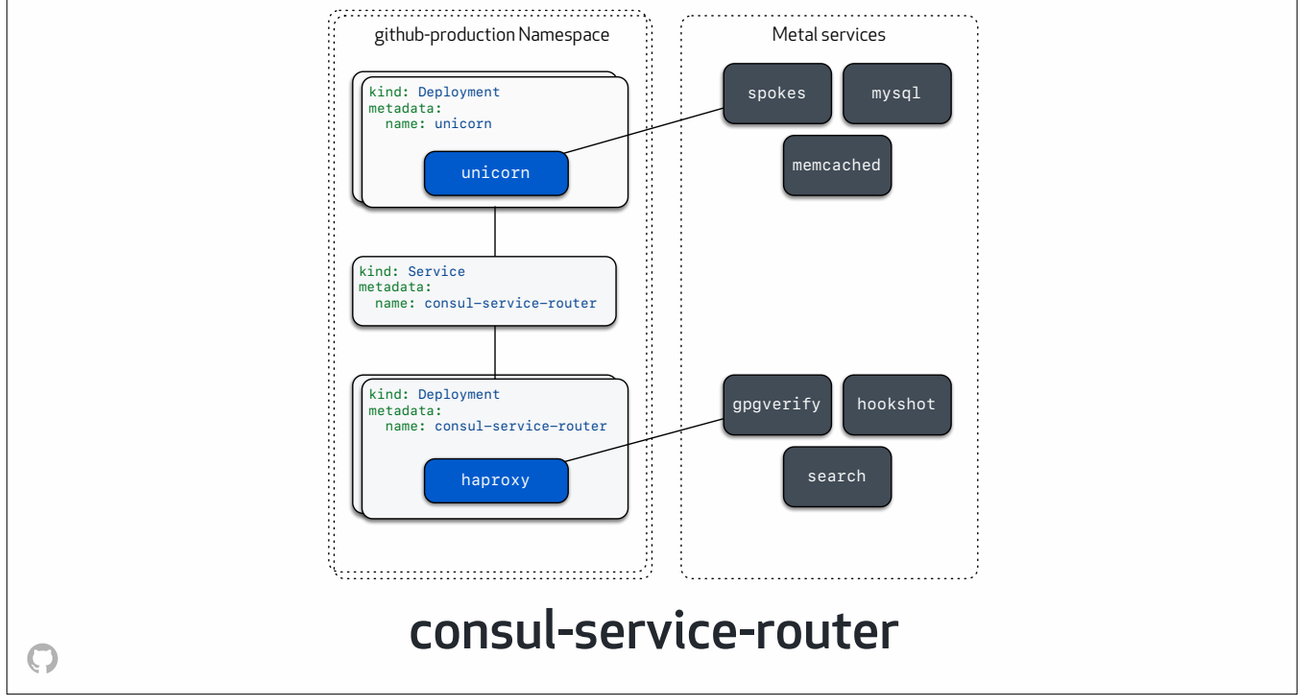
unicorn

And finally a proxy called failbot that accepts, decorates, and buffers exceptions reported by the Rails application before sending them to an exception tracking system

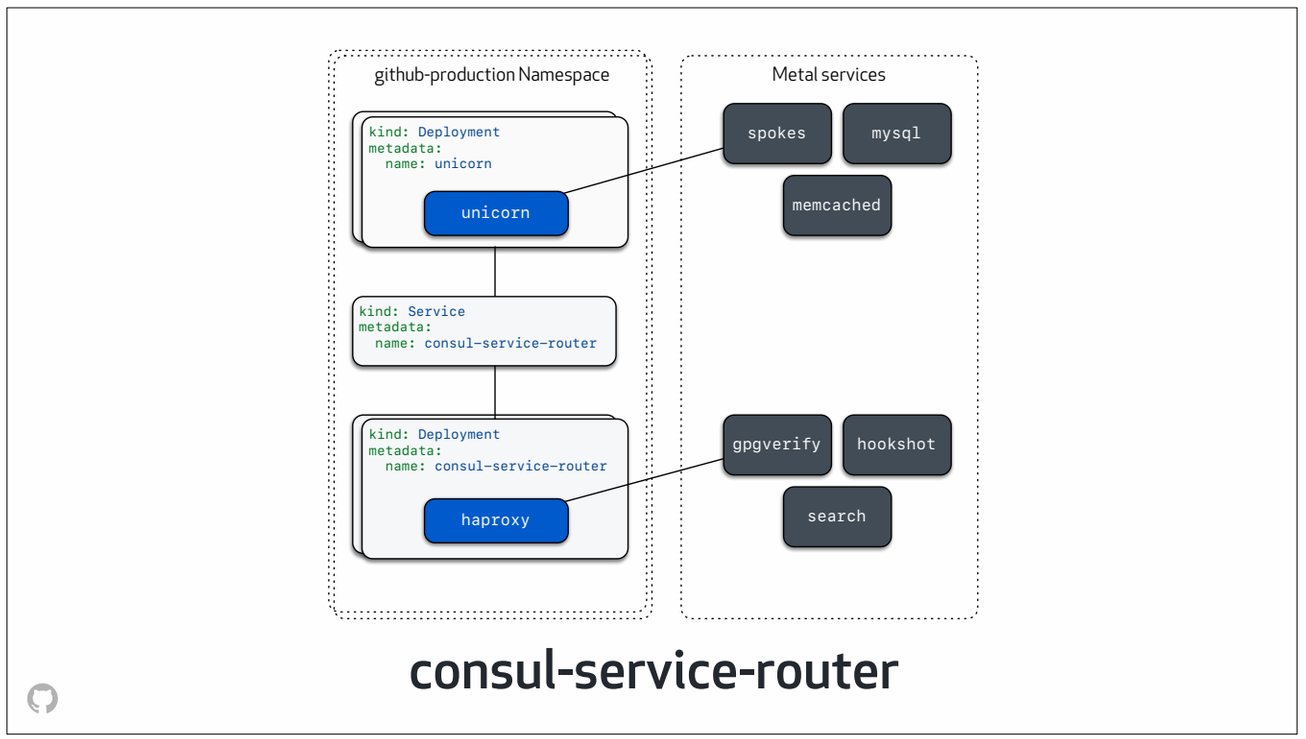


unicorn-api

The **unicorn api** deployment powers api dot github dot com, and is basically identical to the unicorn deployment. We deploy it separately due to some differences in seasonality – computers wake up and go to sleep more frequently than we do, it turns out.

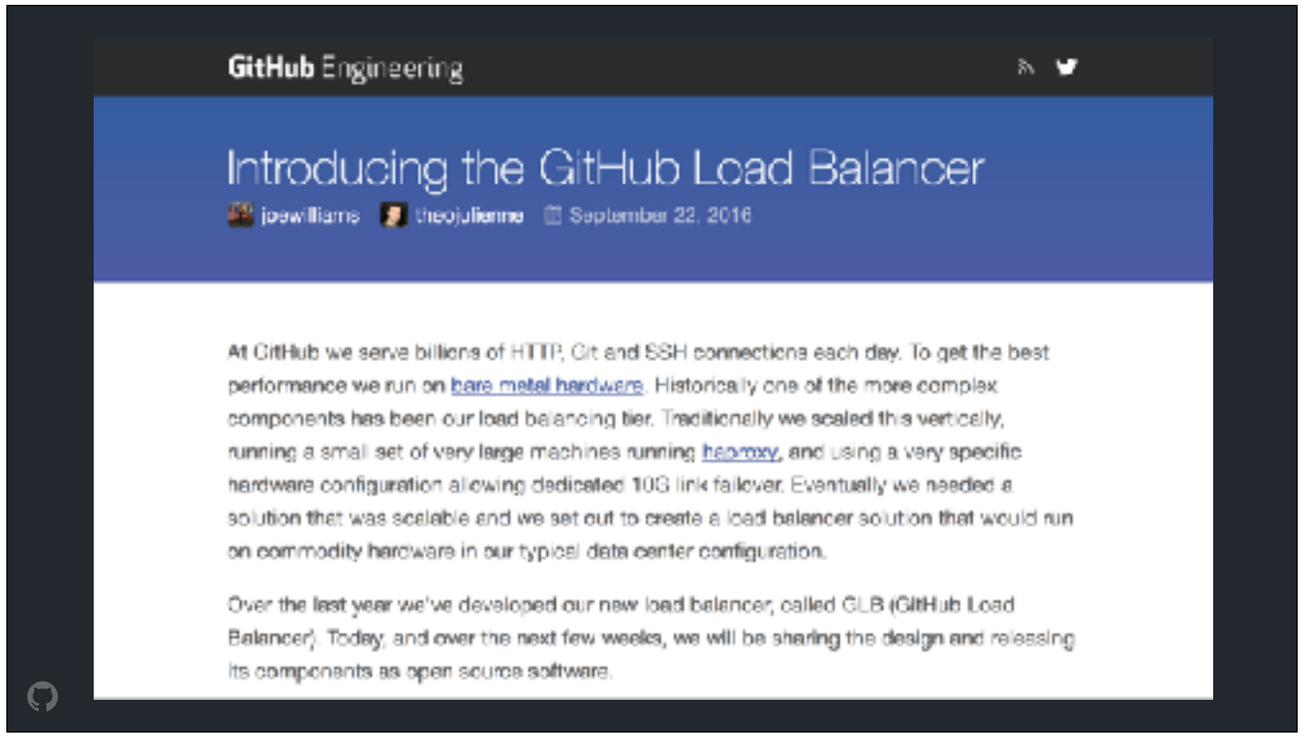


The third deployment that's running in the github-production namespace is a tiny service called consul-service-router, which routes traffic from application processes to services not running in a Kubernetes cluster - like our search clusters or our web hook delivery service - using a combination of haproxy and consul-template

