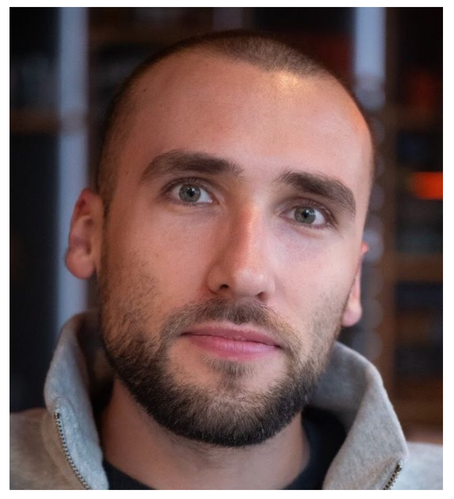
Managing Applications in Production
Helm vs. ytt and kapp
MAERSK
MAERSK

Shatarupa Nandi
shatarupan@vmware.com
@rupaNandi



Dmitriy Kalinin
dkalinin@vmware.com
@dmitriykalinin

In today's talk we will cover...

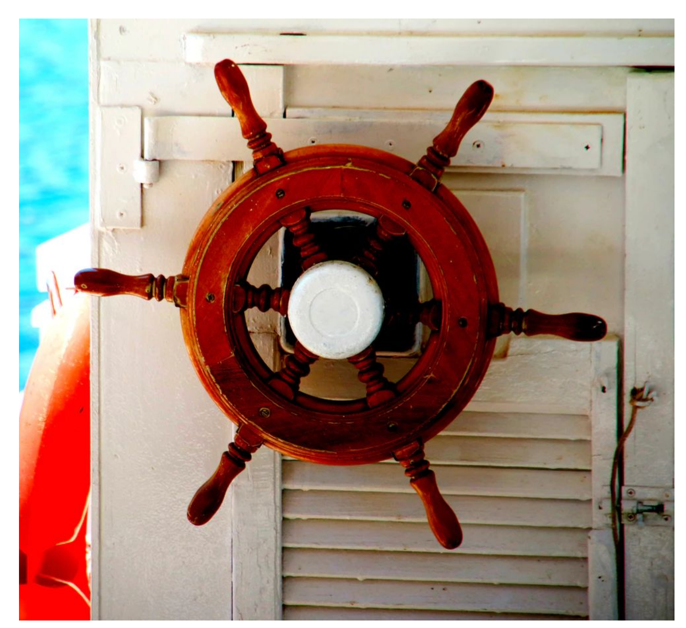
- What do we really want from deployment tools?
- How we see deployment workflow on Kubernetes
- What are some challenges we faced with Helm?
- Insights learnt from deploying in production
- Why we built kapp and ytt, and our experience using them

What do we really want from deployment tools?

- **Safety**: catch mistakes before making them in production
- **Reliability**: tool works as expected
- **Transparency**: know what's going on, while it's going on
- **Debuggability**: easy to fix when things go wrong
- **Speed**: small overhead, fast feedback loops

Ultimately, we want tools to
be boring, and **get out of our way**

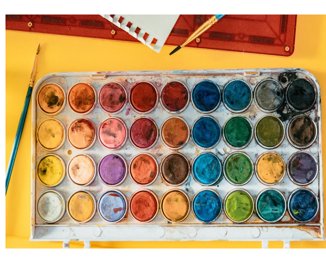
Let’s talk Helm...



End to end deployment workflow

Configuration Authoring

Packaging & Distribution



Customizing Configuration

Deploying

End to end deployment workflow

Third party

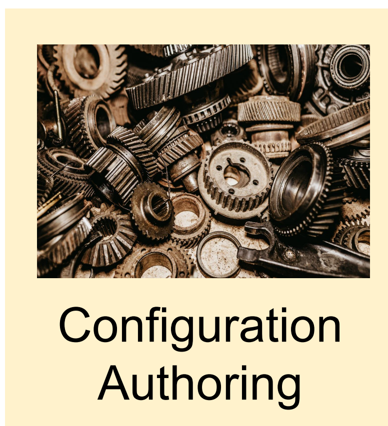
Author Configuration

Package & Distribute

Customize Configuration

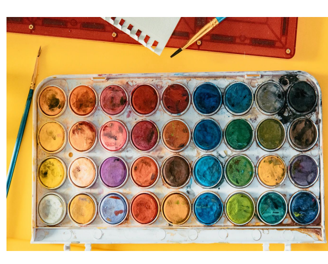
Deploy

End to end deployment workflow



Configuration Authoring

Packaging & Distribution



Customizing Configuration

Deploying

Text templating

- Figuring out how to indent text chunks

```
spec:
  template:
    metadata:
      annotations:
        {{ toYaml $.Values.pod.annotations | indent 8 }}
```

