Kubernetes & Helm

How Cradlepoint uses Kubernetes and Helm to simplify code deployment.

Boise Cloud Computing Meetup, September 25 2018

Matt Messinger *Matt Howell* *Daniel Lubovich*
@BoiseMatt

cradlepoint

Context: What and Why?

Matt Messinger

cradlepoint

What does Cradlepoint build?

NetCloud Manager
Network Management | Analytics | Security

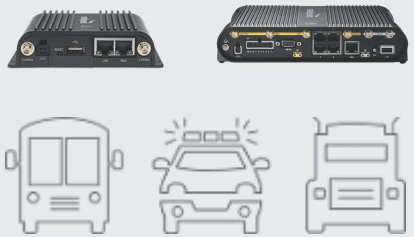
NetCloud Perimeter
Software-Defined Perimeter | Virtual Networks

cradlepoint NetCloud™

amazon web services™

Branch Networks
SD-WAN | Edge Security | WLAN

IoT Networks
M2M | Edge Compute



Mobile Networks
SD-WAN | WiFi | Telematics

Industry leader in 4G/LTE network solutions

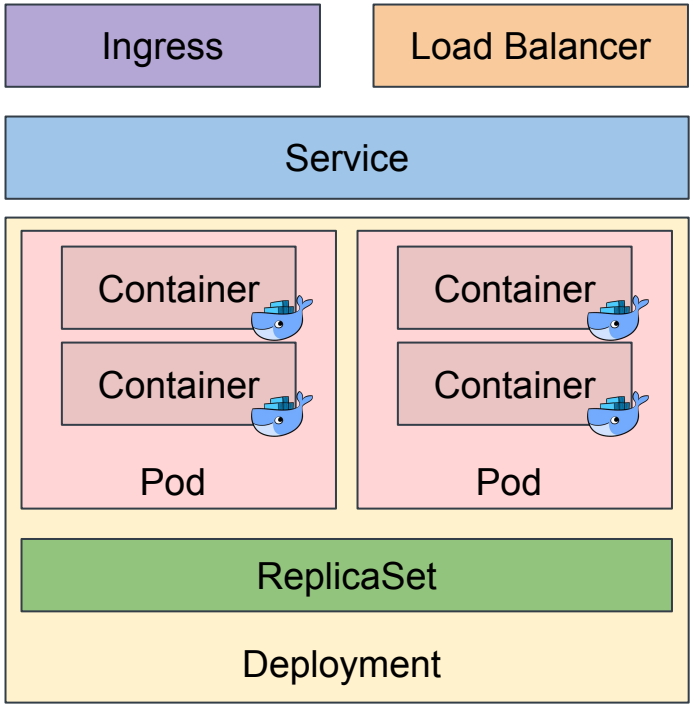
How Cradlepoint Uses Kubernetes

Matt Howell

cradlepoint

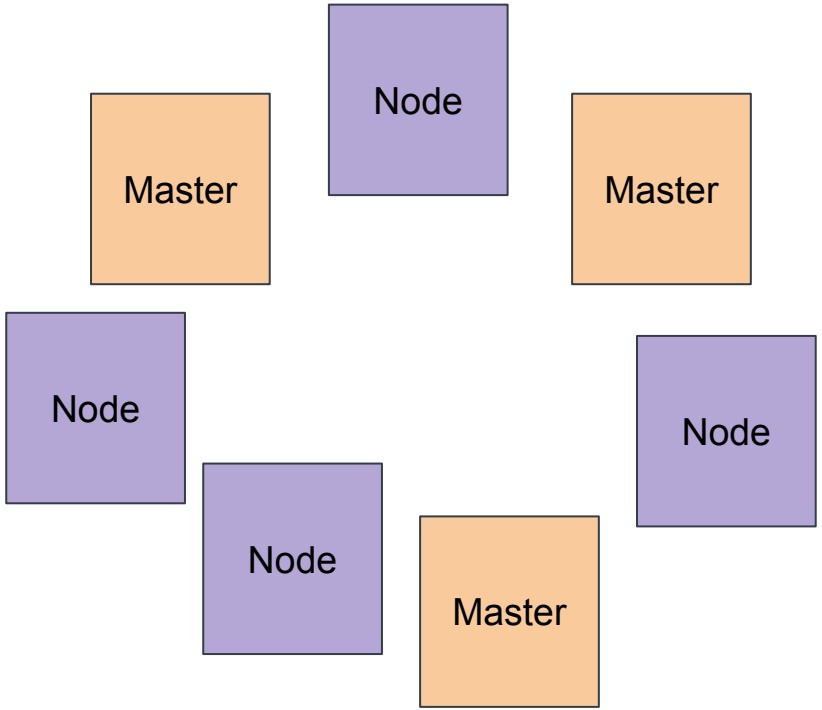
Kubernetes overview - basic resources

- **Ingress**
  - define external access to Service for non-LoadBalancer Services
- **Service**
  - represent a network endpoint for a set of identical Pods
  - LoadBalancer type also has a Google LB or AWS ELB
- **Deployment**
  - ReplicaSet of Pods *plus* upgrade strategy, security, ...
- **ReplicaSet**
  - monitor Pod health and ensure correct number of Pods
- **Pod**
  - 1+ containers, definition of Volumes, ...
- **Container**
  - (docker) image, environment, health checks, ...



Kubernetes overview - the cluster

- **Kubernetes Master**
  - api for resources
  - monitors state of defined resources
  - works to make sure they are in desired state
- **Nodes**
  - worker machines (VMs, physical)
  - runs “workflows” (containers)



Kubernetes implements desired resource state

- resources described in yaml or json, Kubernetes cluster “makes them so”
  - **Deployment**: containers, ports, volumes, replicas, update policy, authz, readi/live-ness, etc
  - **Service**: cname for group of containers, port mapping, possibly cluster ingress
  - **Job**: container to run at startup or other lifecycle stages, or at a cron interval
  - **ConfigMap**: configuration key-val’s or files
  - **Secret**: like a ConfigMap but with extra authz
  - Ingress, PersistentVolumeClaim, PodDisruptionBudget, ServiceAccount, Role, RoleBinding, ...

- lots of yaml files… leads to questions like:
  - how do you organize the yaml files?
  - is there any way to templatize them?
  - how do you track which have been installed?
  - how is configuration handled?
  - are secrets protected?

Helm simplifies and enhances using Kubernetes

- Helm is the **package manager** for Kubernetes
- Helm enables (golang) **templating**
- Helm has 2 parts: a *client* `helm` and cluster-resident *server* tiller
  - `helm` *client*
    - operates on a **chart** - directory with standard files and subdirectories
    - chart operations have a **release** - name either specified or autogenerated
    - assembles dictionary from values and helm-isms like ReleaseName and ChartName
    - templatizes files in the templates directory with dictionary into yaml
    - these yaml files are sent to tiller in context of the release