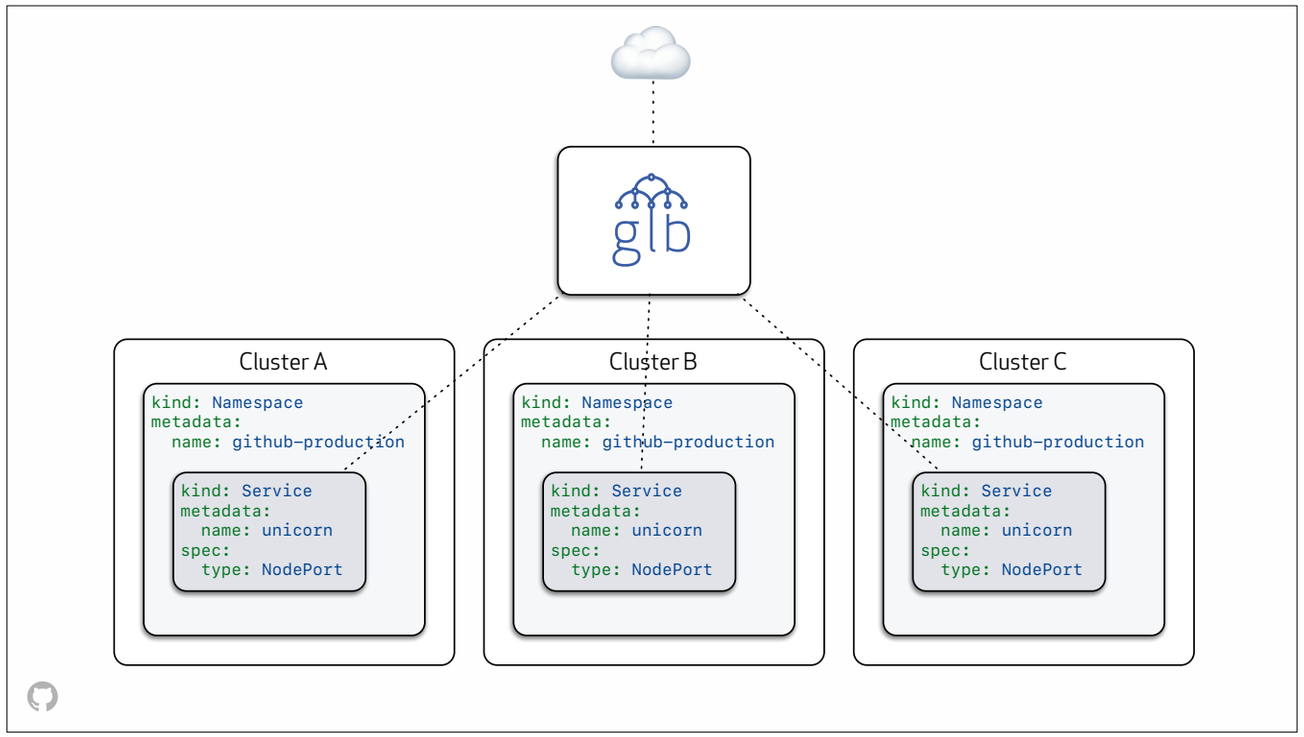


consul-service-router

This service is a direct reimplementation of the approach we used to solve this problem before Kubernetes. We're hoping to replace this, uh, *hand-rolled service mesh* with Envoy sometime soon as a part of a project that aims to improve the observability of **all** service to service traffic.



To accompany our metal cloud service, we've built GLB - a horizontally scalable load balancer service that can route external or internal traffic to a group of nodes.



We GLB along with NodePort services to get traffic into our clusters, currently using every node in the cluster as a backend. Most of the NodePort services configured in GLB route to a ingress deployment, but a few skip that step for various reasons.

Tools to support operations

- kube-testlib
  - Continuously running suite of conformance tests
- kube-health-proxy
  - Adjust weight of incoming traffic, disable entire clusters at load balancer level
- kube-namespace-defaults
  - Creates default resources in each new namespace, configures imagePullSecrets
- kube-pod-patrol
  - Detects and deletes stuck pods, sets NodeConditions if a node has repeated trouble starting pods
- node-problem-healer
  - Detects NodeConditions, heals them by rebooting nodes

In addition to configuring pretty standard logging, monitoring, and metrics stacks on our clusters, we've built a handful of tools to support ongoing cluster operations

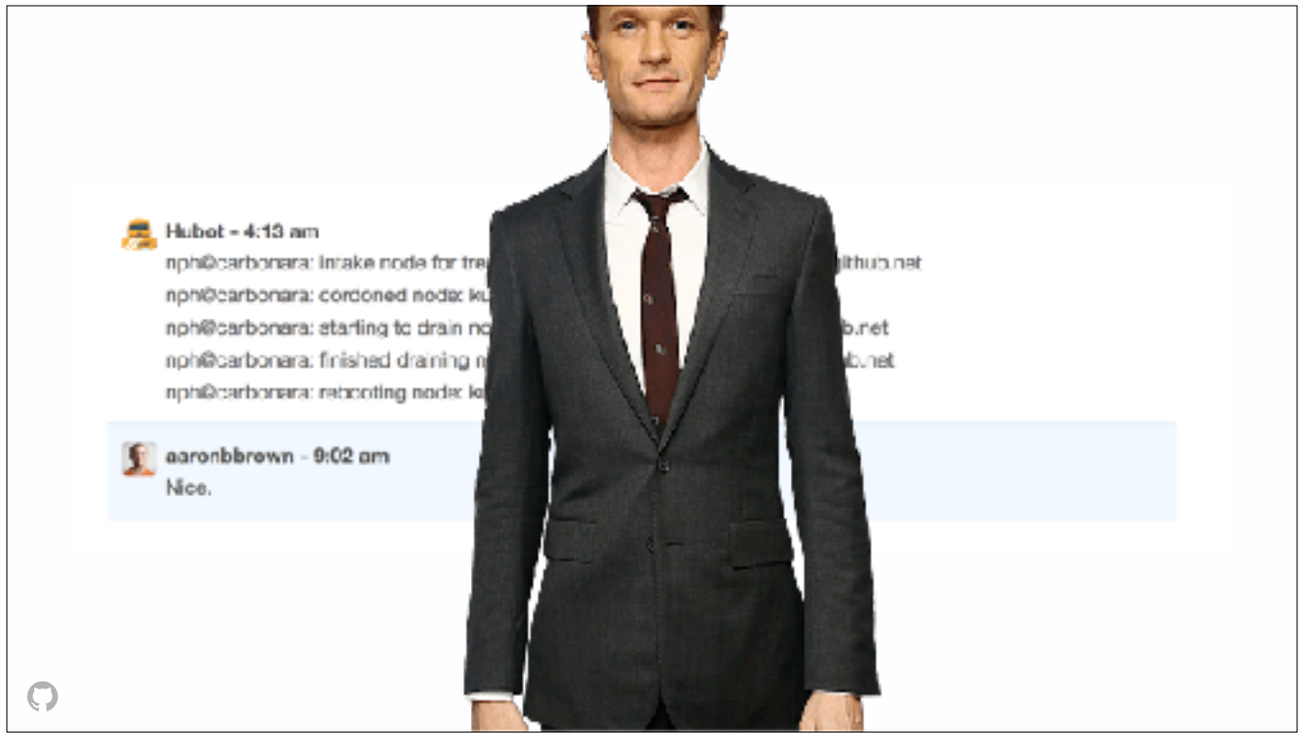
A few of our teams have been building systems on Kubernetes for around a year, and it's been a joy to observe how their workflows have changed over that time. All of these tools that I just mentioned: they're like, real software projects, with tests and stuff! They each have their own repo, which contains the software, a description of how it should be packaged, and a blueprint for running it in production.

A year or so ago, we might have reached for a bash script to solve some of these problems. And to distribute that script to a class of nodes, we might have checked it into our Puppet repo.