- Quoting inserted values correctly

```
data:
  REGION: {{ .Values.region | quote }}
```

Data structure templating over text templating

- ~~Figuring out how to indent text chunks~~

```
spec:
  template:
    metadata:
      annotations: #@ data.values.pod_annotations
```

Associated with a specific YAML node

- ~~Quoting inserted values correctly~~

```
data:
  REGION: #@ data.values.region
```

Functions returning structures, not text

```
#@ load("@ytt:data", "data")
#@ load("@ytt:json", "json")

#@ def config():                                   ← 1
hostPort: #@ "127.0.0.1:{}".format(data.values.jmx_port)
lowercaseOutputName: #@ data.values.output_name
#@ end

kind: ConfigMap
metadata:
  name: foo
  labels: #@ labels()                              ← 2
data:
  config.json: #@ json.encode(config())            ← 3
```

One big function...

Function
ytt templates

Parameters
ytt data values

Result
Data structure

Schema

Modify further

Serialize to YAML

Functions returning structured third-party config

Function returns ytt overlay structure

3
```
#@ def change_replicas():
#@overlay/match by=overlay.subset({"kind": "Deployment"})
---
spec:
  replicas: 3
#@ end
```

Plain YAML from Calico website, downloaded locally

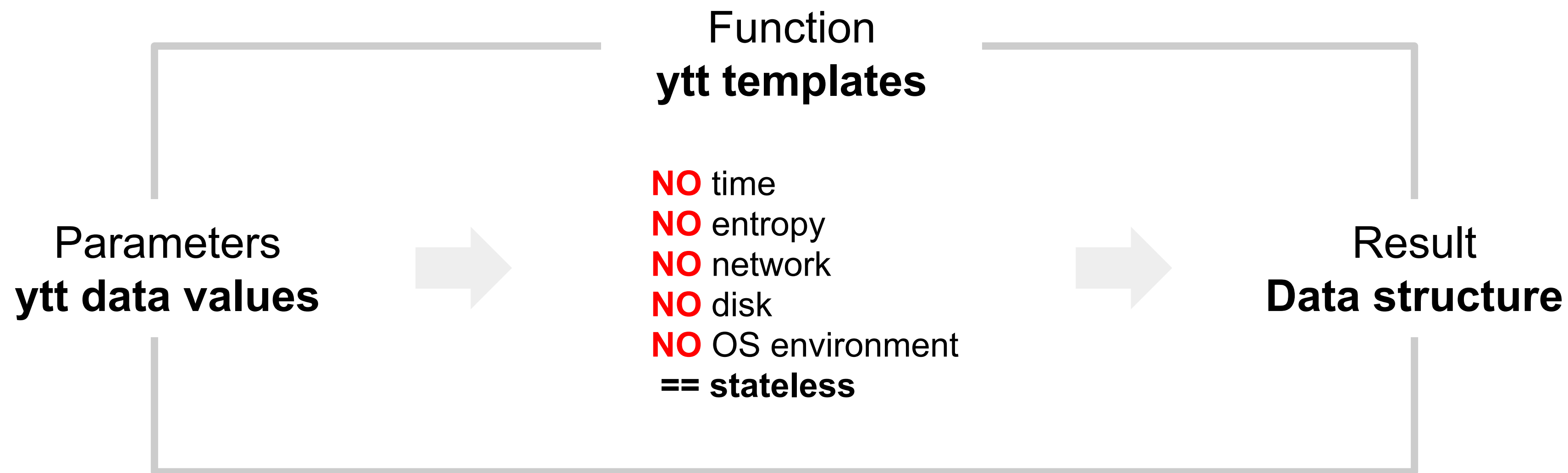
1
```
#@ calico = library.get("calico-typha").eval()
```

