DEVOPS PLATFORM TRAINING WORKSHOP 201

PIPELINE YOUR DEVOPS

GITHUB FLOW

WHAT ARE FLOWS ANYWAY?

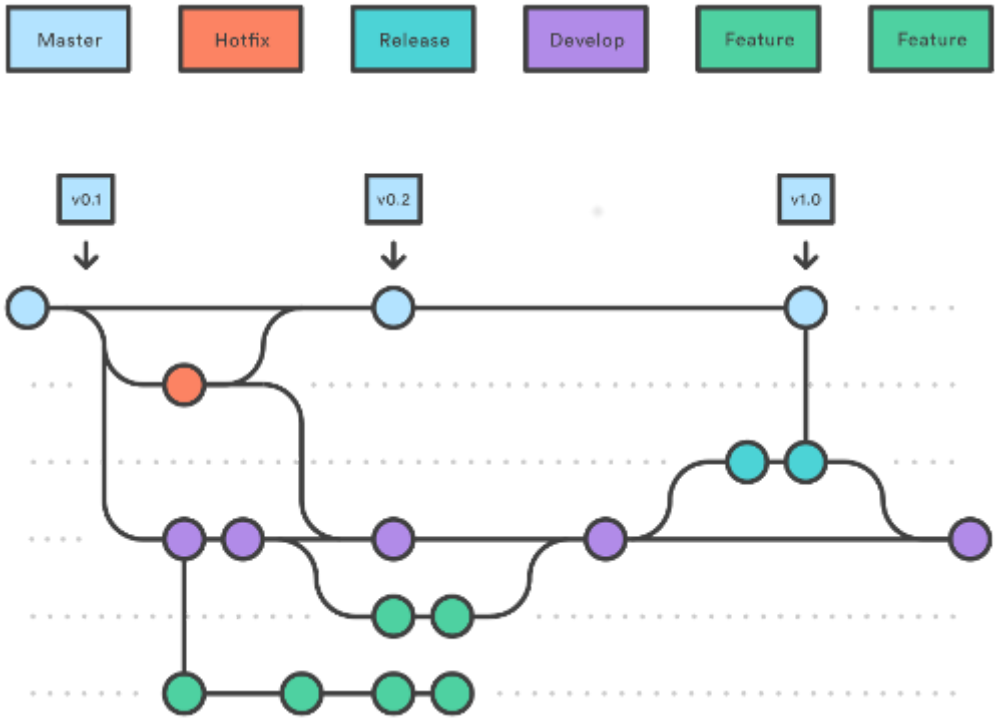
Flows are the way we organize our collaboration.

- Sharing is hard
- Rules make it easier
- Rules are driven by the goals, contraints, and teams
- With many variables, there are many options and opinions

PATHFINDER VARIABLES

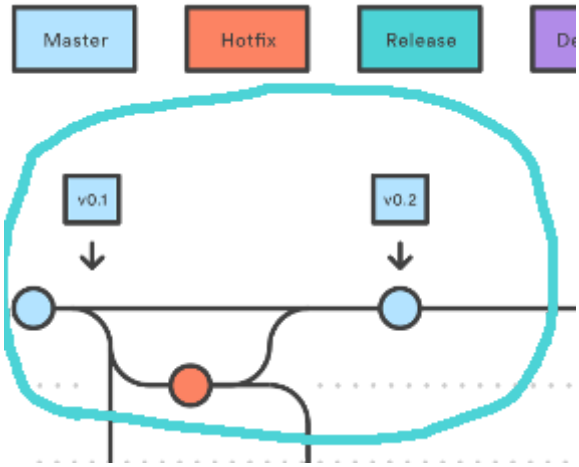
- Pathfinder promotes sprint based development for smaller/faster releases to production
- Teams can have flexible memberships
- Ramp-up needs to be fast

GIT FLOW



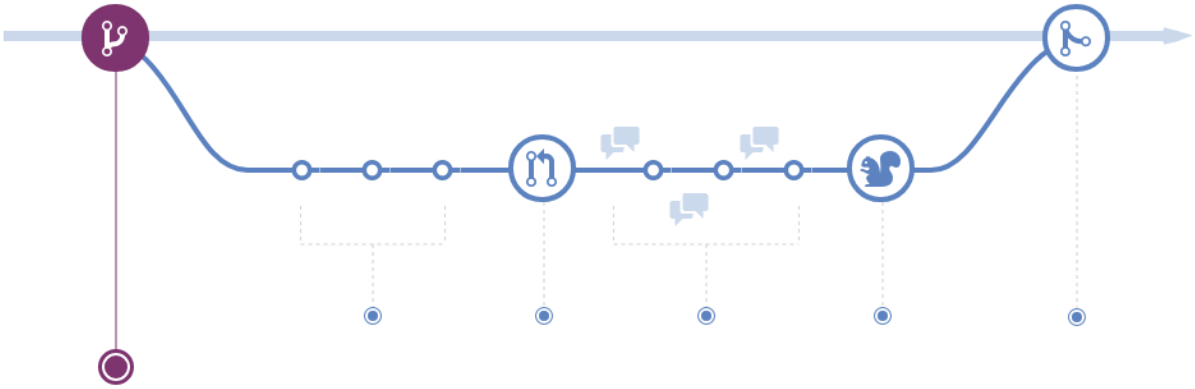
- Allows for complex parallel feature development and structured integration into releases
- Lots of moving parts and integration points

GITHUB FLOW (PATHFINDER RECOMMENDED)



- Focuses on shorter feature branches and faster iteration
- Less complexity to manage
- Fast startup for new teams

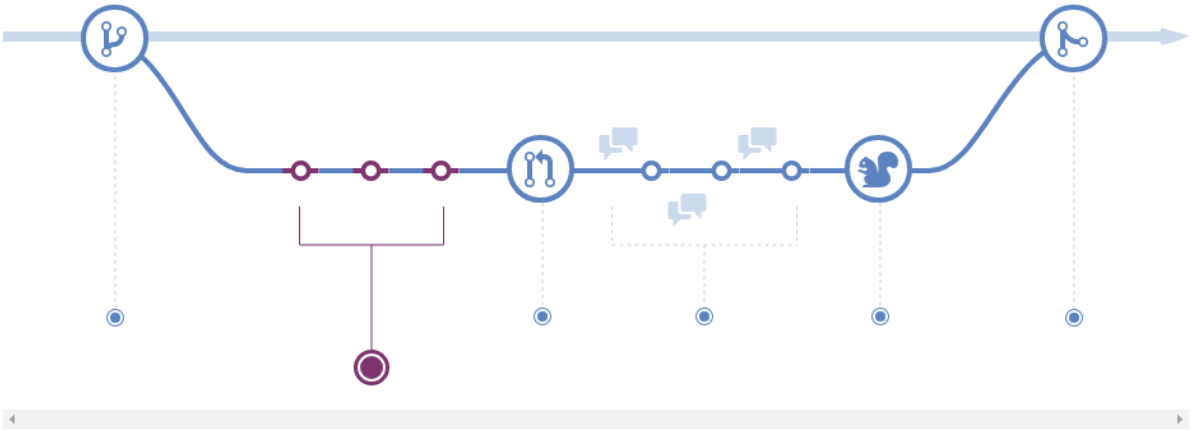
GET STARTED WITH A NEW BRANCH



Create a branch for your feature.

- one rule: `master` branch is always deployable!

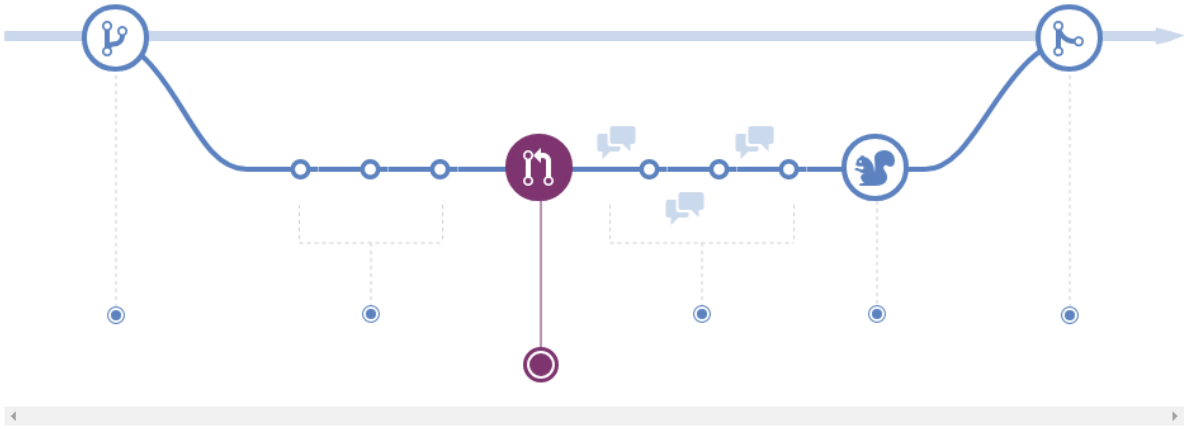
COMMIT CHANGES



Add commits for your feature.

- descriptive commit messages are super helpful

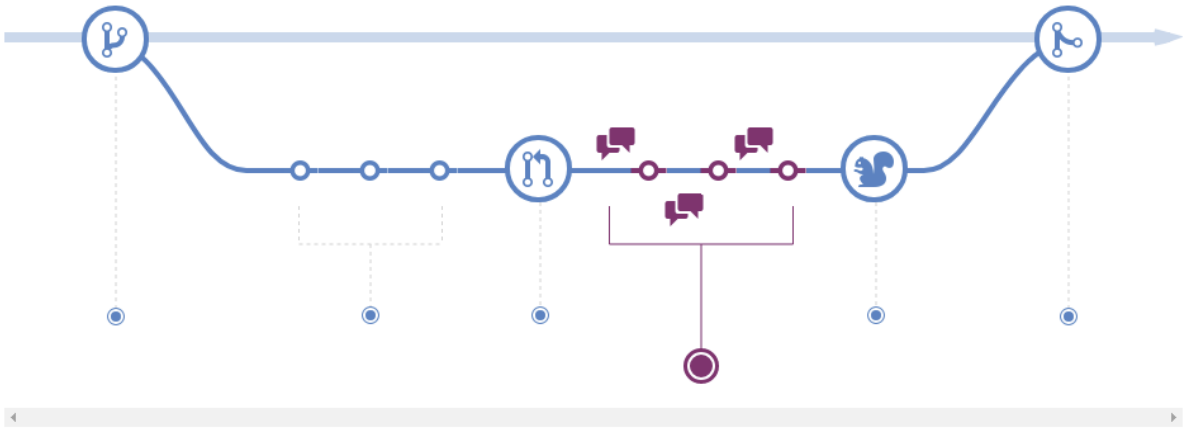
PULL REQUEST



Open a Pull Request (PR)

- Earlier PR is better to let other people comment and contribute sooner!