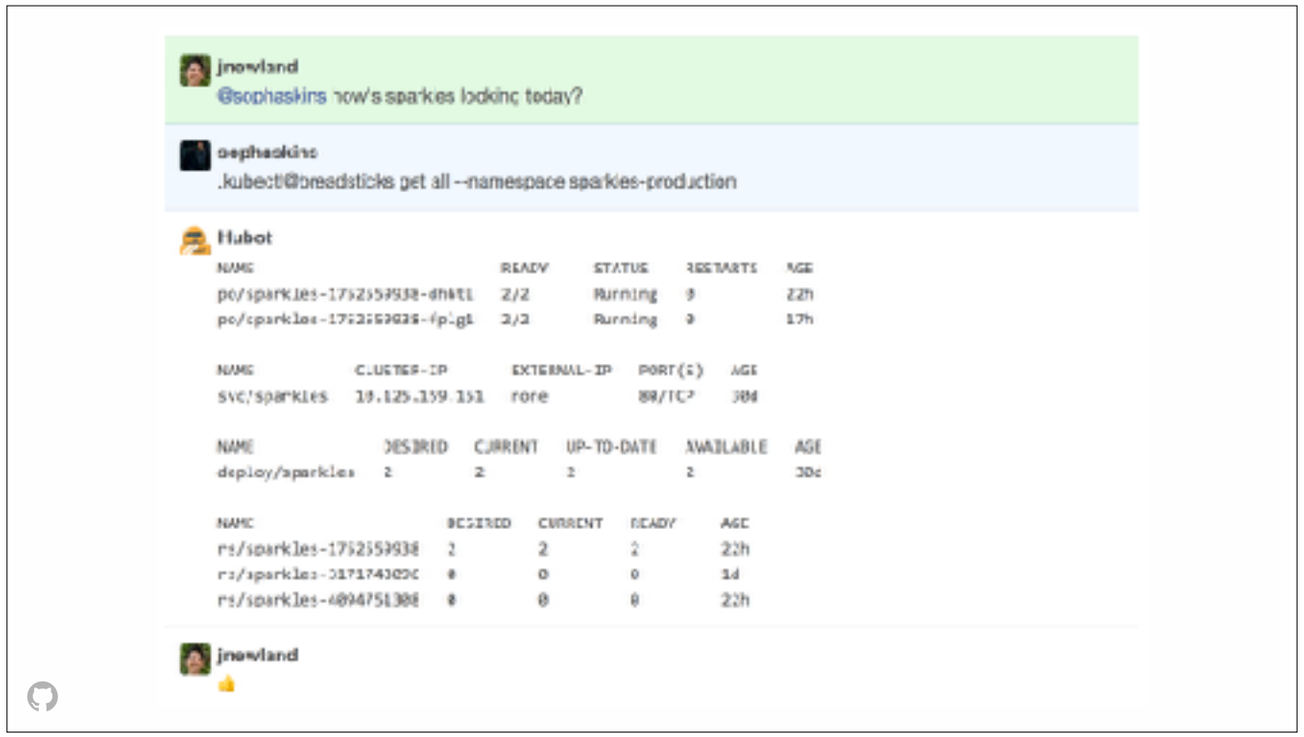
I mean I really love bash, but this workflow left a lot to be desired.

Instead, we've ended up building projects - like Node Problem Healer - that use Kubernetes' incredible APIs.

Node problem healer, in particular, has been a big win for the team, as it fixes a class of problem that sometimes happen in the middle of the night. To be honest, I think it's also been a win because the NPH abbreviation used in it's chat log messages makes us all think about



Neil Patrick Harris more often. I mean, it cheers me up every time I see it.

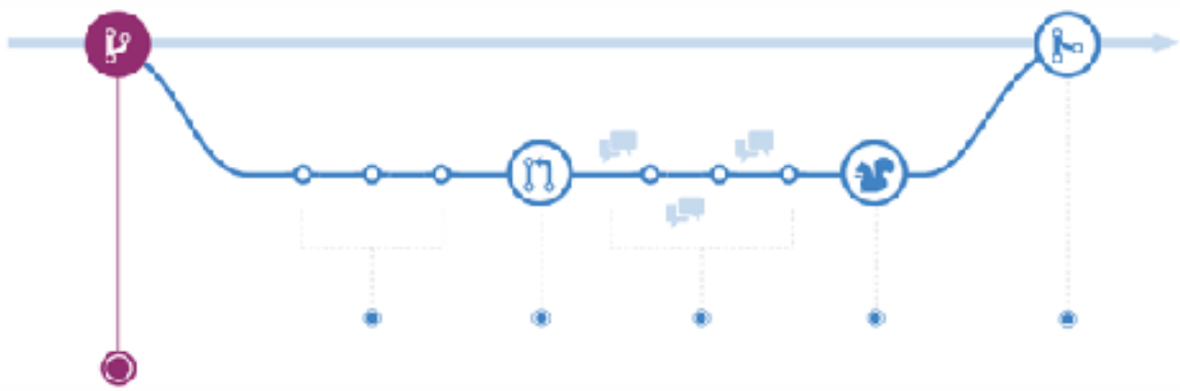


We've also built a tiny wrapper for `kubectl get` that we can run in chat. It's been incredibly useful as we help a bunch of engineers learn how to build systems using Kubernetes primitives.

But these chatops are read only: we encourage engineers to use...

... our deployment tools to declaratively manage the Kubernetes resources that describe their infrastructure as much as possible

GitHub Flow



As opposed to a traditional versioned release strategy, GitHub has historically used a continuous delivery approach called the GitHub flow, where we deploy approved, but open pull requests to production, and then merge them when their behavior has been verified

Conventions

To map this sort of workflow to Kubernetes, we settled on a basic set of conventions, which I'll try to describe with some pseudocode

```
$ docker build -t $service:$sha1 ./Dockerfile
```

First, if a repo contains a Dockerfile, we'll
build that image on each push,
tag it with name of the service and current Git commit,
and then push it to our internal registry

```
$ docker build -t $service:$sha1 ./Dockerfile

$ kubectl create ns $service-$environment
```

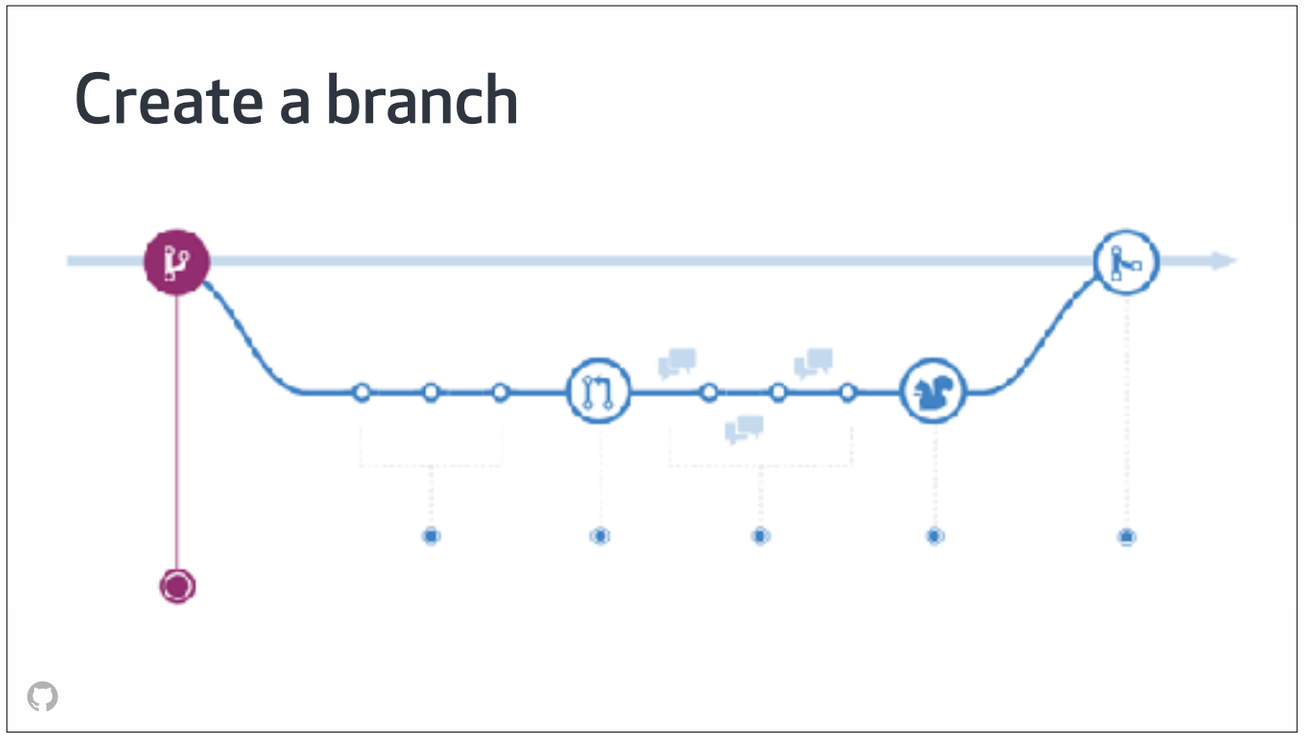
Second, each service and "deploy target" combination gets its own namespace, like github-production for example

```
$ docker build -t $service:$sha1 ./Dockerfile
$ kubectl create ns $service-$environment
$ deploy -Rf ./config/kubernetes/$environment | \
```

Next, we expect our deployment tooling to load and process Kubernetes YAML from config/kubernetes/$environment in the service's repo. This step can modify resources, filter them, or inject additional ones. This step updates image fields to use image built for the commit being deployed, as well as some other actions that are configured with annotations.