Programmatically apply overlay

```
#@ calico = overlay.apply(calico, change_replicas())
```
2

```
--- #@ template.replace(calico)
```

Running in sandboxed execution environment



Templating in action

```
$ ytt -f config/ -f values.yml | kubectl apply -f-
```

Extending third-party YAML configuration

1. Download the Calico networking manifest for the Kubernetes API datastore.

```
curl https://docs.projectcalico.org/manifests/calico-typha.yaml -o calico.yaml
```

2. If you are using pod CIDR `192.168.0.0/16`, skip to the next step. If you are using a different pod CIDR with kubeadm, no changes are required - Calico will automatically detect the CIDR based on the running configuration. For other platforms, make sure you uncomment the CALICO_IPV4POOL_CIDR variable in the manifest and set it to the same value as your chosen pod CIDR.

3. Modify the replica count to the desired number in the `Deployment` named, `calico-typha`.

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: calico-typha
  ...
spec:
  ...
  replicas: <number of replicas>
```

https://docs.projectcalico.org/getting-started/kubernetes/self-managed-onprem/onpremises

Extending third-party YAML configuration

```
#@ calico = library.get("calico-typha").eval()

#@ def change_replicas():
#@ content = {"kind": "Deployment", "metadata": {"name": "calico-typha"}}
#@overlay/match by=overlay.subset(content)
---
spec:
  replicas: #@ data.values.calico_replicas
#@ end

---
kind: ConfigMap
data:
  calico.yml: #@ yaml.encode(overlay.apply(calico, change_replicas()))
```

Extending third-party YAML configuration

```
#@ def config():
calico_replicas: 5
#@ end

#@ setup = library.get("calico-setup").with_data_values(config()).eval()

--- #@ template.replace(setup)
```

End to end deployment workflow

Configuration Authoring

Packaging & Distribution

Customizing Configuration

Deploying

What is in a bundle?

distribution registry

Configuration image

and

App image 1

...

App image N

Any configuration
e.g. plain YAML, ytt
templates, helm chart

Bundle location

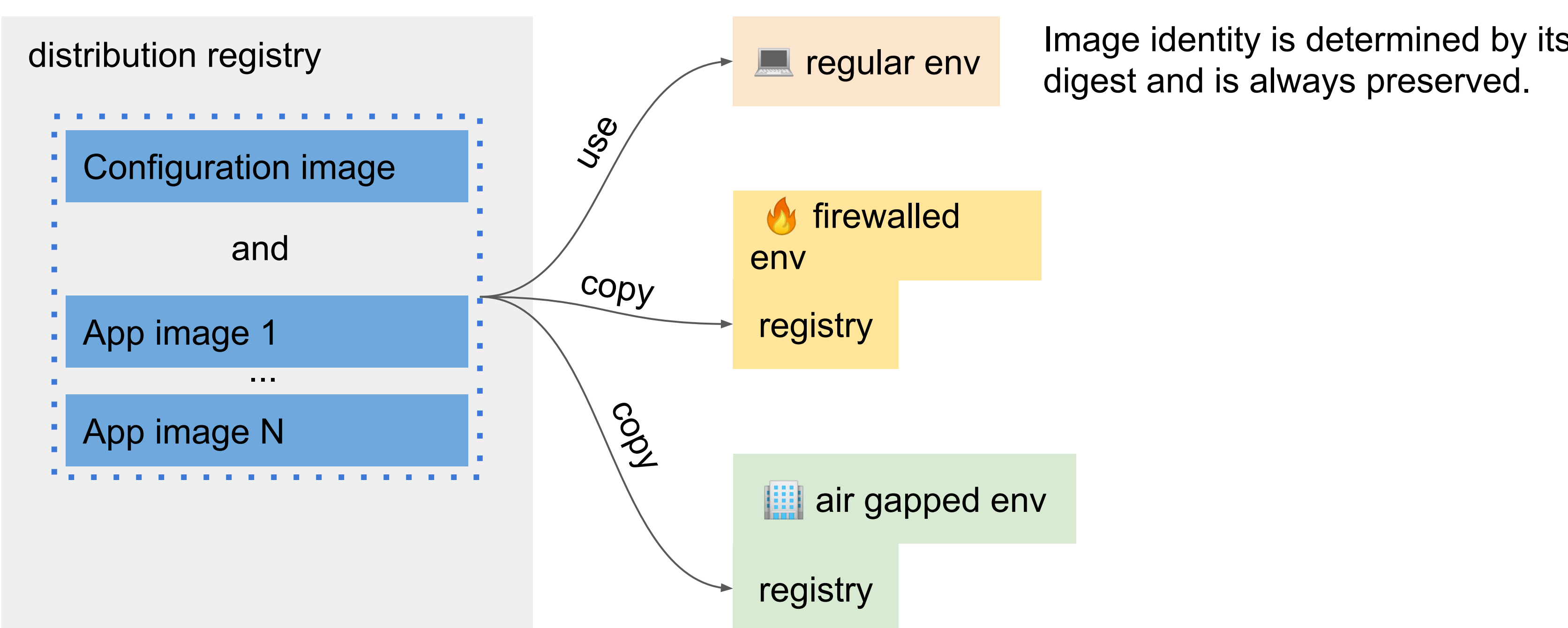


Image identity is determined by its digest and is always preserved.

Distribution in action

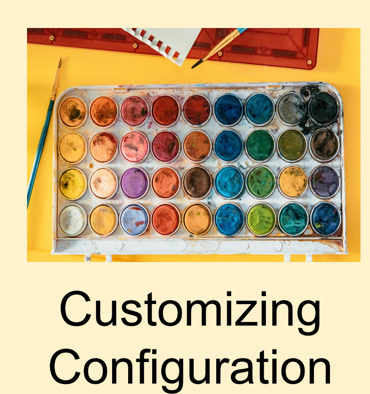
```
$ imgpkg push -b registry.corp.com/org/app1:v0.1.0 -f /app1