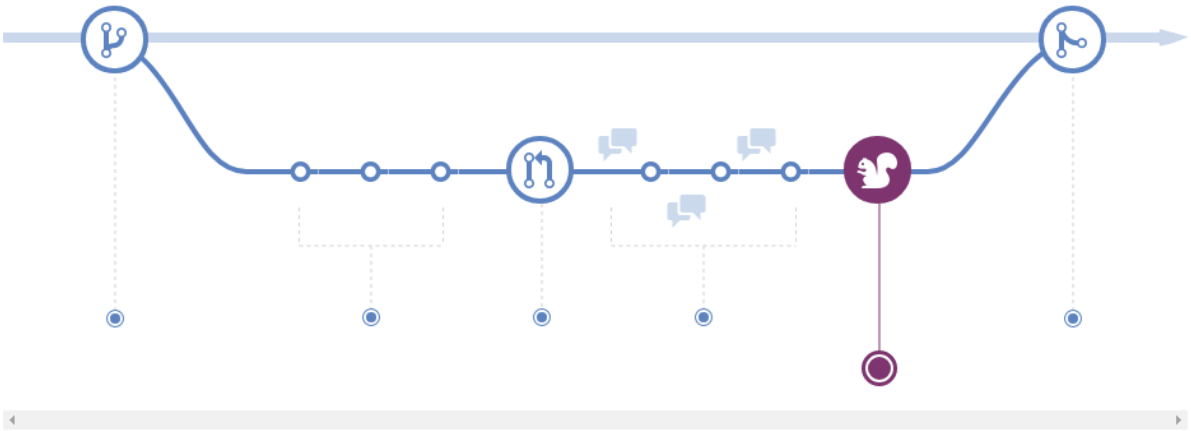
FEEDBACK



Get Feedback!, integrate changes, more commits!

- Github PR comments are in markdown, which allows for nicer formating and embedded images.

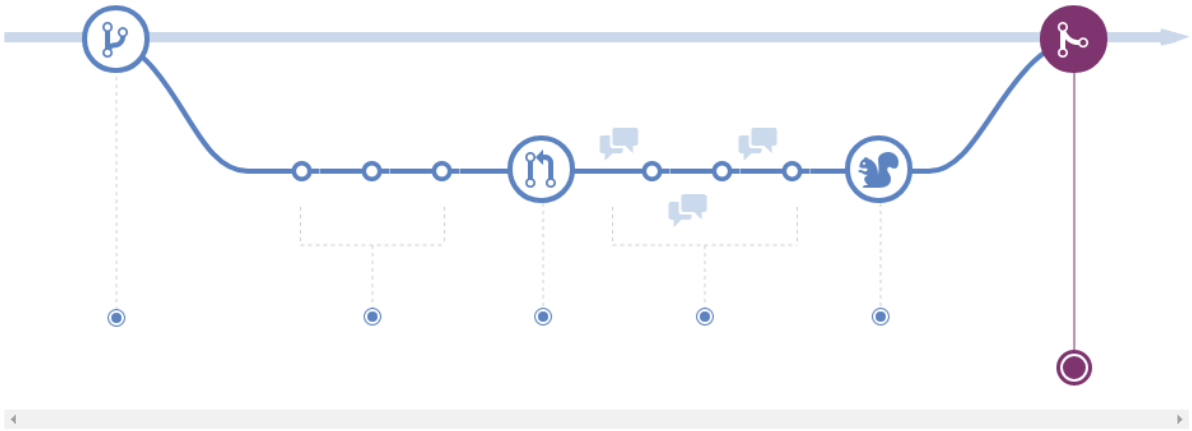
DEPLOY



Deploy from your branch for validation and testing.

- Can deploy to prod from your branch to allow for easier rollback if your branch causes prod issues.

MERGE

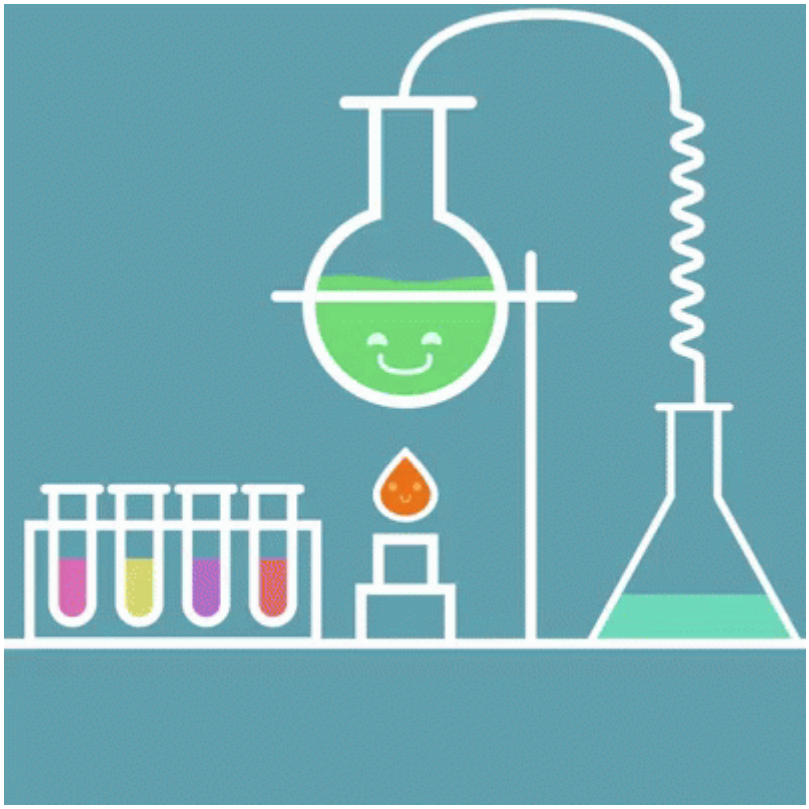


Merge your code to the master branch to finalize your change.

- A merged PR can automatically close any linked issue too!

LAB: GIT BRANCH AND PR

Break for Lab (15m)



CODIFY YOUR OPS

GITOPS

No more critical updates in the GUI forgotten long lost!

- Create *consistently repeatable* OpenShift objects
  - Declarative specifications for your objects
  - Stored with your application, as code in your repository
  - Applied by your pipeline
- Let the GUI visualize your dashboards and environment, while your changes are created and tracked in a PR

WHERE DO I START?

From right where you are today!

- Deploy your application as you are used to (oc new-b
  oc new-app, etc)
- Configure your application with extra objects (secrets
  configmaps, routes, etc)
- To export a group of objects:

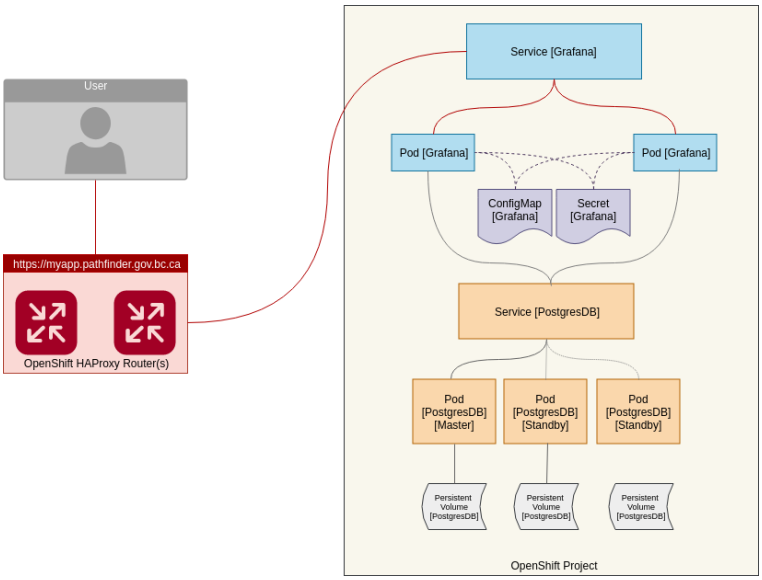
```
oc get --export all -o yaml > new-objects.yaml
oc get --export svc,dc,route -l FOO=BAR -o yaml > filtered-objects
```

- To export an individual object manifest file:

```
oc get --export {object/name} -o yaml > new-object-manifest.yaml
```

DRAW YOUR STUFF

Having a whiteboard session to draw out the connected pieces and dependencies can really help in the design.



DESIGN YOUR DEPLOYMENT

Templates vs static object manifests

- Design templates for the re-usable objects (and dependencies)
  - Deployments (DeploymentConfig, StatefulSet)
  - Abstractions (Services, Routes, ServiceAccounts, RoleBindings, etc)
- Design static object manifests for environment specific objects
  - Secrets, ConfigMaps
  - PVCs

LABELS AND PARAMETERS

- Create **meaningful labels** and add them at the template metadata layer to apply to all objects that the template will manage
- Labels can:
  - be used to group objects in your template
  - make cleanup/redeployment easier
  - identify release versions
- Parameters allow the templates to be reusable
  - application name
  - git configuration (git repo, tag, branch?)
  - PVC Names, secret names, configmaps, etc

DEPLOY YOUR STUFF!