  - tiller *server*
    - receives release yaml files, converts to JSON and sends to kubernetes API
    - makes record of this release: its name, version, and kubernetes resources
- **Helm releases** can be installed, upgraded, rolled-back, and deleted

Helm chart structure and Cradlepoint profiles

- `$ helm create foo` will create this chart directory

```
foo/Chart.yaml          # name and version of chart “foo”

foo/requirements.yaml   # list of “required” charts/versions that are
                        # also installed when “foo” is installed

foo/values.yaml         # defines a dictionary that can be used by golang templating
                        # in templates/* (below).  This dictionary can be extended
                        # with  -f overrides.yaml and/or --set foo.bar=newval.

foo/templates/          # directory of files that are templated and sent to tiller
                        # on helm install or upgrade
```

- Cradlepoint **“profiles”** are subdirectories of profile-specific *overrides*
  - e.g., `foo/profiles/[dev,qa,prod]/values.yaml`
  - `$ helm install foo -f foo/profiles/qa/values.yaml`
- Cradlepoint keeps each helm chart (including **profiles**) with its associated service source repository
  - wait... what about secrets?????

Secrets: SOPS + KMS

- SOPS is editor of encrypted files
  - supports YAML, JSON, or BINARY formats
  - integrates with various encryption technologies (AWS KMS, GCP KMS, Azure Key Vault, etc)
  - https://github.com/mozilla/sops/blob/master/README.rst
- AWS **KMS** is AWS's key management service
  - secret key is only accessible by the service
  - IAM roles control key and action (encrypt/decrypt) access
- Cradlepoint **profiles** subdirectories also include profile-specific *secrets*
  - created with SOPS
  - YAML formatted
  - configured to use AWS KMS via a particular AWS IAM role
  - developer and pipeline IAM users given access to this role

⇒ code and configuration (charts, config & encrypted secrets) are versioned together

Helm Umbrella Chart

- an “**umbrella chart**” is a chart that brings in other charts via its requirements.yaml
- umbrella chart can set dictionary values for requirement charts

```
values.yaml
===========
requirement1: # passed to requirement1
  foo: bar
  animal: dog
requirement2: # passed to requirement2
  foo: bar
```

- Cradlepoint creates an umbrella chart for each collection of service charts that have all passed certain quality - i.e., each umbrella is a group of service charts that work together

Cradlepoint service-framework

- Cradlepoint has over 20 charts - and that number continues to grow…

- Cradlepoint created **service-framework** helm chart to
  - minimize duplicated chart template “code” (and likelihood of “duplication mistakes”)
  - commonize best practices (labeling, RBAC, upgrades, anti-affinity, etc)
  - service charts only specify values and resources pertinent to their service

  ⇒ reduce service developer chart responsibilities - not everyone needs to be a kube expert!