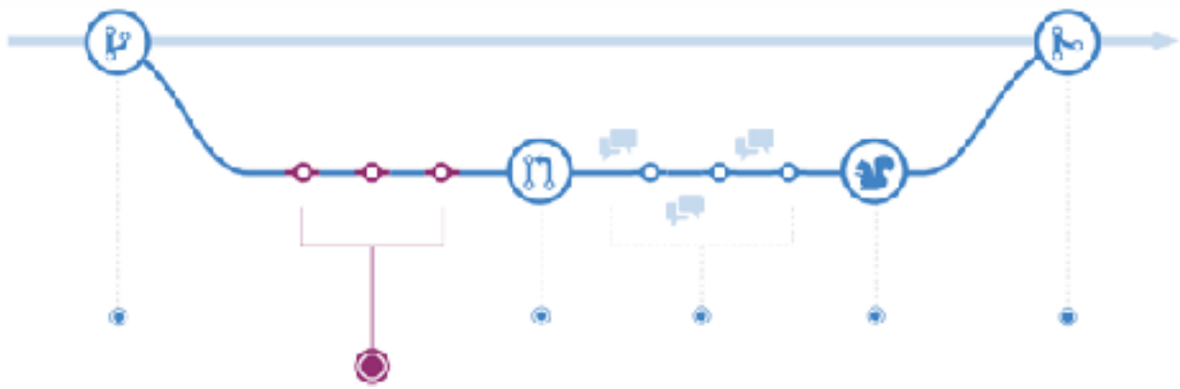
```
$ docker build -t $service:$sha1 ./Dockerfile
$ kubectl create ns $service-$environment
$ deploy -Rf ./config/kubernetes/$environment | \
  kubectl -ns $service-$environment apply -f -
```

Finally, we expect our deployment tooling to declaratively apply the modified resource config using the same semantics as kubectl apply

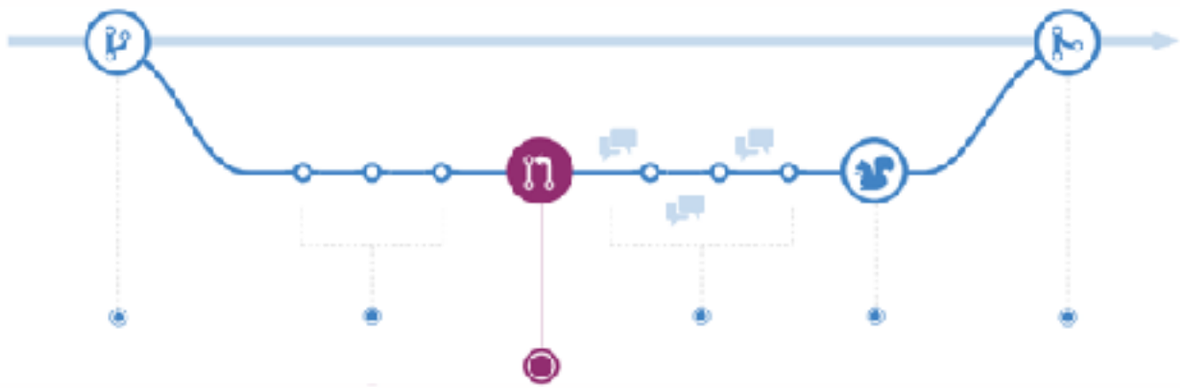


What's really rad about this set of conventions is that they support changes to either the application OR it's kubernetes configuration using the exact same workflow. So no matter what sort of change you're looking to make, the workflow looks the same …

Add some commits



Open a pull request

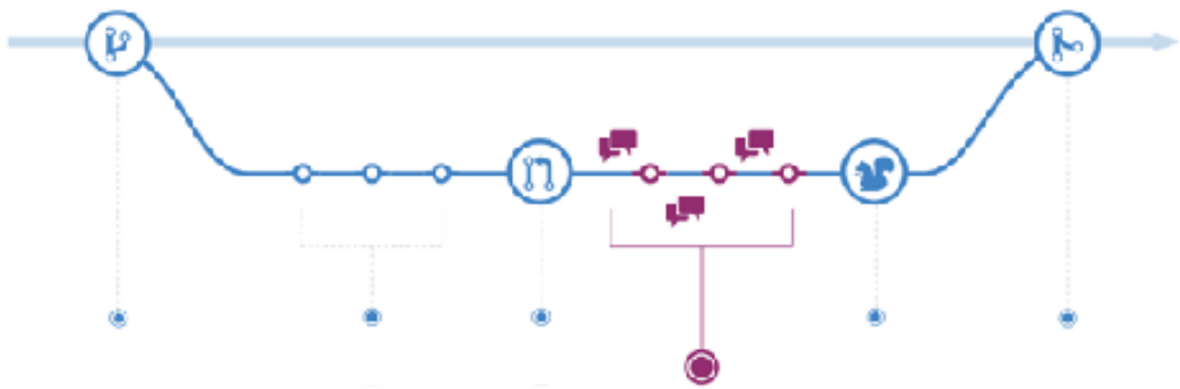


Then push your changes up to GitHub and open a pull request.

Containers built on push, tagged with commit

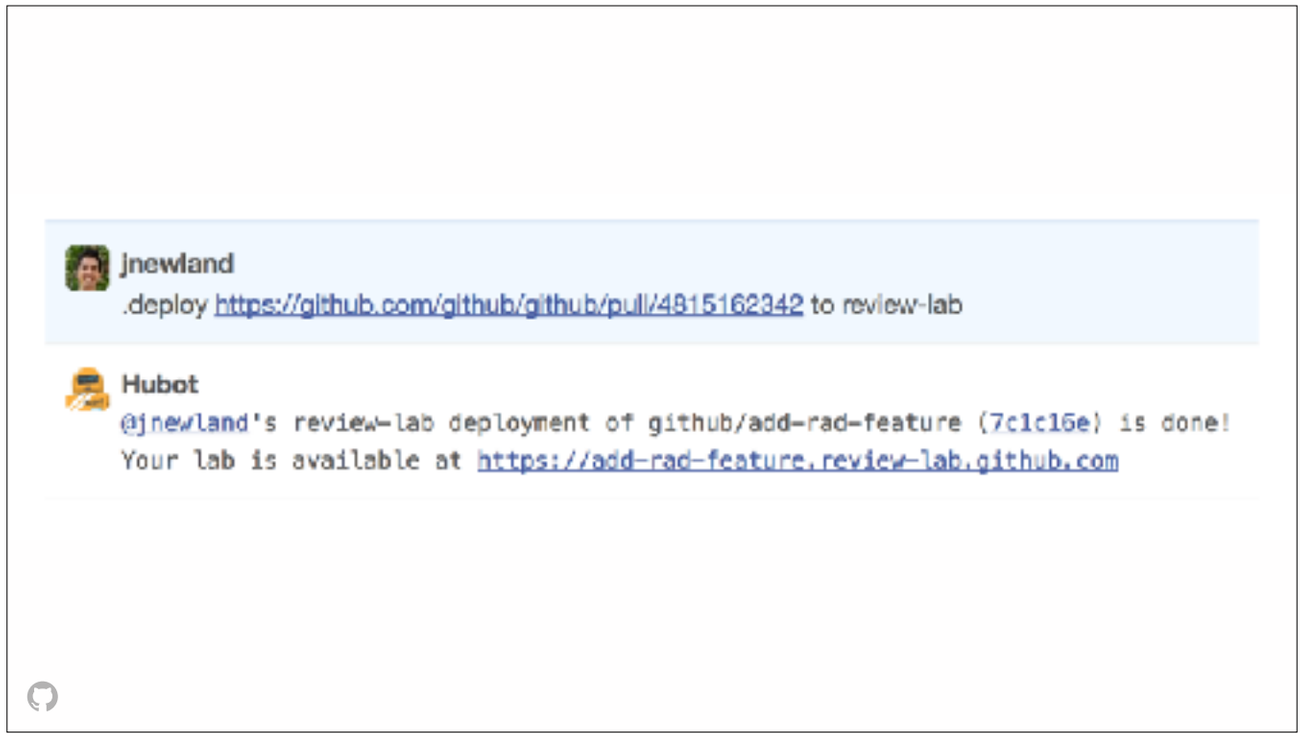
On push, a CI job will build a container image, tag it with the current commit's sha, and then push it to our internal registry

Iterate and review



Next up is everyone's favorite part of pull requests. Code review!

During the course of reviewing a pull request, it's often helpful for both feature authors and reviewers be able to preview the effect of a change, **especially** if that change targets UI elements



To help with this, we support deploying to a “review lab” target. Each review lab gets a unique URL that any GitHub engineer can log into, named after the branch that’s being reviewed.

```
# config/kubernetes/review-lab

# updates image field value to $service:$sha1

# injects a Secret

# injects an Ingress
```

During a review lab deployment, our deployment tools load the YAML in config/kubernetes/review-lab and make few modifications. Image fields are updated with the latest commit sha to pull an up to date image, a secret is injected using values from our internal, encrypted secret store, as well as an ingress resource that configures the branch-specific subdomain.