Your first deploy doesn't have to wait for any parameters; add them in as the objects are refined through testing

- using `oc apply` you can keep an object definition updated from a yaml or json manifest
- use `oc process` to create the object definitions from template for `oc apply`

```
oc apply -f my-configmap.yaml
oc process -f new-template.yaml -p FOO=VAR | oc apply -f -
```

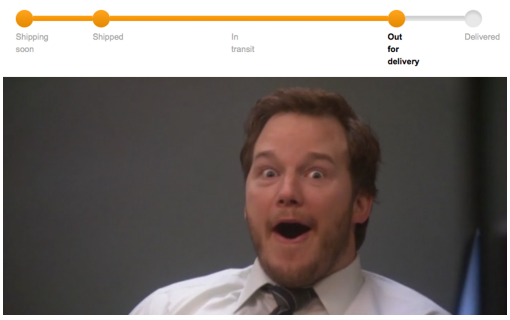
- Parameter files can store multiple parameters for a specific environment

```
oc process -f new-template --param-file=myapp-dev.yaml | oc apply
```

REFINE AND REPEAT

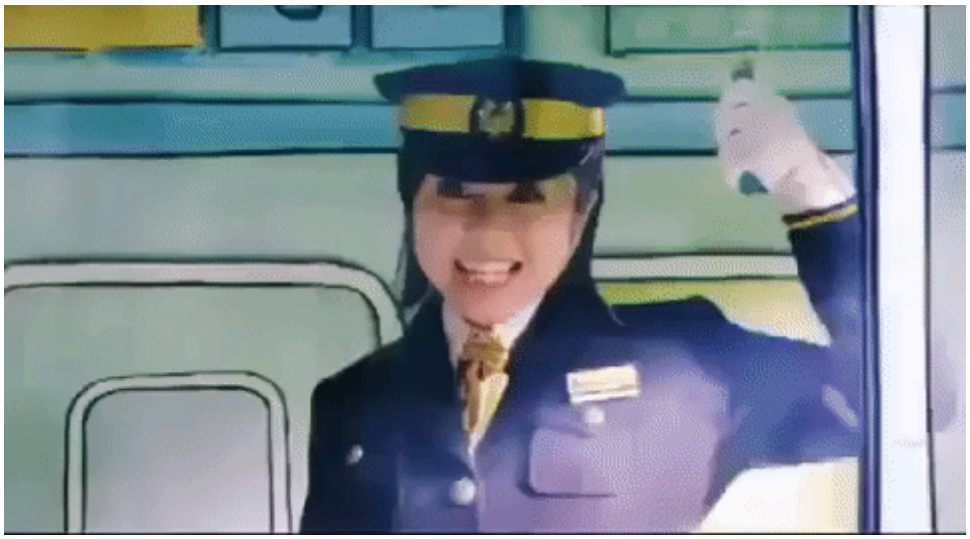
- Modify the template file:
  - Add more parameters (with sane defaults)
  - Process and apply your template in your dev namespace
  - Each successive apply will modify or create your objects. (but not delete them)

DEPLOY TO NEXT ENVIRONMENT



- Use the templates and object files to deploy to a second environment
- Verify objects and application are running as expected

SMELLS LIKE A PIPELINE!



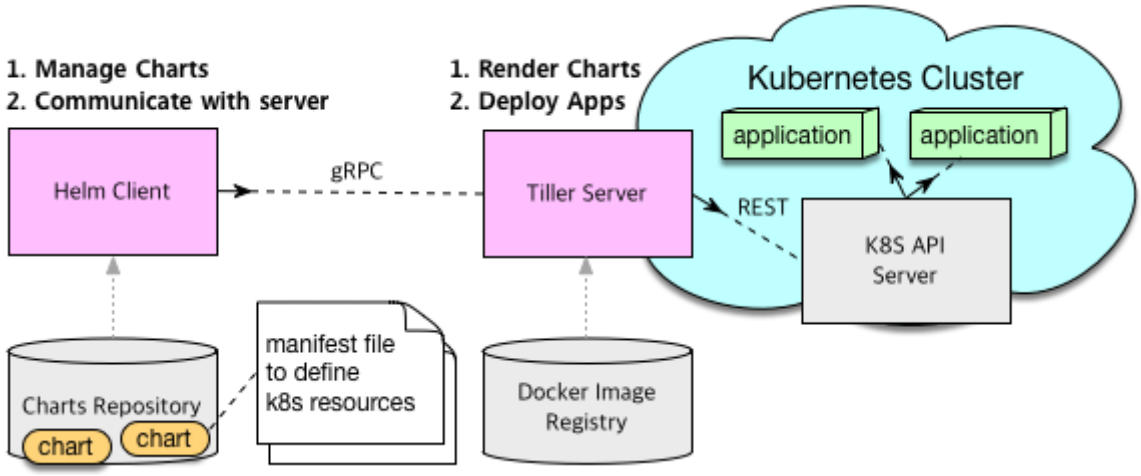
HELM

- Isn't an OpenShift primative
- Somewhat overlaps with OpenShift templates
- Is super duper popular, which is why we are talking about it!

WHAT IS HELM?

- Helm is a (well... the official) Package Manager for Kubernetes
  - package multiple K8s resources into a single logical deployment unit: Chart
  - but it's not just a Package Manager
- Helm is a Deployment Manager for Kubernetes to;
  - perform a repeatable deployment
  - manage dependencies: reuse and share
  - manage multiple configurations
  - update, rollback and test application deployments (Releases)

HELM ARCHITECTURE



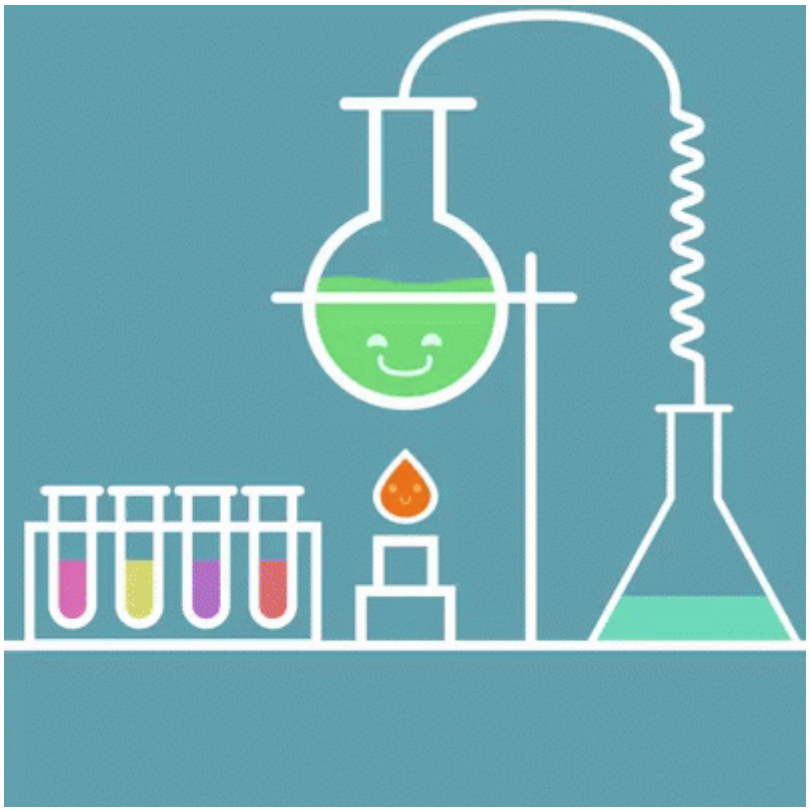
OPENSHIFT TEMPLATES VS HELM

- OpenShift templates are simple "deployment" tools
  - Requires OpenShift; does NOT work across other kubernetes platforms
- Helm is more of a package manager
  - Requires Tiller for package management
  - We do NOT run Tiller in the OpenShift environment due to the elevated privileges it requires
  - Many kubernetes-native apps are being distributed with Helm charts
- Helm CAN be used to deploy applications (without package management)
  - Helm is used in client-only mode
  - Helm is used in template mode to render the artifacts before applying them
  - This has similar functionality to OpenShift templates

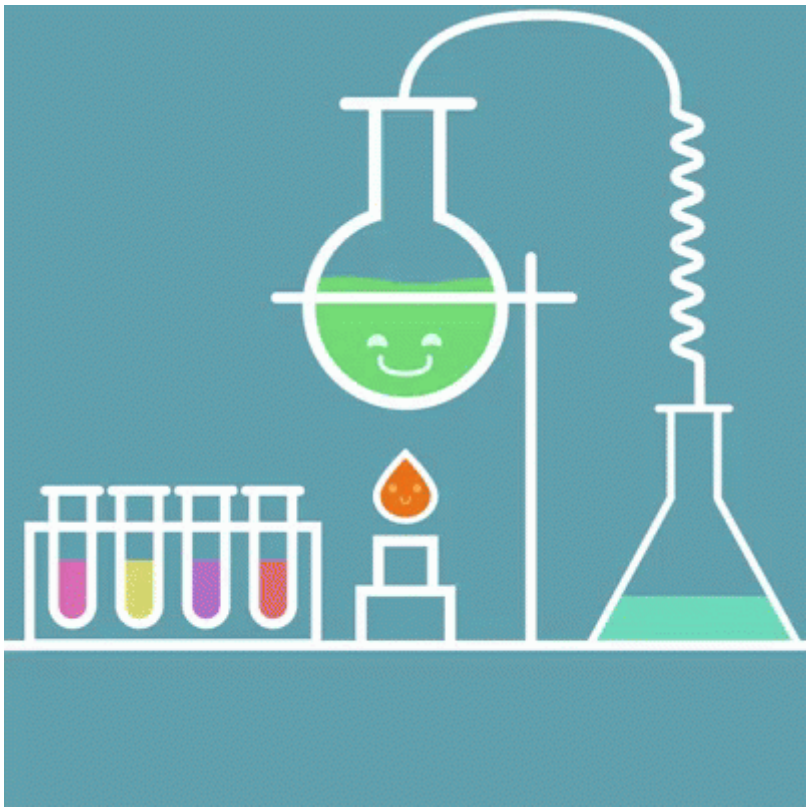
USING HELM WITHOUT TILLER

```
helm repo add [chartname] [chart-path]
helm fetch [chartname]/[component]
helm template ./[component].tgz --set \
  key1=value1 \
  key2=value2 \
  | oc apply -f -
```