- service charts include service-framework via their requirements.yaml

- note: Cradlepoint is working to open source service-framework - github.com/cradlepoint

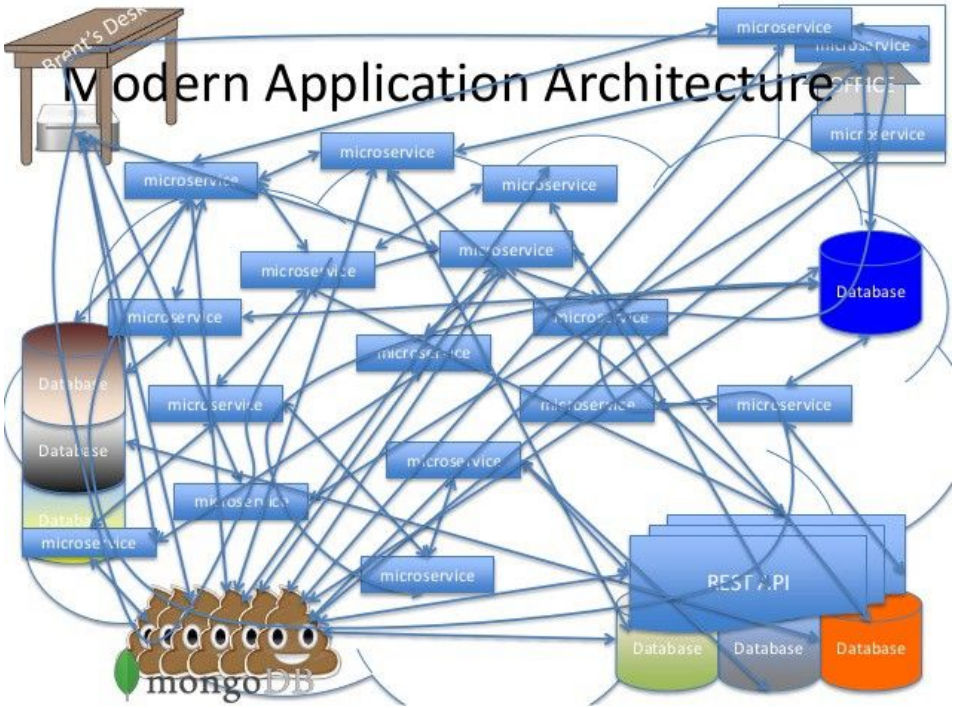
Developer Workflow

Daniel Lubovich

cradlepoint

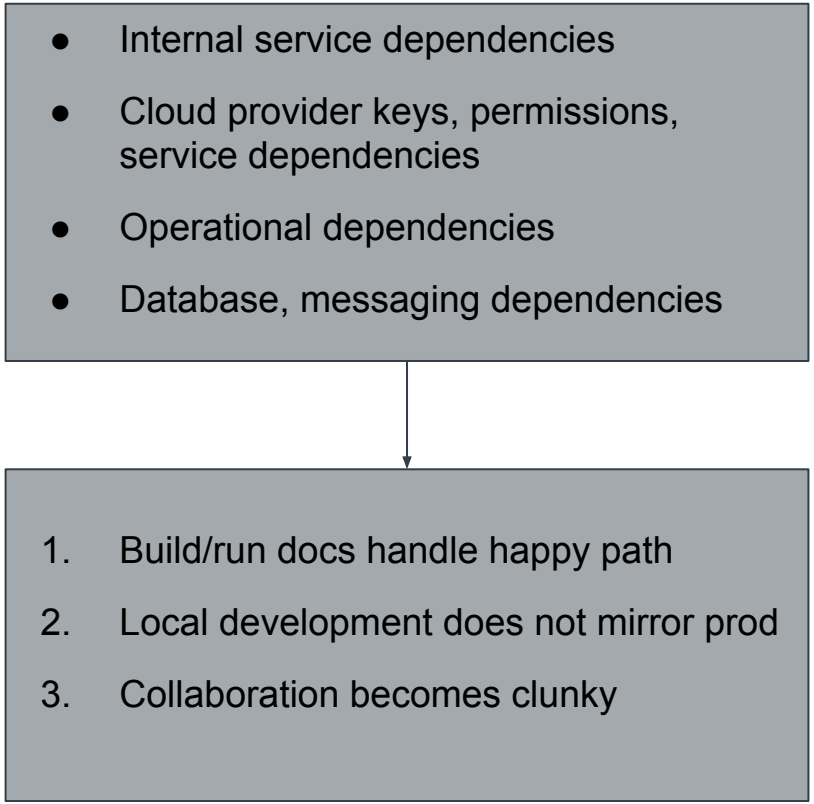
Developer Workflow & Common Pain Points

- **Complicated local development**
- Complicated deployment
- Complicated qualification