$ imgpkg pull -b registry.corp.com/org/app1:v0.1.0 -o /app1
$ # do something with /app1
```

End to end deployment workflow

Configuration Authoring

Packaging & Distribution



Customizing Configuration

Deploying

Explosion of configuration options

```
116  securityContext:
117    enabled: true
118    runAsUser: 1000
119    fsGroup: 1000
120
121  priorityClassName: ""
122
123  podDisruptionBudget: {}
124    # maxUnavailable: 1
125    # minAvailable: 2
126
127  nodeSelector: {}
128
129  affinity: {}
130
131  tolerations: []
132
133  extraVolumeMounts: []
134  ## Additional volumeMoun
```

→ API mirroring

Explosion of configuration options

```yaml
116  securityContext:
117    enabled: true
118    runAsUser: 1000
119    fsGroup: 1000
120
121  priorityClassName: ""
122
123  podDisruptionBudget: {}
124    # maxUnavailable: 1
125    # minAvailable: 2
126
127  nodeSelector: {}
128
129  affinity: {}
130
131  tolerations: []
132
133  extraVolumeMounts: []
134  ## Additional volumeMoun
```

Tweak configuration in right places

Function
ytt templates

(a) Parameters
ytt data values

Result
Data structure

(b) **ytt overlays** modify templating result when template parameters do not provide enough configurability

Serialize to YAML

Overlays in action

```
#@ load("@ytt:overlay", "overlay")

#@overlay/match by=overlay.subset({"kind": "Deployment"})
---
spec:
  template:
    spec:
      priorityClassName: important
```

Which YAML documents to match

What to change within YAML document

Remove size constraints for dev env