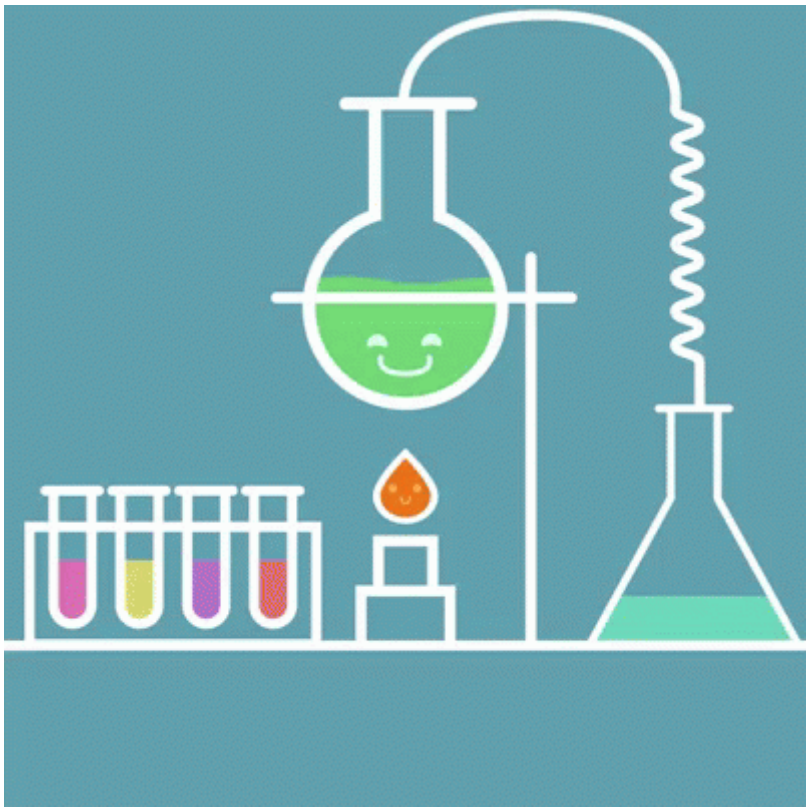
LAB: BUILD OBJECT MANIFESTS



LAB: BUILD TEMPLATE



LAB: RE-USE / RE-DEPLOY



BREAK (60 MINUTES / FOOD)

CI/CD AND PIPELINE BASICS

CI/CD

- Continuous Integration
  - the developer's working copies are synchronized with a shared mainline several times a day
- Continuous Delivery
  - the logical evolution of continuous integration: Always be able to put a product into production
- Continuous Deployment
  - automatically deploy the product into production whenever it passes QA

OPENSHIFT INTERNAL CI/CD CAPABILITIES

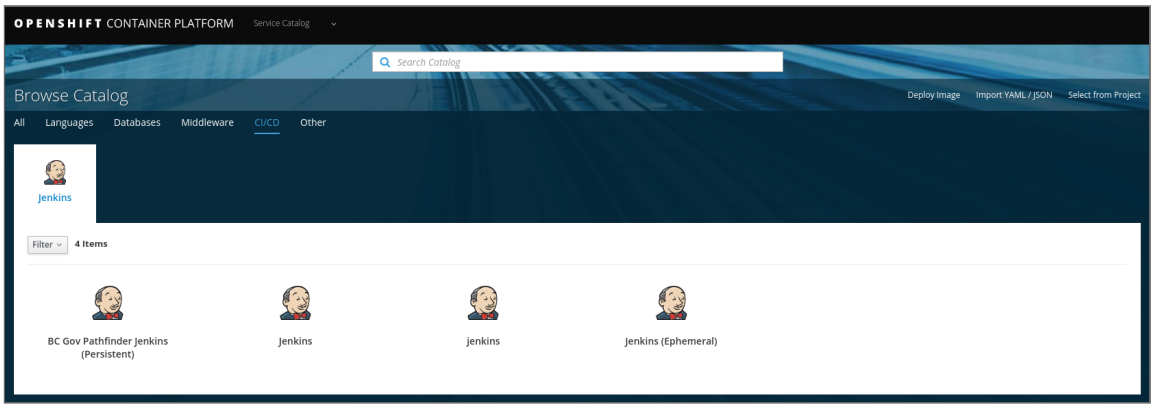
- Webhook triggers for builds
  - Build my application image when my code changes
- ImageChangeTriggers for builds
  - Build my application image when my base image changes
- Post-build test hooks
  - Test my application image before I push it to a registry
- ImageChangeTriggers for deployments
  - Redeploy my application when my image changes (e.g. after a build)

OPENSHIFT EXTERNAL CI/CD CAPABILITIES

- API calls from external CI/CD system
  - Trigger a build
  - Trigger a deployment
  - Tag an image between registries
- Via oc client or direct API invocations

JENKINS INTRODUCTION

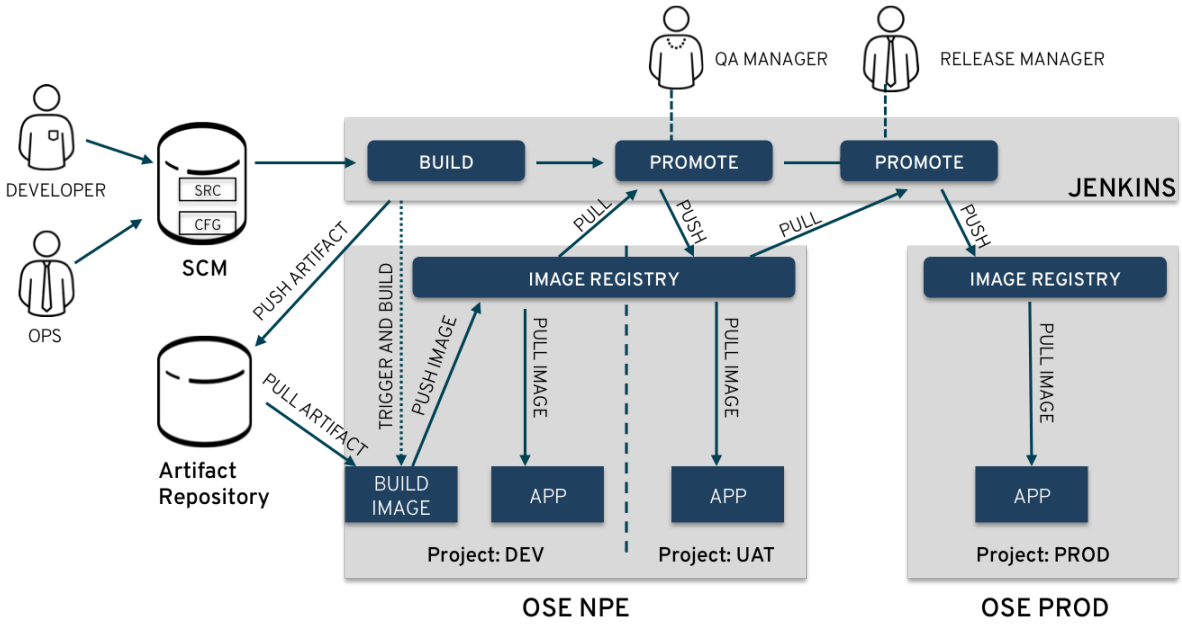
- The defacto CI/CD tool provided by Red Hat in OpenShift
- Provides flexible capabilties that are integrated into OpenShift
- Multiple catalog items exist



PERSISTENT VS. EPHEMERAL INSTANCES

- Persistent Jenkins masters are the most common deployment
- Often slower than ephemeral instances, however:
  - Custom configuration is maintained across pod restarts
  - Build history is maintained
- Ephemeral instances lose all configuration / history upon restart of a pod

JENKINS PIPELINES - PROMOTION ACROSS ENVIRONMENTS



JENKINS PIPELINES - PROMOTION ACROSS ENVIRONMENTS

- Jenkins should only run in the `tools` project
  - Service accounts are provided access to `dev`, `test`, and `prod` projects
  - Each project service accounts group is provided `system:image-puller` role with permission to pull from the tools project
  - Image tagging should be used diligently to ensure the right images are depoyed to the appropriate namespaces

JENKINS OPENSHIFT INTEGRATION

- OpenShift Jenkins Integration Documentation
  - **https://docs.openshift.com/container-platform/3.11/dev_guide/openshift_pipeli jenkins-Pipeline-plugin**
- OpenShift Client Plugin
  - **https://github.com/openshift/jenkins-clie**
  - OpenShift API interaction with a pipeline DSL
  - oc client integration
- OpenShift Pipeline Plugin / Build Strategy

OPENSHIFT PIPELINES

- OpenShift and Jenkins together:
  - Integrated Jenkins master and slave images
  - Synchronization logic between Jenkins and OpenShift
  - Web console for viewing Pipelines
  - Autoprovisioning of Jenkins as needed via Jenkinsfile detection
  - Pipeline BuildConfig strategy type
    - Treat a Pipeline like any other OpenShift build
      - `oc start-build`
      - `oc get builds`
    - Appears in web console
    - Build status reflects pipeline success
    - Supports all typical build triggers
    - Has no builder image, no output
    - Cannot use oc logs (there is no pod)

JENKINS PLUGIN DSL & THE JENKINSFILE

- Groovy syntax used in the Jenkinsfile
- Jenkinsfile can be included in the BuildConfig object, alongside the application code
- Types of OpenShift actions supported:
  - Builds - trigger, watch, verify
  - Deployments - trigger, scale, verify
  - Services - verify availability
  - Images - tag
  - Resources - create from yaml/json, delete by nam label, or yaml/json
- **https://jenkins.io/doc/pipeline/steps/opensh pipeline/**