Developer Workflow & Common Pain Points

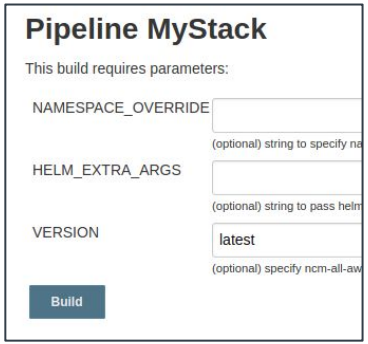
- **Complicated local development**
- Complicated deployment
- Complicated qualification

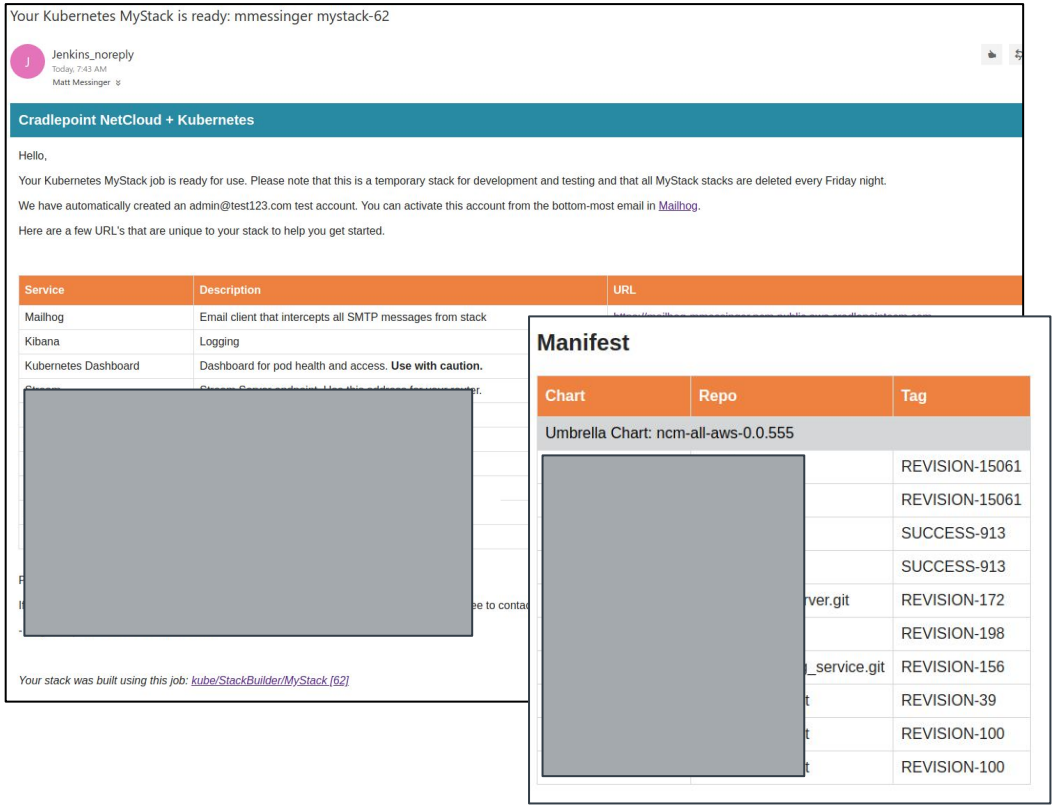


Local Development After Kube

MyStack

- No need to manage/run multiple microservices locally
- Architectural parity with production
- Solves the 'works on my machine' problem