```
#@ for/end kind in ["Deployment", "DaemonSet", "StatefulSet"]:
#@overlay/match by=overlay.subset({"kind": kind}),expects="1+"
---
spec:
  template:
    spec:
    #@overlay/match when=1
    initContainers:
    #@overlay/match by=overlay.all,when="1+"
    #@overlay/match-child-defaults missing_ok=True
    -
      #@overlay/remove
      resources: {}
      #@overlay/remove
      livenessProbe: {}
...
```

If there are any init containers

Data values and overlays in action

templates

consumer overlays

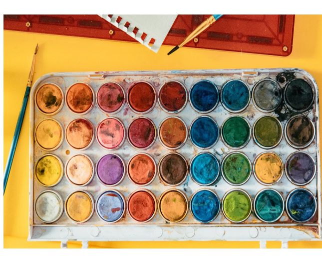
```
$ ytt -f config/ -f values.yml -f fix-priority.yml | ...
```

consumer data values

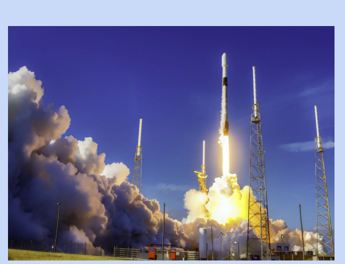
End to end deployment workflow

Configuration Authoring

Packaging & Distribution



Customizing Configuration



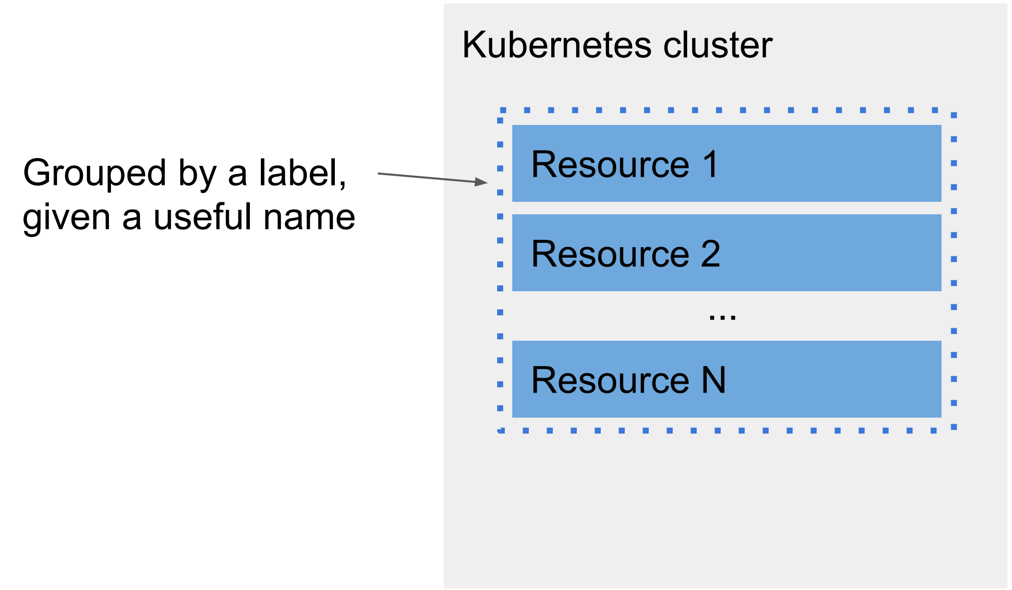
Deploying

In search for better deploys

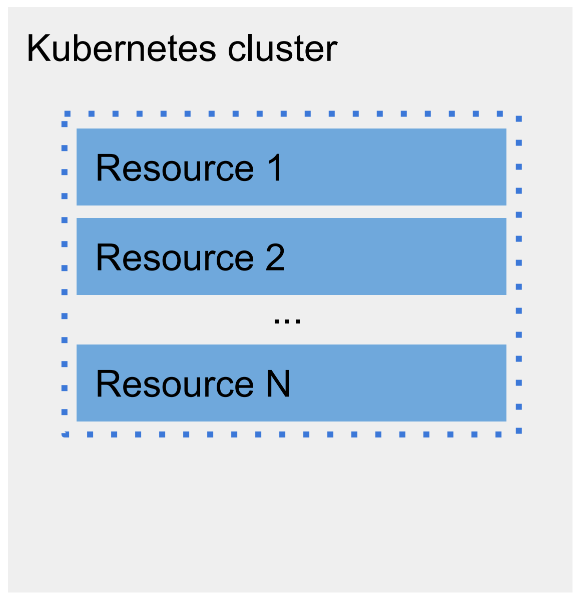
- Had challenges debugging failures during deploys, such as
  ```
  Error: UPGRADE FAILED: "..." has no deployed releases
  ```
- Lack of confidence about operations that will take place (e.g. `--force` flag may affect unintended resources)

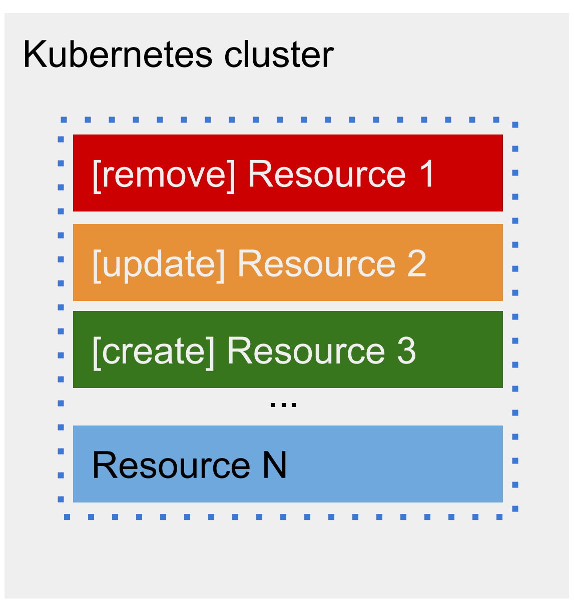
Research resulted in a lightweight deployment tool, kapp, that focuses on safety, reliability, and transparency