PIPELINE SAMPLE CONFIGURATION

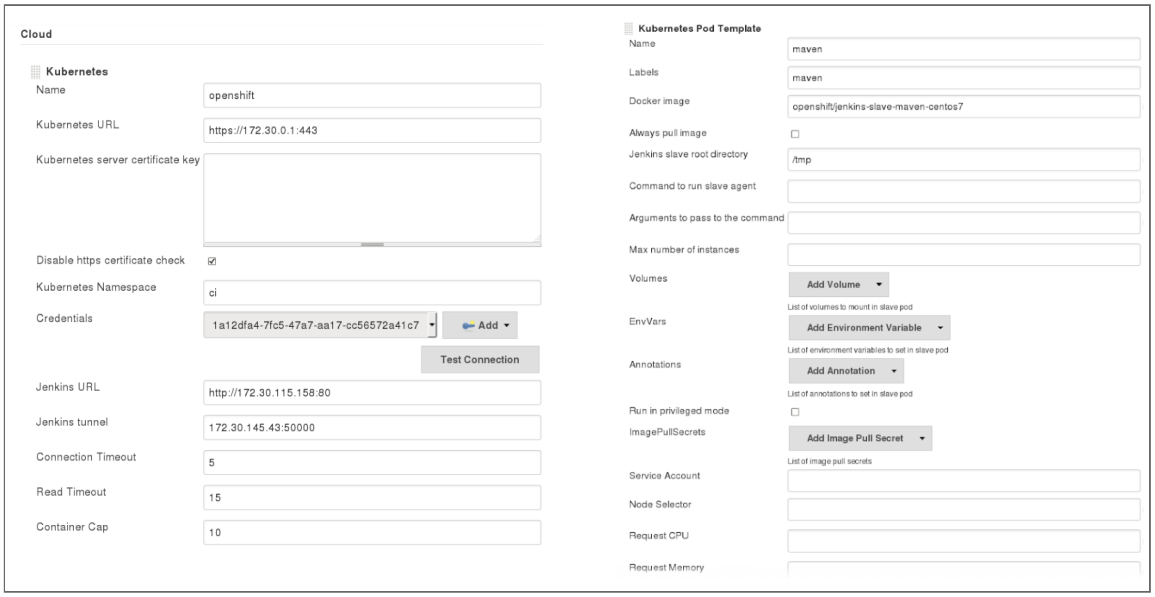
```
kind: BuildConfig
apiVersion: v1
metadata:
  name: sample-pipeline
  labels:
    name: sample-pipeline
  annotations:
    pipeline.alpha.openshift.io/uses: '[{"name":
"frontend", "namespace": "", "kind":
      "DeploymentConfig"}]'
spec:
  triggers:
  - type: GitHub
    github:
      secret: secret101
  - type: Generic
    generic:
      secret: secret101
strategy:
  type: JenkinsPipeline
  jenkinsPipelineStrategy:
    jenkinsfile: |-
      node('maven') {
        stage 'build'
        openshiftBuild(buildConfig:
'ruby-sample-build', showBuildLogs: 'true')
        stage 'deploy'
        openshiftDeploy(deploymentConfig: 'frontend')
      }
```

JENKINS IMAGES

- Jenkins Master Image
  - OpenShift Pipeline plugin, Jenkins Pipeline plugin
  - Kubernetes plugin for slave pod launching
  - oc client, git client tools for shell operations
  - Sync plugin for synchronizing pipeline state
  - Self-configures slave templates
- Jenkins Slave Images
  - Maven, NodeJS
  - Include oc, git, framework specific tools
  - Auto-instantiated in pods via Kubernetes slave pod plugin
  - Connects back to jenkins master via service definition environment

JENKINS AGENT IMAGES

- Custom agents can easily be used to extend the build capabilities
- Through the GUI, this can be done in the Jenkins configuration:



JENKINS AGENT IMAGES

- Or in your jenkinsfile:

```
// Pod Templates for Build
podTemplate(
label: 'tf-pod',
cloud: 'kubernetes',
containers: [
    containerTemplate(name: 'terraform',
                    image: 'docker.io/stewartshea/terraform-googlesdk-slave:alpine-0.2',
                    resourceLimitMemory: '512Mi',
                    workingDir:'/home/jenkins',
                    command: '/bin/sh',
                    args: '',
                    ttyEnabled: true,
                    envVars: [
                        envVar(key: 'TEST', value: 'HELLO')
                        ]
                    )
            ],
        volumes: [
                        secretVolume(secretName: 'ci-automation-creds', mountPath: '/home/jenkins/creds', defaultmode: 0600),
                ],
        // I don't think this works
        imagePullPolicy: ['Always']
)



// Pipeline Execution
{
    node('tf-pod') {
        container('terraform'){
```

JENKINS AUTO PROVISIONING

- The auto-provisioned Jenkins instance creates:
  - Route
  - Service
  - Deployment Config
  - Oauth credentials
  - Service Account
  - Edit permission binding
  - Persistent storage
- Users that want to customize the Jenkins deployment (ie. different route name, ephemeral storage, etc) can manually deploy the instance from the Catalog
- Autoprovisioning will not occur if an existing Jenkins instance exists

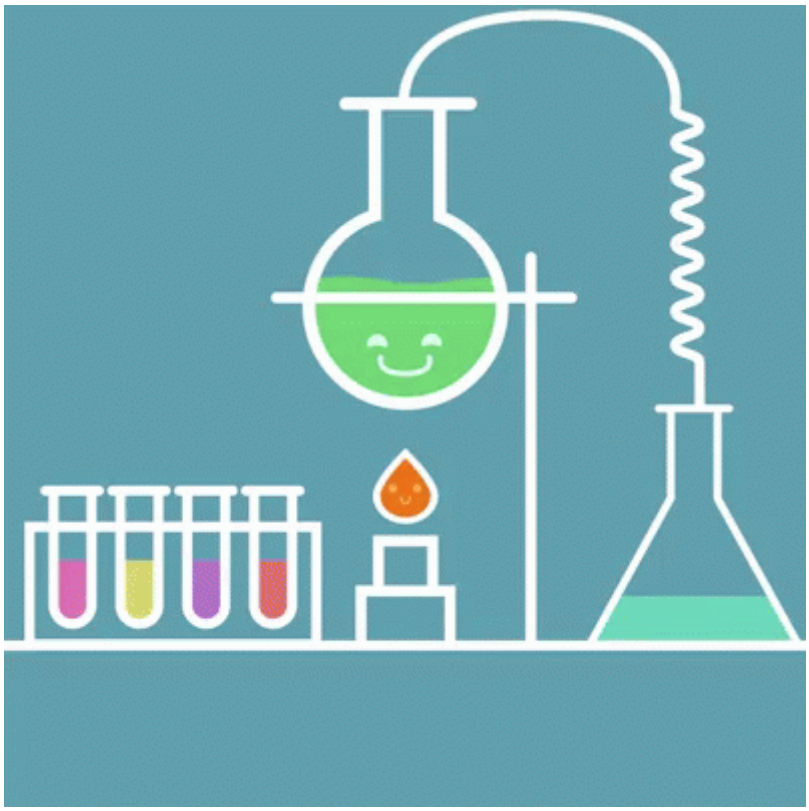
JENKINS PIPELINES - EXTENDING JENKINS WITH PLUGINS

- Add additional plugins that may not be included in the base image
- Avoid manual installation by setting the INSTALL_PLUGINS environment variable
- Manual install / configuration requires persistent storage

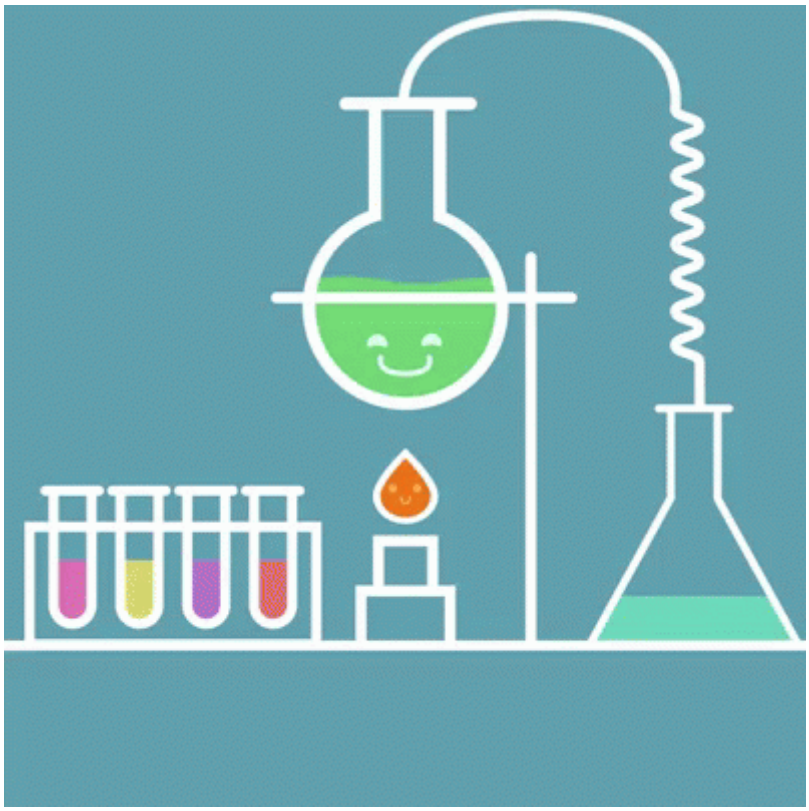
JENKINS OPINIONATED BEST PRACTICES

- Instance configuration
  - Leverage ephemeral if possible
  - Avoid complex manual configuration if possible
  - Attempt to treat this as cattle, not a pet
  - If persistent, keep your plugins and components upgraded
- Permission structures
  - Only the Jenkins service account has access to deploy to prod (regular users do not)
- Jenkinsfile
  - stored with code (not in BuildConfig)
  - contains as much as possible to reduce manual configuration

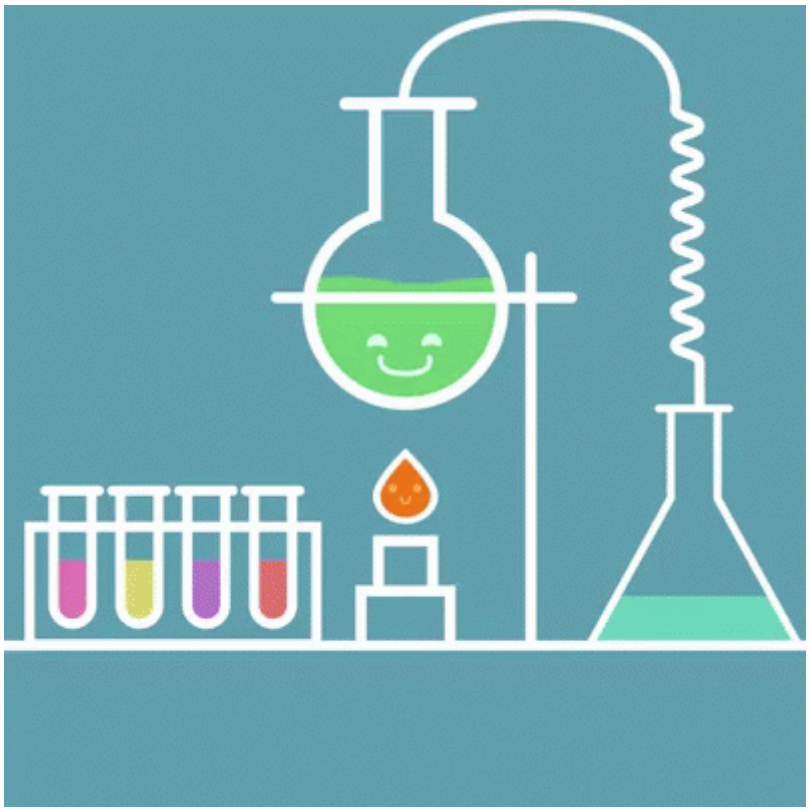
LAB: JENKINS BASIC PIPELINE



LAB: JENKINS MORE ADVANCED PIPELINE



LAB: OPENSHIFT OBJECTS IN JENKINS PIPELINES



DEPLOYMENT STRATEGIES


60% OF THE TIME
DEPLOYMENTS WORK
EVERYTIME!