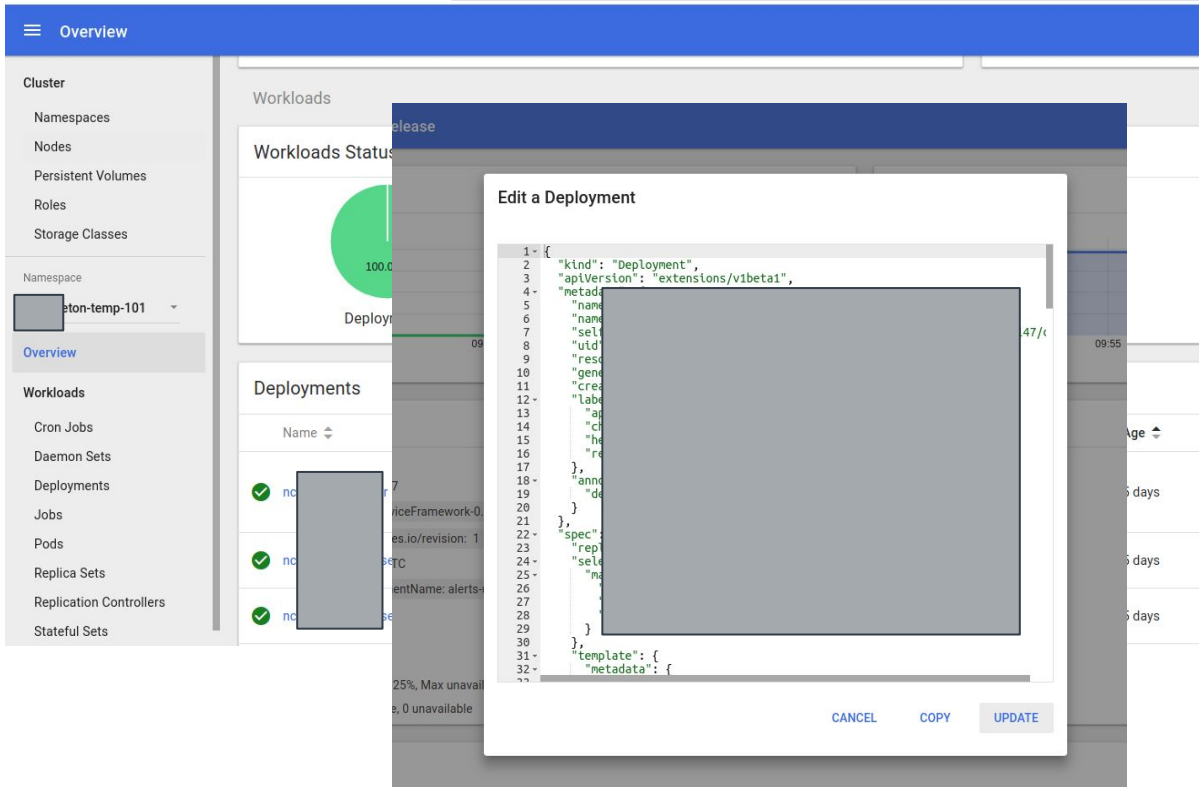




Local Development After Kube

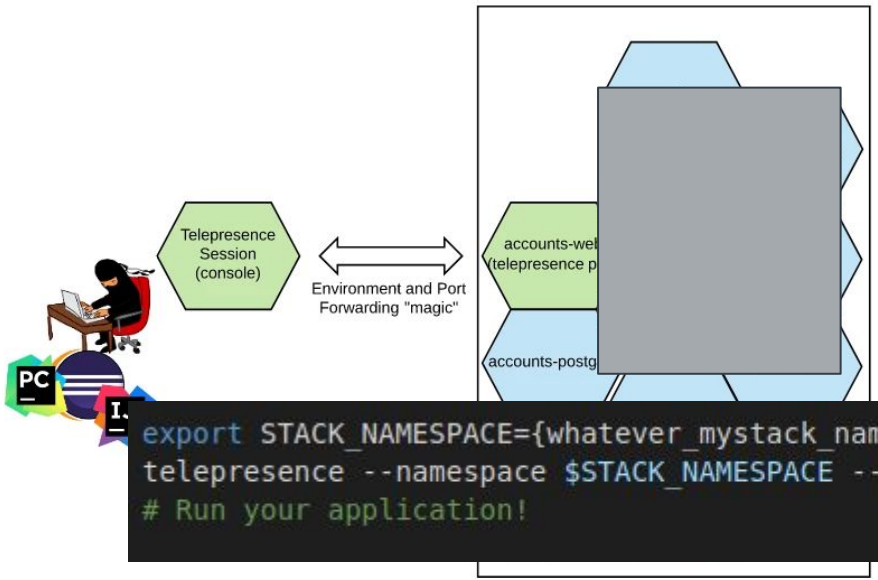
MyStack

- Dashboard gives devs:
  - Deployment health
  - Editable deployments
  - Application logs
  - Start, stop, restart pods