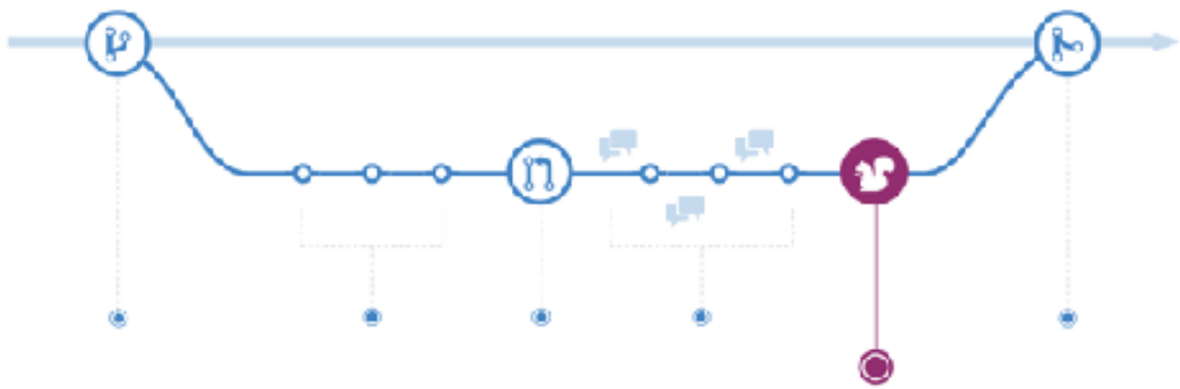
```
$ kubectl create ns review-lab-$branch

$ kubectl apply -ns review-lab-$branch -f -
```

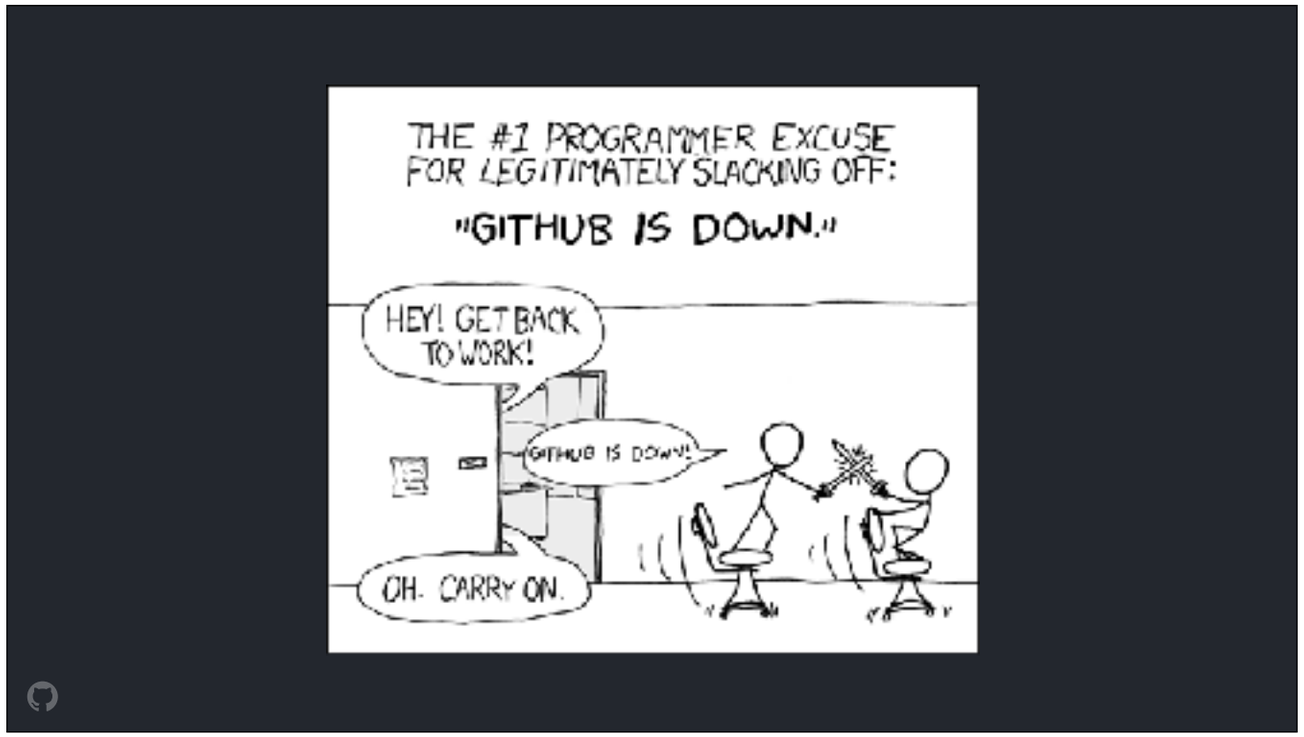
Our deployment tooling then creates a namespace named after the branch you're deploying and applies the modified configuration in the new namespace. This workflow brings up a an entirely isolated frontend tier for each branch that needs to be reviewed, which is automatically cleaned up after 24 hours of inactivity.

We're really enjoying this workflow, and hope to bring it to other apps soon.

Deploy



Once your PR has been reviewed and approved, it's time to deploy. Deploying a change to a large system like GitHub is inherently risky, and we're always looking for ways to reduce that risk.



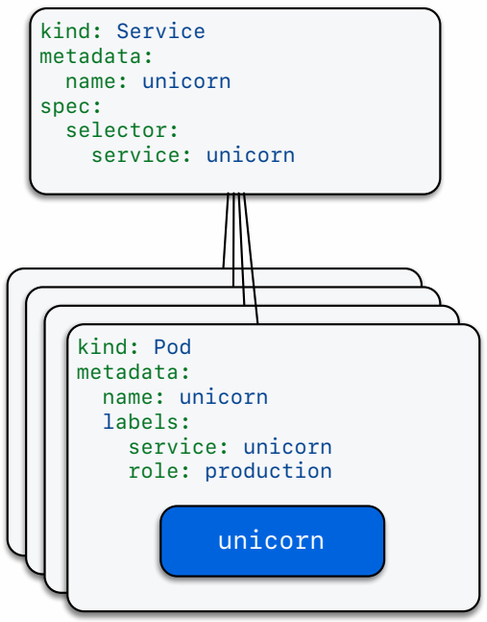
We know that GitHub going down can have a significant impact on the entire software industry: workflows are interrupted, release schedules are affected, and all sorts of productivity is lost. So after looking closely at the factors that contributed to outages in the past, we added another step to our production deployment pipeline a few years back

The first step of a production deploy is now a **canary deploy**, which exposes your change to a small percentage of traffic to help you confirm that it doesn't materially impact the latency or error rate of requests flowing through the system

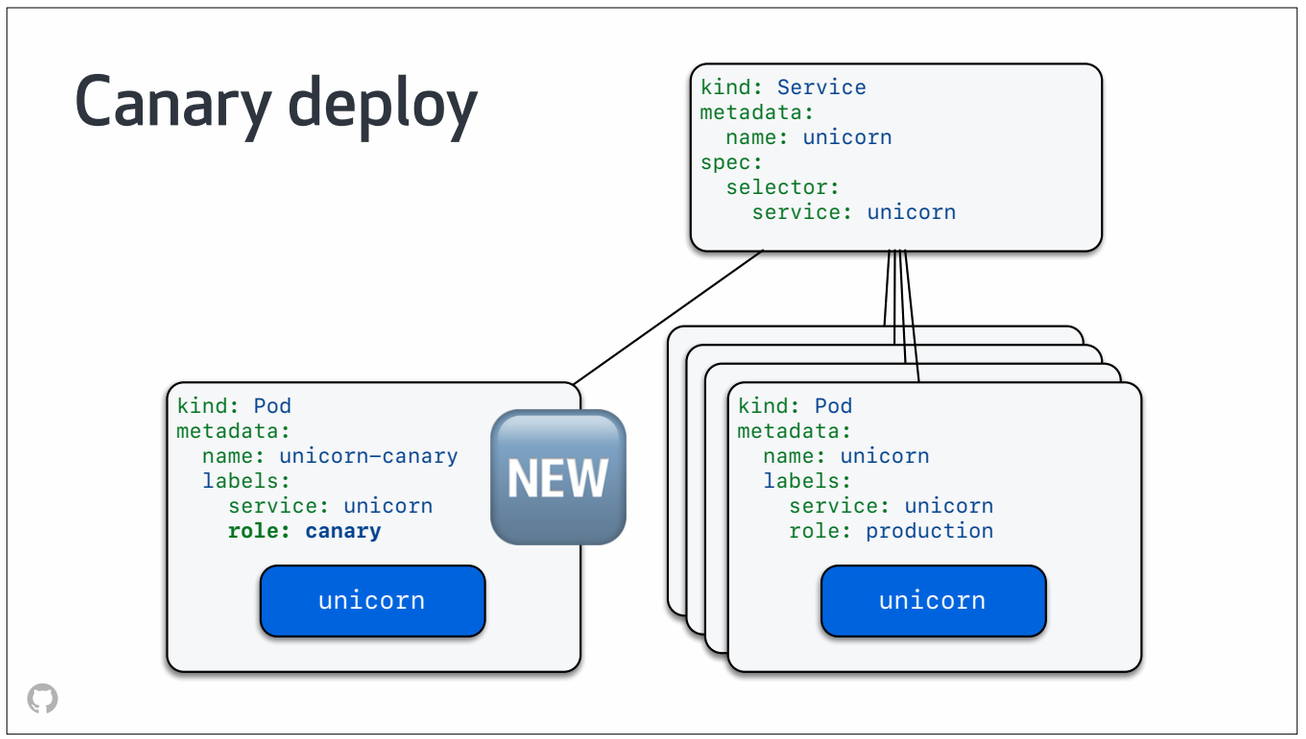
```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: unicorn
  annotations:
    ...
    heaven.githubapp.com/canary: "true"
    heaven.githubapp.com/canary-label: "role"
spec:
  replicas: 100
  template:
    metadata:
      labels:
        service: unicorn
        role: production
```