DEPLOYMENT STYLES RECAP

- Standard OpenShift Deployment Styles
  - Rolling vs. Recreate
  - Rolling
    - Supports life-cycle hooks for injecting code into deployment process
    - Waits for pods to pass readiness check before scaling down old components
    - Does not allow pods that do not pass readiness check within timeout
    - Used by default if no strategy specified on deployment configuration
  - Recreate
    - Has basic rollout behavior
    - Supports life-cycle hooks for injecting code into deployment process
    - Steps in recreate strategy deployment:
    - Execute pre life-cycle hook
    - Scale down previous deployment to zero
    - Scale up new deployment
    - Execute post life-cycle hook
  - Rollbacks
  - Triggers

CANARY RELEASES

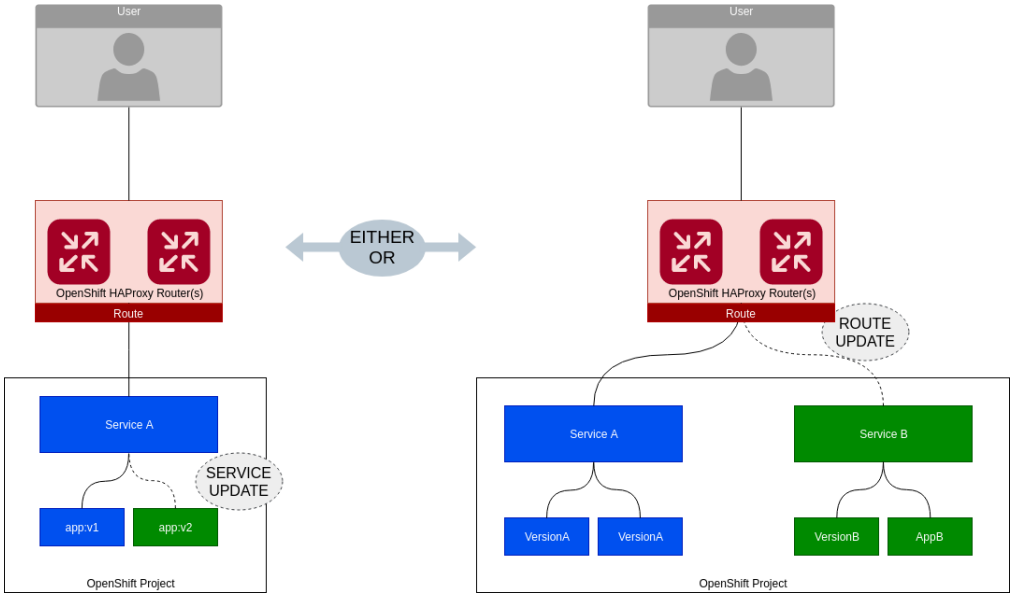
- Canary Release
  - Simple implementation is with a Rolling deployment strategy
  - Traffic is never routed to the new version if readiness probes do not succeed
  - The more advanced the readiness indicator, the better the outocomes
  - Other interpretations and implementations of this could be:
    - slowly releasing a new version of the application gradually to the same set of users and monitoring errors
    - deploying to a subset of users with separate environments

BLUE - GREEN

- Technique for releasing application in predictable manner
  - Goal: Reducing downtime associated with release
- Quick way to both:
  - Prime application before releasing
  - Quickly roll application back if you find issues
- Two identical environments (infrastructure): "blue" and "green"
- 'Green' environment hosts current production applications
- Run two different versions of application side by side
- After validating new software version, can switch all traffic to new version
- Lets you update and roll back code and databases in one step

BLUE/GREEN DEPLOYMENT

- Completely re-route traffic to new version of the application
- Easily roll back to existing *running* version if errors arise
- Otherwise, remove old version of application when ready

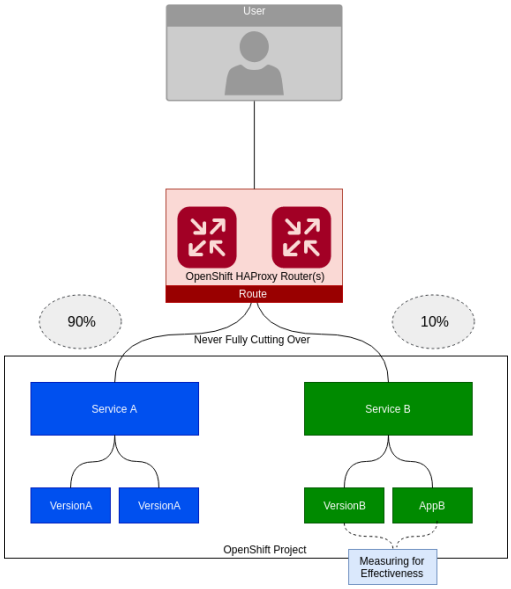


A/B TESTING

- Method of testing application features
  - Test for usability, popularity, noticeability, etc.
  - Also how these factors influence bottom line
- Usually associated with application UI
- Back-end services need to be available to implement
- Can implement with either application-level or static switches

A/B VIA PERCENTAGE OR SEPARATE ROUTE

- Gradually direct *some* users to the new version
- Monitor feature performance
- Remove new feature and roll changes into main build

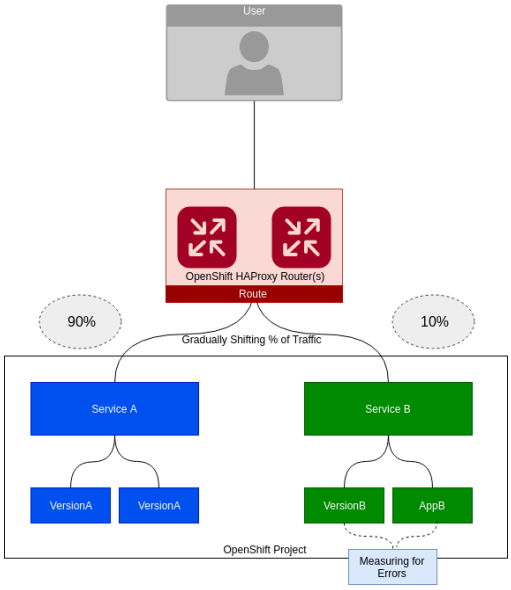


BLUE/GREEN VS A/B TESTING

- A/B testing measures application functionality
- Blue-green deployment releases software safely and lets you roll back predictably
- Can combine the two:
  - Use blue-green deployments to deploy new features
  - Use A/B testing to test those features

CANARY VIA PERCENTAGE

- Gradually direct *some* users to the new version
- Monitor application health (logs, apm, etc)
- Increase percentage of users directed to the new version until 100%



BREAK (DAY 1 COMPLETE!)

STATEFUL SETS

For when you can't dump all your state

WHAT ARE THEY GOOD FOR?

Stateful sets are the answer for applications that require sticky identities (pets) for their scaling pods.

- Known/predictable pod names (app-[0..n-1])
- Unique PVC for each pod (either pre-created, or dynamically provisioned)
- Predictable startup and shutdown order (more control when re-deploying or scaling)

POD SCALING AND UPDATES

When a stateful set is being deployed, the pods will always have a stable hostname based on it's sequence.

- **OrderedReady** pod management (default):
  - Created sequentially in order from {0..N-1}.
  - Deleted sequentially in reverse order from {N-1..0}
  - Before a scaling operation is applied (either create or terminate), all of it's predecessors must be Running and Ready.
- **Parallel** pod management:
  - will launch or terminate all pods in parallel without the above restrictions.

UPDATE STRATEGIES

Updates include changes to containers, labels, resource request/limits, and annotations for the pods.

- **RollingUpdate** *(default)*: Triggered deployment that deletes and re-creates each pod in the same order as pod termination.
- **OnDelete** : Requires manually deleting a pod before it will be updated.

PVC STORAGE

Each pod in a stateful set has a unique PVC.

- Must either be pre-created, or leverage a StorageClass that has auto-provisioning
- This PVC will **NOT** be deleted when scaling down or otherwise removing pods.

KEY DRAWBACKS

- No auto-scaling
- Unique storage per pod (replication/synchronization of data is the responsibility of the application)
- Storage must be provisioned at or before creation time, and will not be removed when you delete the stateful set.