Local Development After Kube

Telepresence




```
export STACK_NAMESPACE={whatever_mystack_name}
telepresence --namespace $STACK_NAMESPACE --method=inject-tcp --swap-deployment accounts-web
# Run your application!
```

- Drop-and-replace services into MyStack
- Local debug
- No environment setup
- Simple to start

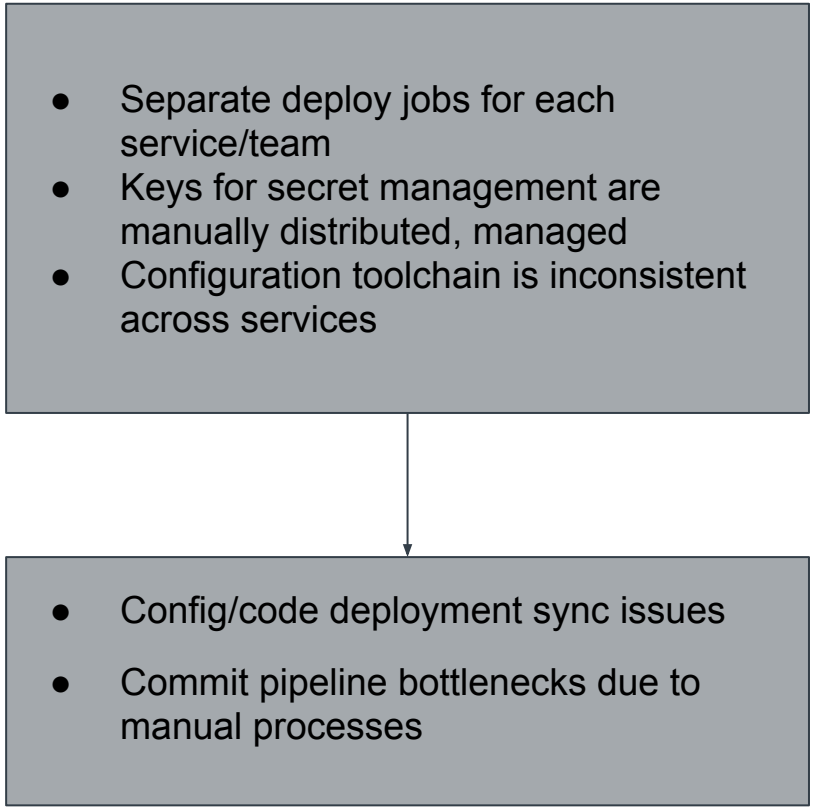
Local Development After Kube

Minikube

- Local development on steroids
  - Runs a full cluster on your machine
  - Maintains its own docker registry
- One or many services may be run
- Primarily used for testing & building new images

Developer Workflow & Common Pain Points

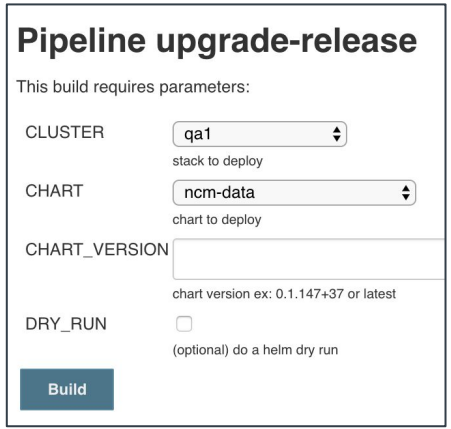
- Complicated local development
- **Complicated deployment**
- Complicated qualification



Solution to Complicated Deployment

Push-button deployment

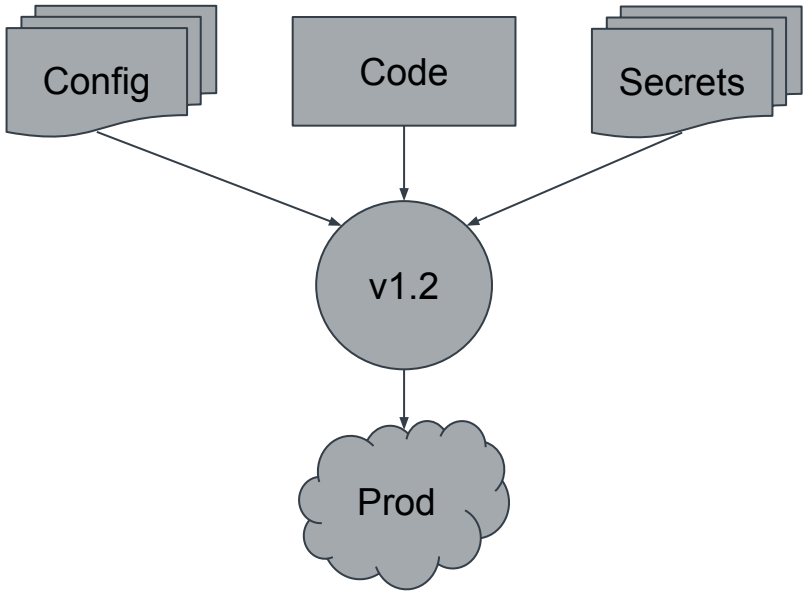
```
helm repo update
helm fetch cradlepoint/example-chart --untar
pushd example-chart/profiles/qa1/ && sops -d secrets.yaml > /mnt/ramdisk/secrets.yaml.decrypted && popd
helm upgrade --namespace example -f example-chart/profiles/qa1/values.yaml -f /mnt/ramdisk/secrets.yaml.decrypted example-release example-chart
rm /mnt/ramdisk/secrets.yaml.decrypted
```