What is a Kubernetes application?



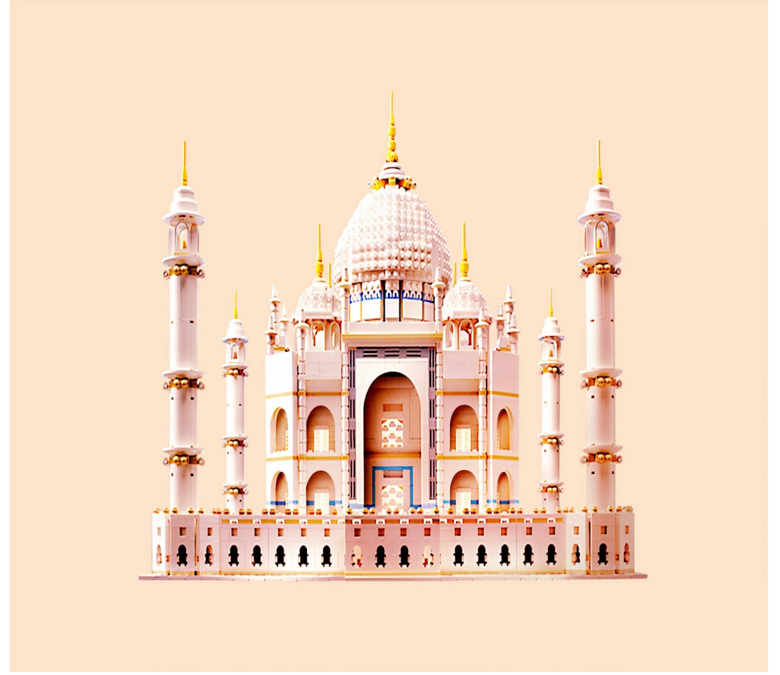
Changing Kubernetes application





Deploy in action

How does it all work together?



Client side deployment examples

```
$ ytt -f config/ -f values.yml | kubectl apply -f-
```

```
$ imgpkg pull -b registry.corp.com/org/app1:v0.1.0 -o /app1
$ kapp deploy -a app1 -f <(ytt -f /app1/config/ -f values.yml)
```

```
$ kapp deploy -a app1 -f <(helm template x --values values.yml)
```

On-cluster deployment example

```yaml
apiVersion: kappctrl.k14s.io/v1alpha1
kind: App
metadata:
  name: app1
  namespace: default
spec:
  serviceAccountName: default-ns-sa
  fetch:
  - image:
      url: registry.corp.com/org/app1:v0.1.0
  template:
  - ytt: {}
  deploy:
  - kapp: {}
```

App CRD is provided by kapp-controller

Various fetch strategies (e.g. git, http, helm fetch, Kubernetes ConfigMap)

Various template strategies (e.g. helm template, ytt)

What's next?

- If these ideas resonate with you, learn more at k14s.io, get-ytt.io, and get-kapp.io
- Share how you template and deploy today in our #k14s channel in Kubernetes Slack
- We welcome contributions to any of the projects!
- Make our day by letting us know how our tools help you