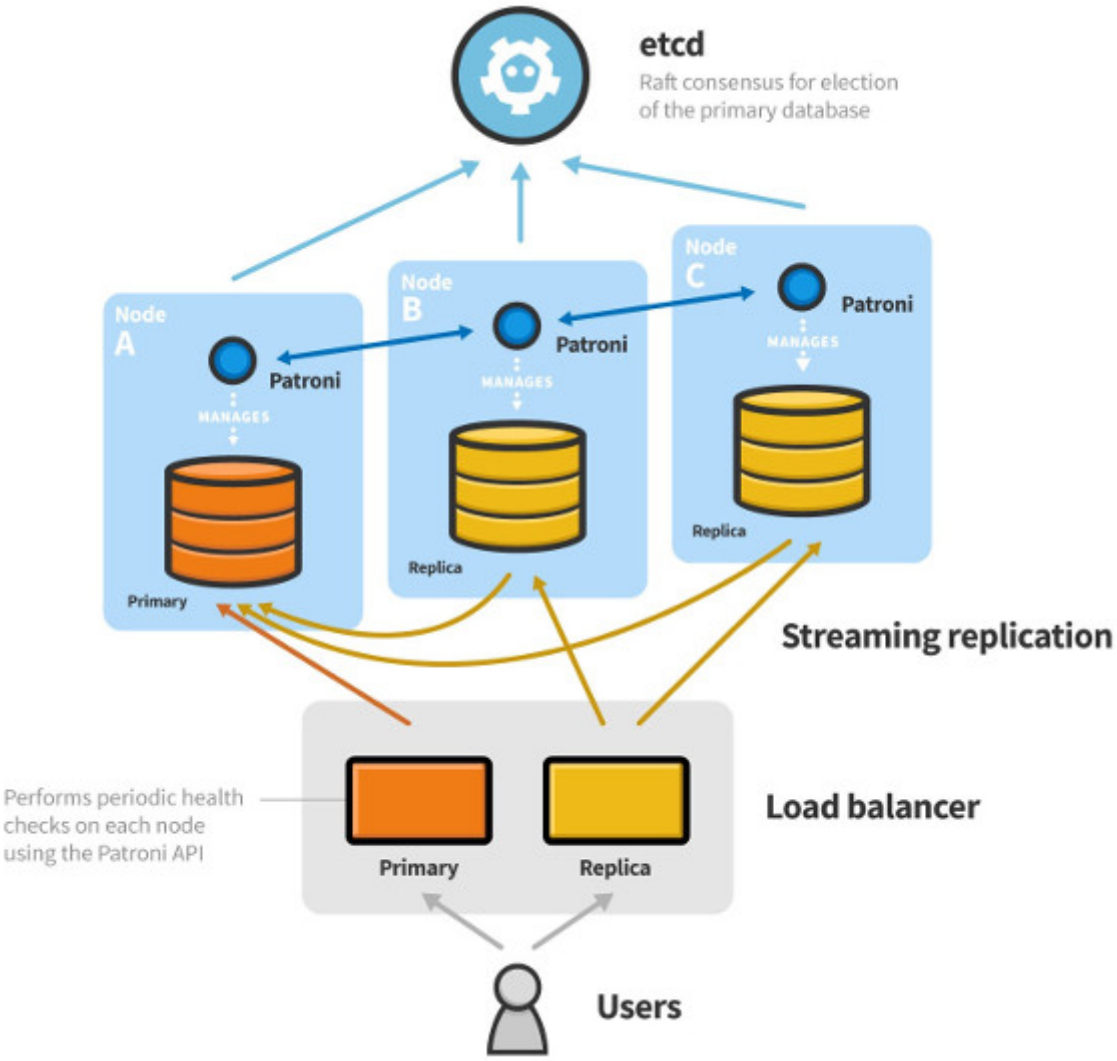
USE CASE: HA DATABASE!

- Databases provide high availability through service load balancing (clustering) and data replication (synchronizing).
- DB clustering needs to know who the cluster partners are
- Does not do well with short-lived hostnames or shared storage

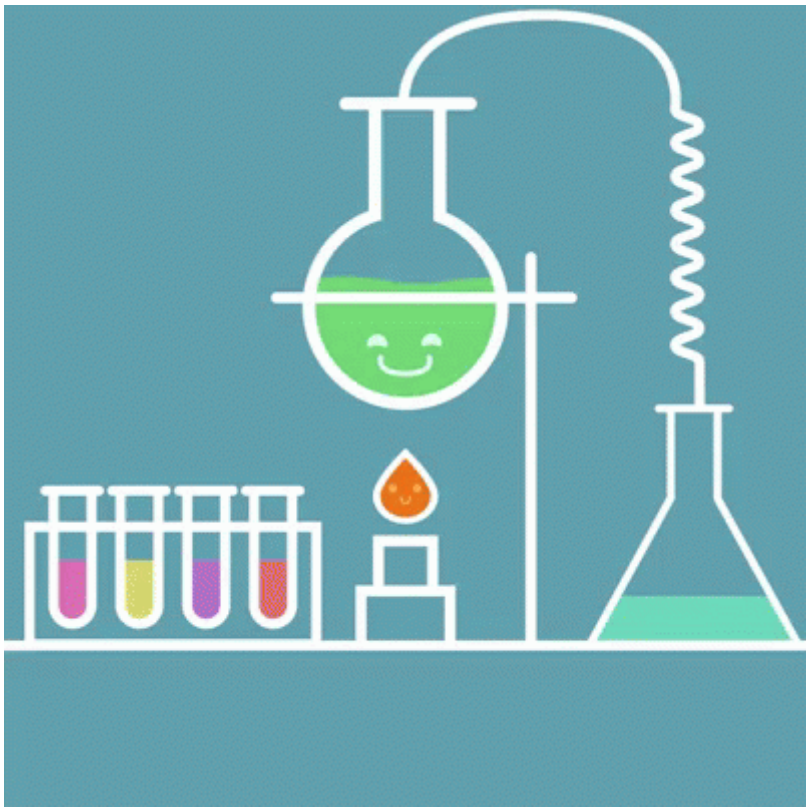
Patroni is **NOT** a postgres cluster, but rather a template for you to create your own customized, high-availability solution and a toolkit to help:

- manage the postgres cluster configuration
- manage the openshift service mappings to match the postgres cluster state

PATRONI ARCHITECTURE

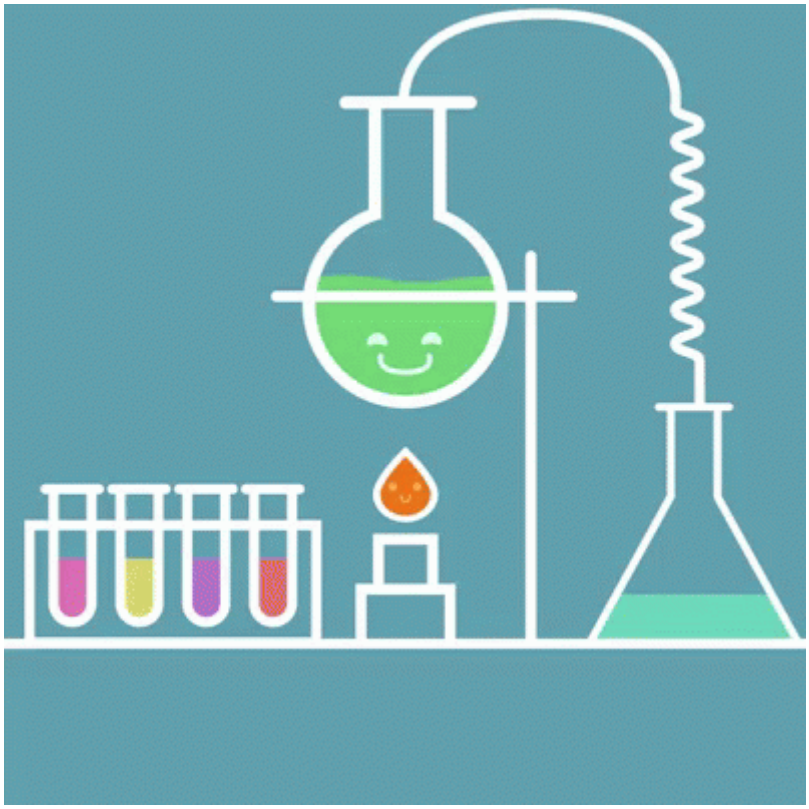


LAB: CREATE PATRONI POSTGRES CLUSTER



LAB: DB CONFIGURATION CHANGES

Break for Lab



WHO NEEDS DBAS?

We do! DBA functions are still a thing.

What role has taken on the DBA functions?

ARCHITECT YOUR DB SERVICE

- Synchronous or asynchronous updates between cluster members?
- How many replicas do you need?
- Load-balancing reads across slaves?
- Supported tools?

OPERATING YOUR DB SERVICE

- Backups and restores
- Tweaking your DB settings (Dynamic, Local, Environment)
- Growing storage
- Troubleshooting when things go wrong

DATABASE BACKUPS

WHAT COULD GO WRONG?

HMMM...

GitLab.com melts down after wrong directory deleted, backups fail

Upstart said it had outgrown the cloud – now five out of five restore tools have failed

By Simon Sharwood 1 Feb 2017 at 02:02 143 SHARE ▼

Source-code hub GitLab.com is in meltdown after experiencing data loss as a result of what it has suddenly discovered are ineffectual backups.

DON'T START WITH A BACKUP

Define your recovery scenarios, and then let those scenarios drive the backup requirements

- **Service failover**: When a service fails, have another copy ready to go immediately
- **Change rollback**: When our rollout goes bad, be able to rollback to the last known good
- **Migration**: Moving somewhere else, be able to take our service with us
- **Disaster recovery**: When a disaster strikes, have the ability to get some amount of functionality back
- **Archive Retrieval**: Have the ability to look back in time

EXAMPLE BACKUP TYPES

- **Service failover**: synchronized copy used as a failover target (HA Databases)
- **Change rollback**: Short term backup used as a rollback point. (Single Version export)
- **Migration**: Used to move data from one system into another. Can be made up of multiple backup/restore moments and generally lives a bit longer than a change rollback. (Single Version + deltas for time delayed changes)
- **Disaster recovery**: Copy retained in another location. (Single Version export stored remotely)
- **Archival**: Complete application snapshot to deploy as a copy from a different time (1 copy per time travel)

I'M SURE IT'S FINE

It sucks when your backup plan falls flat.