- Deploy job works across technologies, teams
- All stacks are deployed from a single job
- More frequent, automated deploys
- Legible deployment scripts

Solution to Complicated Deployment

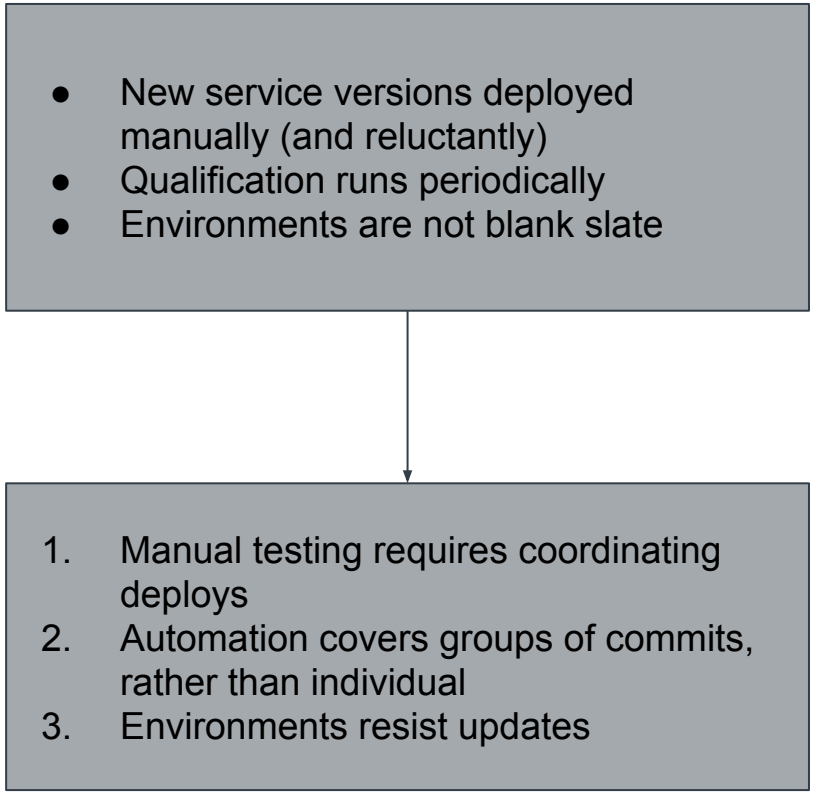
Config & Secret Management



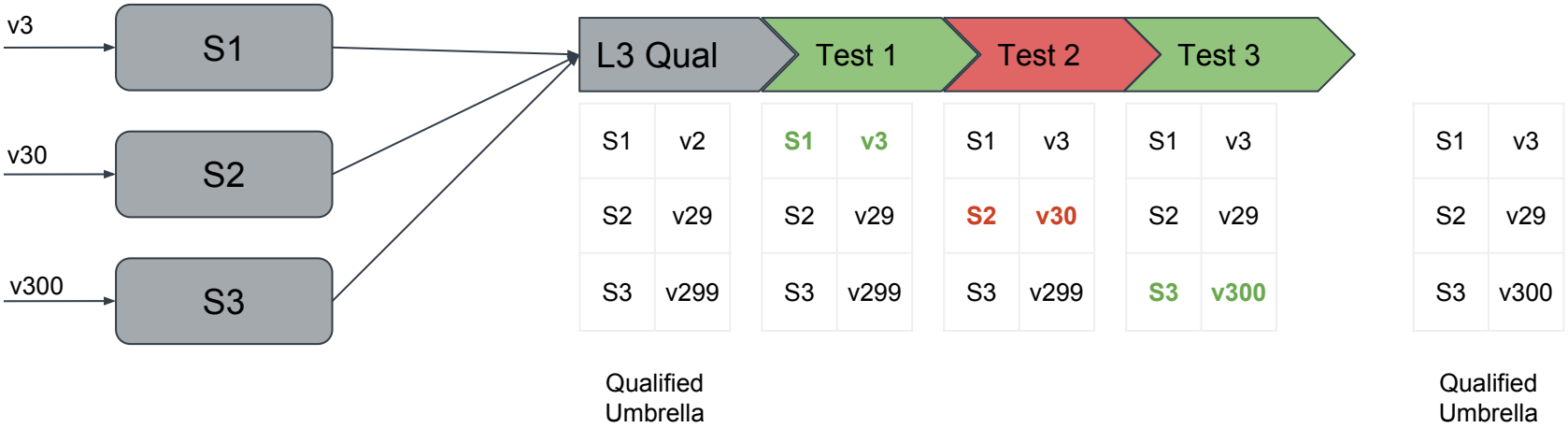
- Code, config, secrets are versioned, tested, promoted together
- Resource requirements & infrastructure defined by config
- SOPS
  - Permissions handled by IAM
  - Permissions per profile
  - Secrets are encrypted in code