To support this canary workflow, our deployment tooling looks for deployments with a couple of special annotations: one declaring a deployment as "canary-able", and another selecting a label to be modified.
During a canary deployment, our deployment tooling clones any deployment marked as a canary and sets the value of the replicas field to 1. It changes the value of the configured label to be “canary”, and then deploys **only** the cloned canary resources.

Steady state



In a normal state of operation, the flow of traffic to our unicorn deployment looks like this: a unicorn Service sends traffic to all pods labeled with service equals unicorn.

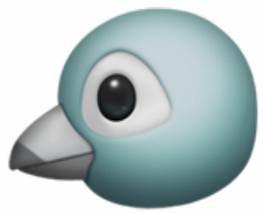


During a canary deploy, a single-pod unicorn-canary deployment is created from the new version of code, with an identical service label but a different role label. Since our Service resource doesn't include the role label in its selector, traffic goes to the canary pod at the same rate as any other pod.

Once the deploying engineer has verified that the error rate and latency of the canary deployment hasn't regressed, they then deploy their change to production, which fans your changes out to a bunch of different targets: some of this app's workloads still run on metal, so **these** environments are updated in parallel with..

```
$ kubectl apply \
          --namespace github-production \
          -Rf config/kubernetes/production
```

each cluster, again essentially running kubectl apply on the YAML in config/kubernetes/production after our deployment tooling has processed it.



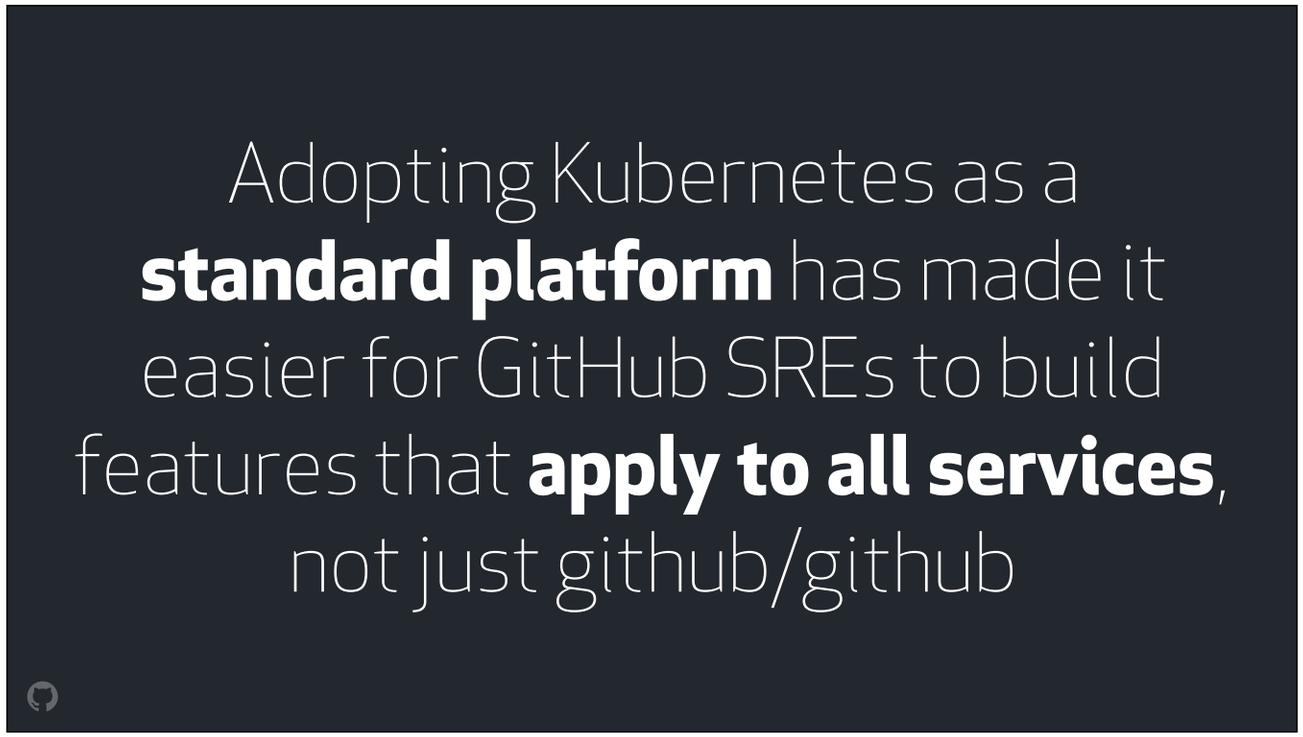
This canary deployment model isn't unique to GitHub, and to be honest our implementation is pretty naive in its current state. But it's important to point out that previously, this deployment workflow was only available when deploying the github/github service, not any of our other services.

By implementing this workflow using common Kubernetes primitives…

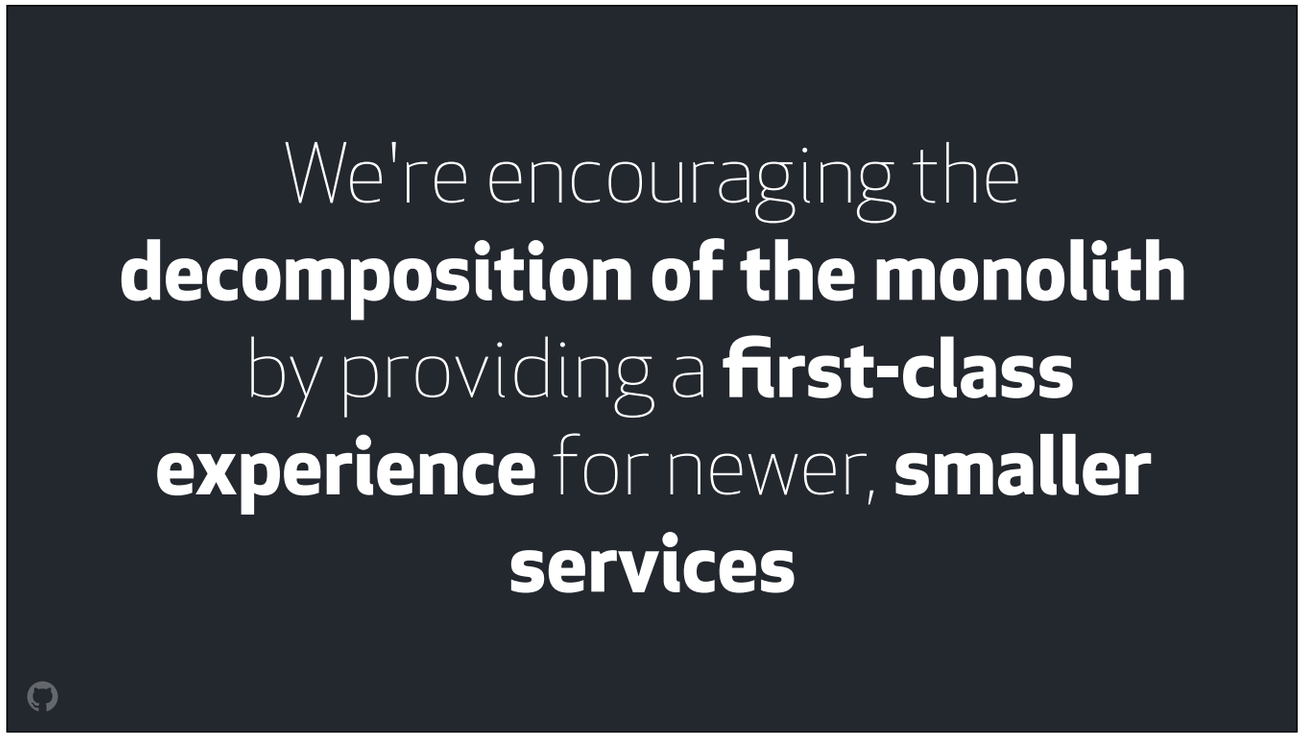
All of the other services deployed to our Kubernetes clusters can now use this canary workflow

… all of the other services deployed to our Kubernetes clusters can now use same workflow if they want.