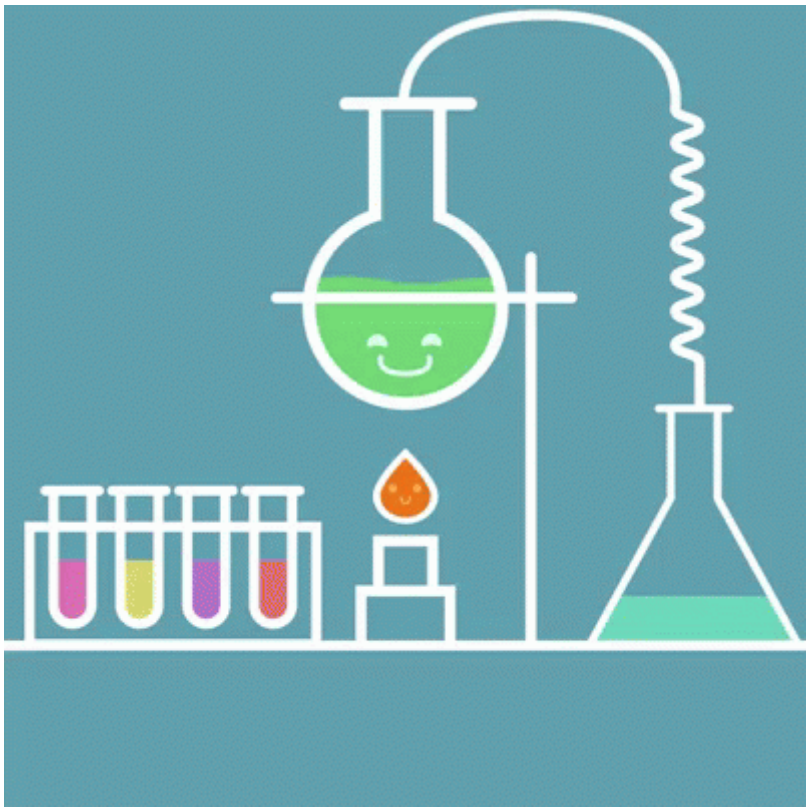
BACKUP, AND IMMEDIATELY TEST THE RESTORE

Design and implement your recovery tests at the same time that you are implementing your backup functions

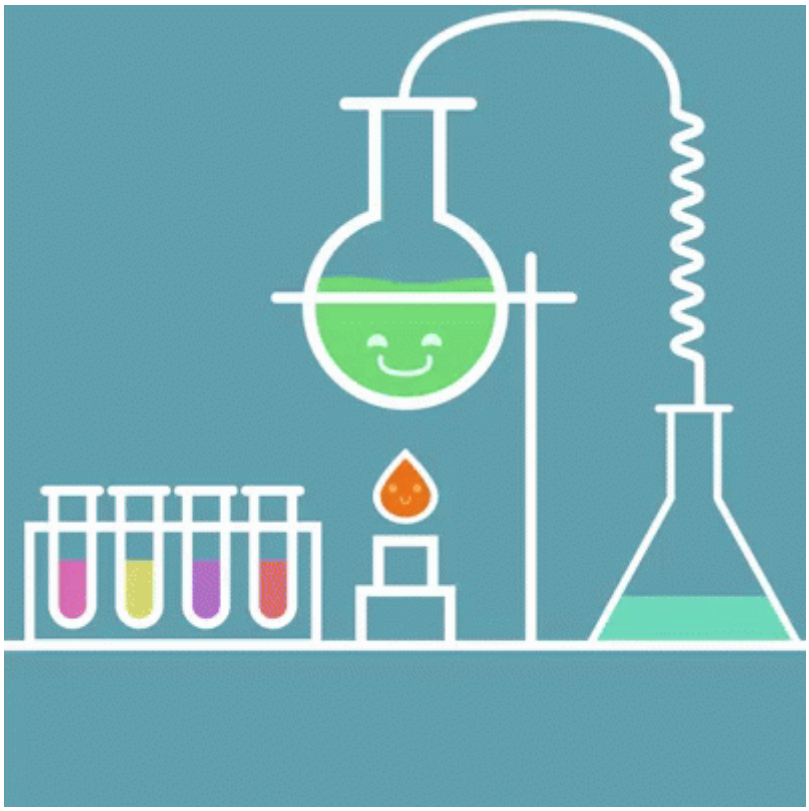
PATHFINDER OPINIONATED BEST PRACTICES

- Build resilient databases using HA DB clusters (eg: patroni stateful sets with postgres)
    - **Service Failover**: HA Failover Testing with each re-deployment
- Automate DB backup and add automated restore tests for each backup taken
    - **Database exports/imports**: Recover into temporary DB to allow immediate content validation
- Keep at least one copy of your tested DB backup outside of the cluster
    - **StorageClass nfs-backup**: External storage that is designed for self-serve integration with BCGov Enterprise backup
    - **Push to an external service**: Push/pull from your own external service

LAB: BACKUP YOUR POSTGRES



LAB: FAILOVER SCENARIOS



POD HEALTH, REQUESTS, LIMITS AND QUOTAS

HEALTH CHECKS

- **readinessProbe** checks whether a pod is ready/able to service requests
    - Action: add/remove the pod from the endpoint controller (service)
- **livenessProbe** checks whether the container is running.
    - Action: kill the pod and restart
- **initialDelaySeconds**: starts from container start
- **timeoutSeconds**: only applicable to external checks

HTTP AND TCP SOCKET CHECKS

HTTP check is successful when response is between 200 and 399

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

TCP Socket check is successful when a connection is successful

```
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

EXEC CHECKS

- Executes a command inside the container
  - Simple exec checks can be defined in the Probe definition
  - Complex scripts can be included in the build or mounted and executed as long as they return 0 for ok, and 1 for not ok.
  - Exec probes cannot define the timeoutSeconds, instead any timeouts would need to be added to the custom command/exec.

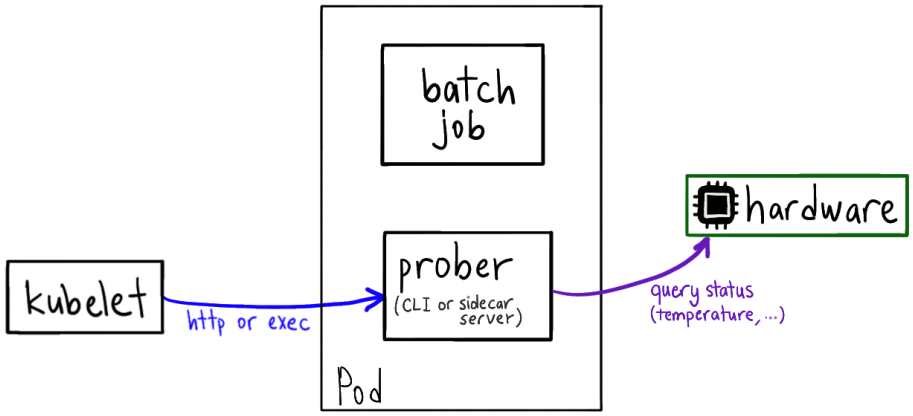
```
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/health
  initialDelaySeconds: 15
```

```
livenessProbe:
  exec:
    command:
    - timeout 10 /app/mycustom_probe.sh
  initialDelaySeconds: 15
```

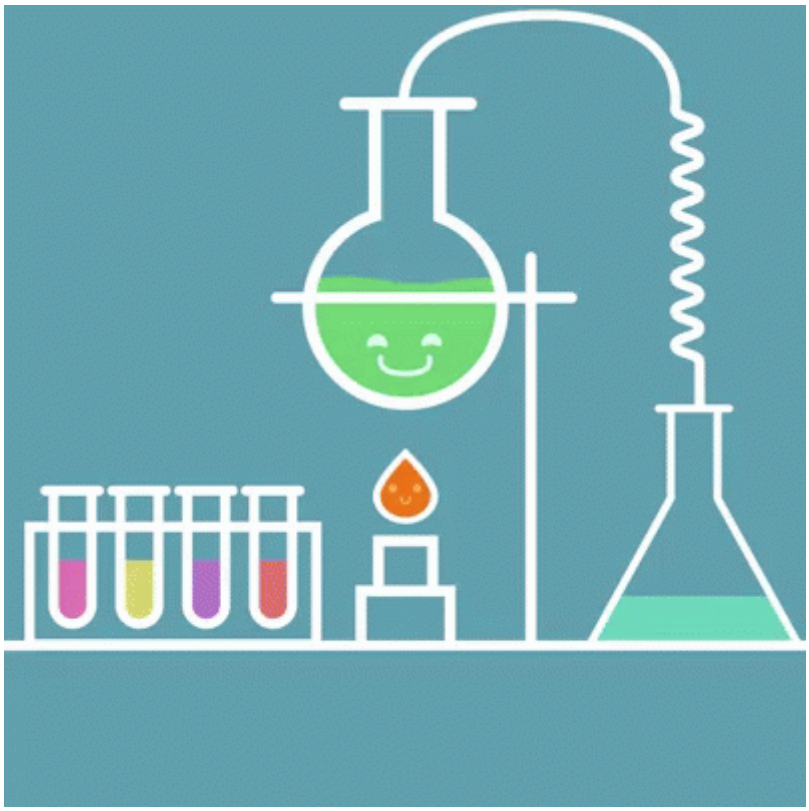
ADVANCED HEALTH CHECK

Example: Machine learning jobs heating up the GPU after a while, causing slower computation. Moving the job to a different node can remedy this.

- Add a probe CLI tool (or a sidecar server) that queries the hardware status (GPU Temp), fail when a threshold is passed. The new Pod will be rescheduled (to another physical node).



LAB: CREATE A CUSTOM HEALTH CHECK



QUOTAS

- Applied by cluster administrators to constrain the number of objects or amount of resources used in a namespace.
- **ResourceQuota** applies limits to resource consumption and/or number of objects per project
- **LimitRange** applies compute resource limits and/or defaults for individual objects per project

```
oc get quota
oc describe quota {name}

oc get limitrange
oc describe limitrange {name}
```

OVER CONSUMPTION

Within a shared environment, hoarding is harmful to the whole. The closer actual consumption is to requested resources, the better the platform can:

- schedule workload
- anticipate growth patterns for expansion
- reduce overall cost/waste

REQUESTS AND LIMITS

Vertical scaling with Requests and Limits

Ensure that the following snip is included in every container object definition:

```
spec:
  containers:
    image:
    ...
    resources:
      requests:
        cpu: 100m
        memory: 512Mi
      limits:
        cpu: 250m
        memory: 1Gi
```

QUALITY OF SERVICE TIERS

Based on the request and limit values specified for each resource:

- **BestEffort** when request and limit are not specified
  - can consume as much of the resource as available on the node, but runs at the lowest priority
- **Burstable** when request is lower than limit
  - guaranteed the requested amount, shares the rest of the node's resources equally when bursting up to the *limit* amount
- **Guaranteed** when request and limit are the same
  - guaranteed the amount of resources requested, no access to additional resources on a node

EXPLORING OBJECTS

OPENSHIFT API OBJECTS

The best place to find documentation for your objects:

```
oc api-resources

oc explain {objectType}

oc explain {objectType}.{field}
```

INTERESTING OBJECTS

- **cronjobs** are k8s scheduled jobs that manage the sc
  https://github.com/BCDevOps/backup-container/blob/master/openshift/templates/
- **routes** are the method used to expose services outsi
  - BlueGreen example: **https://docs.openshift.c platform/3.11/architecture/networking/ro**
  - Annotations for security: **https://docs.openshi platform/3.11/architecture/networking/ro**
- **poddisruptionbudgets** are used to define the max collection of pods.