Developer Workflow & Common Pain Points

- Complicated local development
- Complicated deployment
- **Complicated qualification**



Solution to Complicated Qualification



- L3 testing happens serially, after every change
- Integration runs new change against qualified services
- Result is a qualified umbrella chart
- All test stacks are blank state

Check our our R&D Blog @ cradlepoint.com/blog

Behind the Code Series: How We Moved from Monolith to Microservices

Submitted by Matt Messinger on August 9, 2018

Microservices Architecture Allows Development to Scale Horizontally

Industry-adopted best practices for building scalable web applications are always evolving. One of the most significant is the transition to microservices architecture — breaking a large single-service "monolithic" web application into smaller, independently deployable, independently scalable, decoupled web services. This is an architecture that has been proven in large companies like Netflix and pushed by industry experts like Adrian Cockcroft and Martin Fowler.

Cradlepoint is well down the road of adopting a microservices architecture for our SaaS products — specifically NetCloud Manager (NCM).

Outgrowing Our Monolith

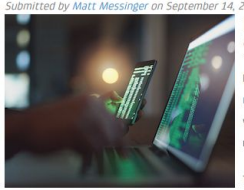
When NCM first released, it was written as a monolithic application. A single small team of engineers contributed to this codebase. As is typical in monolithic deployments, we achieved horizontal scale and high availability by replicating this application across multiple virtual machines and using a load balancer to distribute client requests to those machines.

This architecture worked well for a couple of years, but NCM's success drove new requirements to meet business demands.

Requirement	Example
We needed to be able to scale different parts of our application differently.	NCM holds persistent connections to hundreds of thousands of Cradlepoint routers at all times. This puts a heavy and nearly steady-state pressure on the router connection portion of the application. In contrast, the NCM web service that hosts cradlepointecm.com creates less load and only sees usage spikes during business hours. The router connection service and web service needed to scale independently.
	You can accomplish this with a monolith, but our router connection requirements

Behind the Code Series: How We Migrated to Kubernetes

Submitted by Matt Messinger on September 14, 2018



How Choosing Kubernetes Simplifies Code Deployment

In my previous R&D blog post I detailed Cradlepoint's adoption of a microservices architecture. The proliferation of microservices meant we had more applications that needed built, deployed, and monitored. This became an operational challenge at scale.

This post is the first in a series that discusses how we solved this challenge by migrating our service deployments to Kubernetes.

Too Many Ways to Deploy Code

Cradlepoint uses Amazon Web Services (AWS) to host both NetCloud Manager (NCM) and NetCloud Perimeter (NCP). Each of these applications is composed of many microservices — each requiring scripts and tools to deploy to AWS.

- Since both applications were developed independently (NCP was derived from the Pertino codebase), each application used different deployment tools.
- When a team created a new service, they deployed it with tools they were familiar with, or they started to use something new — either built in-house or open source.
- We never migrated older services to use new deployment tools, because the new tools were only marginally better.

Over time, we had too many ways to deploy our code. The table below illustrates the various ways we deployed a subset of our services.

Service(s)	CI Build	Artifact	Deployment Scripts (master AWS)	OS Configuration on EC2 (install system dependencies)	Service Installation on EC2	Service Configuration on EC2 (environment & secrets)
A1, A2	Docker	debian	"stack-builder" tool	Chef	Native	Chef
B1	Docker	Docker &	custom bash +	Chef	Native	Chef