Network effects like this are, by far, the biggest benefit we've seen as a result of adopting Kubernetes.

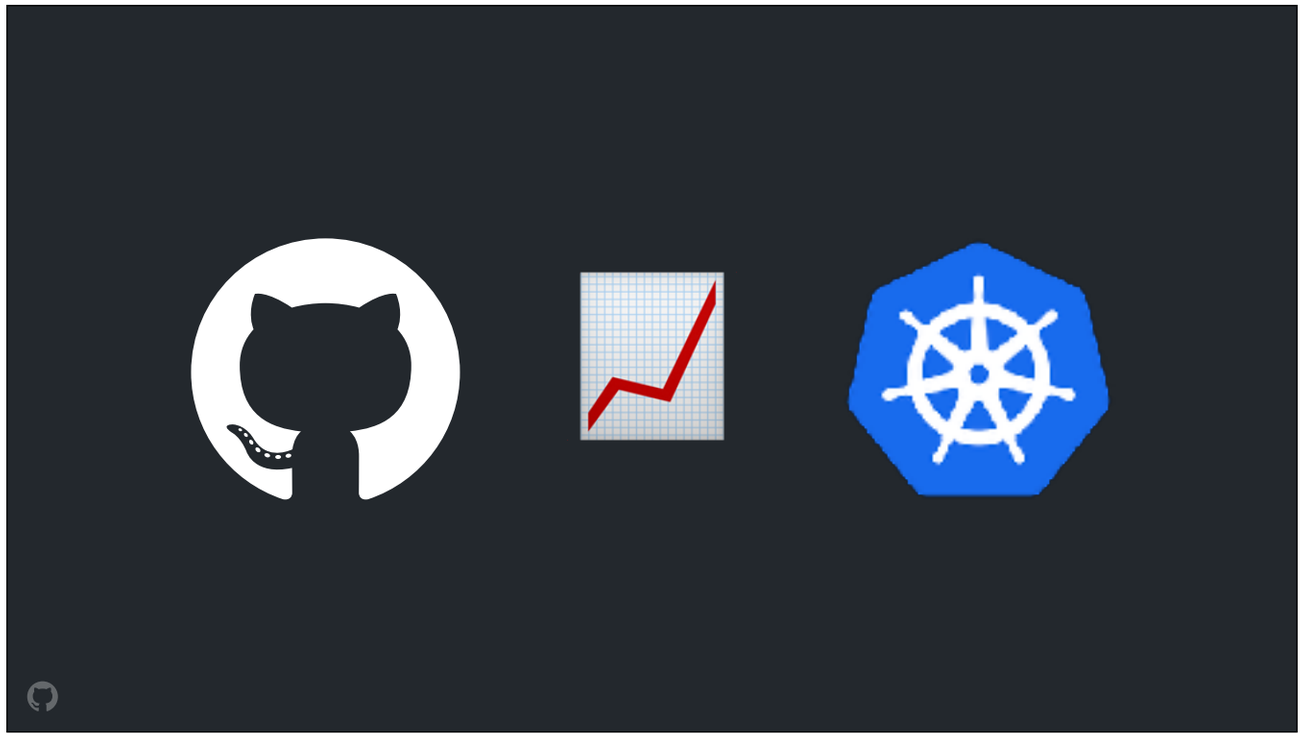


Adopting Kubernetes as a standard platform has made it easier for GitHub SREs to build features that apply to all our services, not just github/github. We're really happy about this, and are excited to continue working to improve the experience that our engineers have deploying and running **any** service. That's really important to us, as it supports another goal we're working towards:



By providing a first-class, self-service experience for newer services, we're helping to encourage the decomposition of our Ruby on Rails monolith, which is an important goal for our entire engineering org

I'll leave you today with a quick word about GitHub's plans for Kubernetes in the new year.



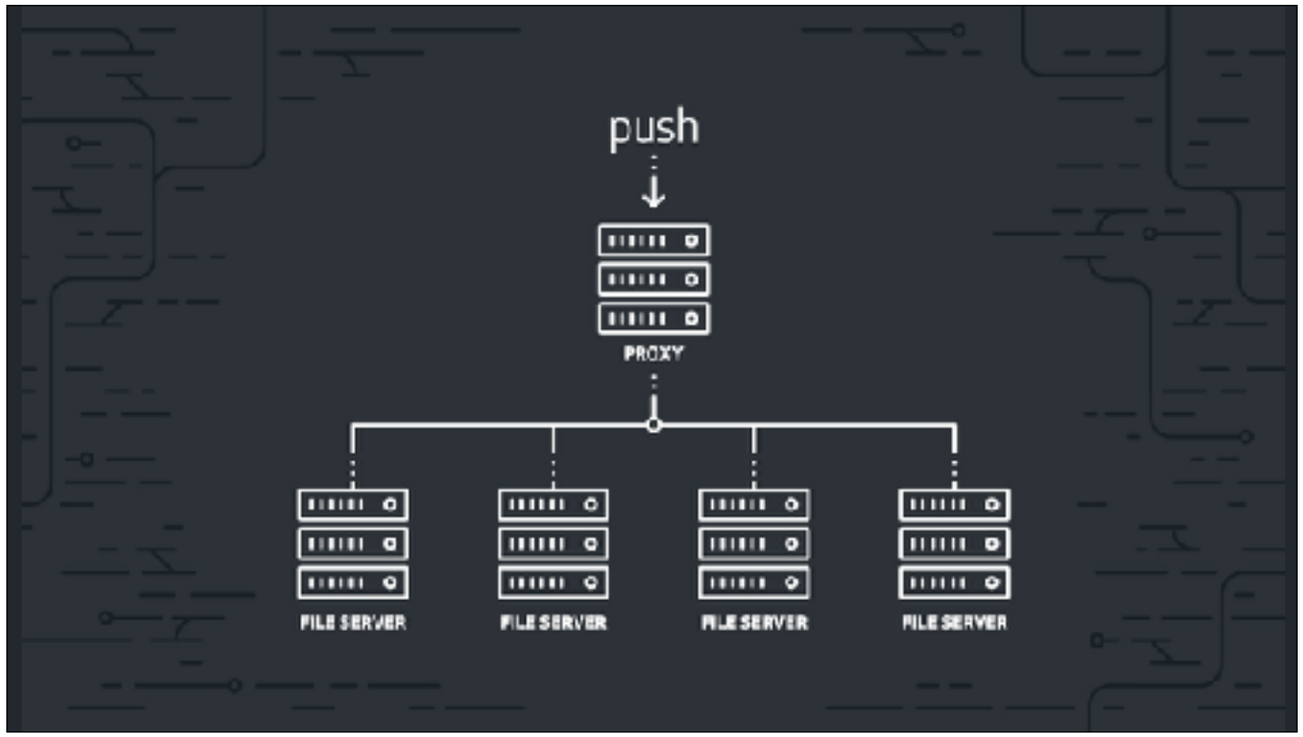
In the coming year, we're gonna be focusing a lot of our energy on **enablement** with Kubernetes - that is, we're going to be continuously evaluating the needs of engineering teams around the org, and iterating on our container orchestration platform to meet those needs

One request we've heard from a handful of engineers already is for Persistent Volume support in our on-prem clusters. State is pretty important for GitHub, right? Storing your data is at the core of our product, but right now our Kubernetes clusters can only run stateless workloads.

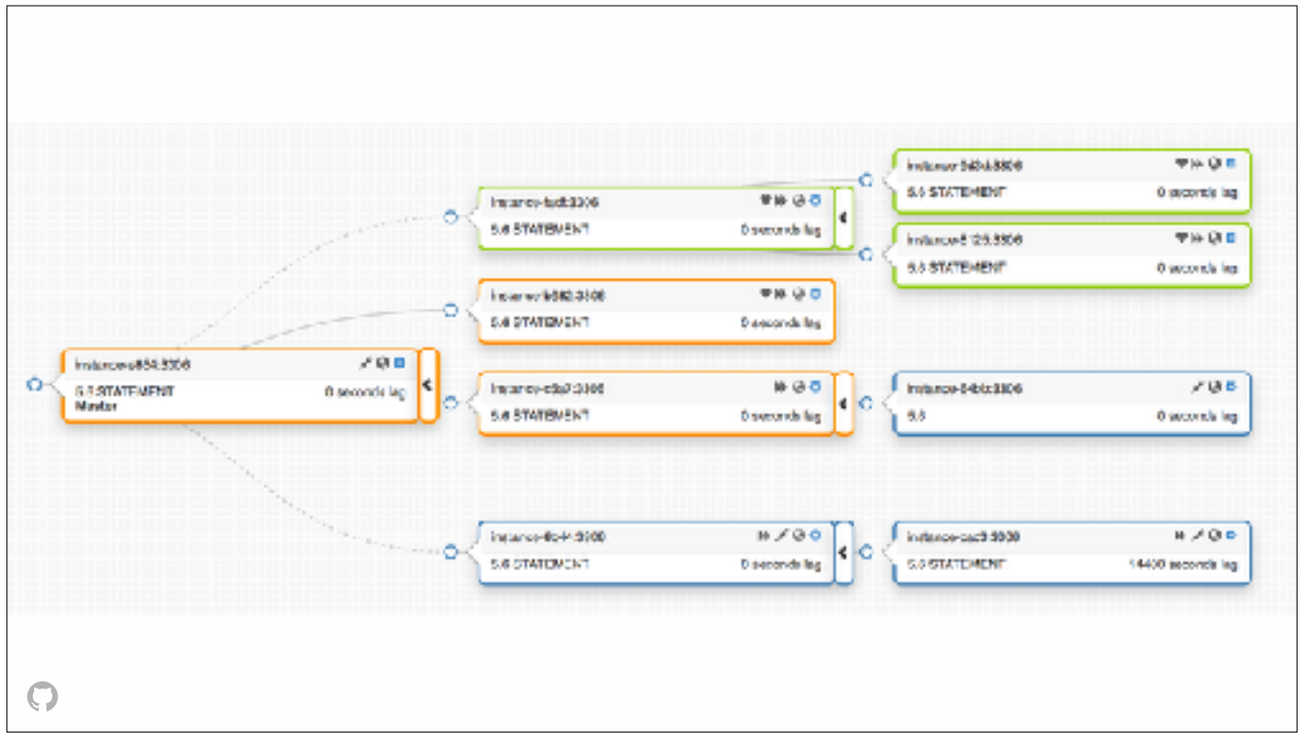
A solution to this problem is slightly complicated by our metal cloud's lack of a service like Google Cloud Persistent Disk or EBS. But even if we had such a service, network storage won't give us the performance we need for some of our systems, especially Git. Many of our systems are designed to use the SSDs available to each node to locally store a small part of a replicated, distributed dataset.

Distributed systems often use **replication** to provide fault tolerance, and can therefore **tolerate node failures**. However, data gravity is preferred for **reducing replication traffic** and cold startup latencies.

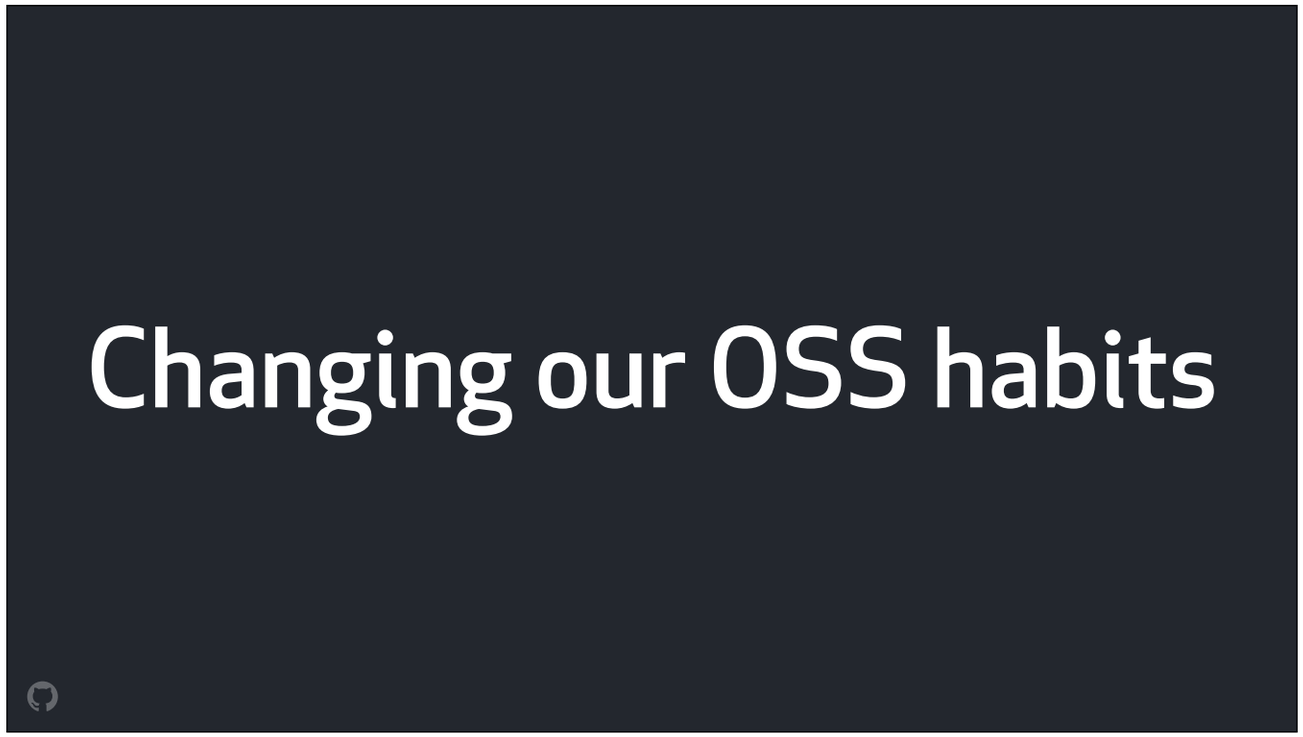
The local storage management proposal includes a use case covering persistent local storage that strongly matches some of the systems we're running today.



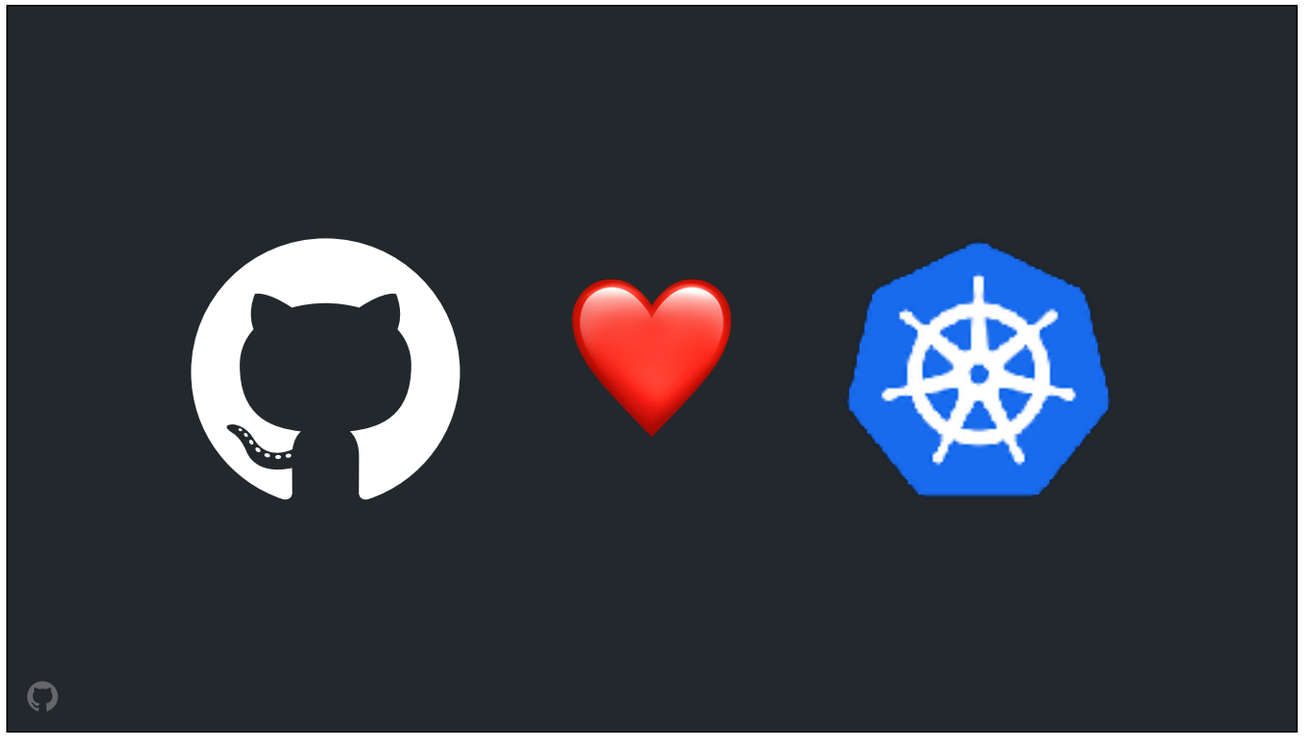
Spokes, our distributed system that stores and replicates Git repositories, looks like a great fit for this use case,



as does part of our My Ess Que Ell infrastructure, especially the fleet of replicas that we use to scale out read traffic. We're excited to begin experimenting with local storage soon.



And we're also excited to break ourselves out of some bad open source habits in the new year. None of the projects I've described in this talk are open source yet, and to be entirely honest I'm quite embarrassed by that. I'm super excited that we'll be able to commit some time to both open sourcing some of these existing tools, and to paving a path that will enable similar tools we build in the future to be developed in the open by default.



It's the least we can do to give back to the most kind, welcoming, and talented community I've ever had the privilege of considering myself a part of.

Thank you **so much** for letting me share a bit of GitHub's story with you this morning. If you're interested in chatting more about anything I just mentioned, please do reach out, or find me at the GitHub booth later today!