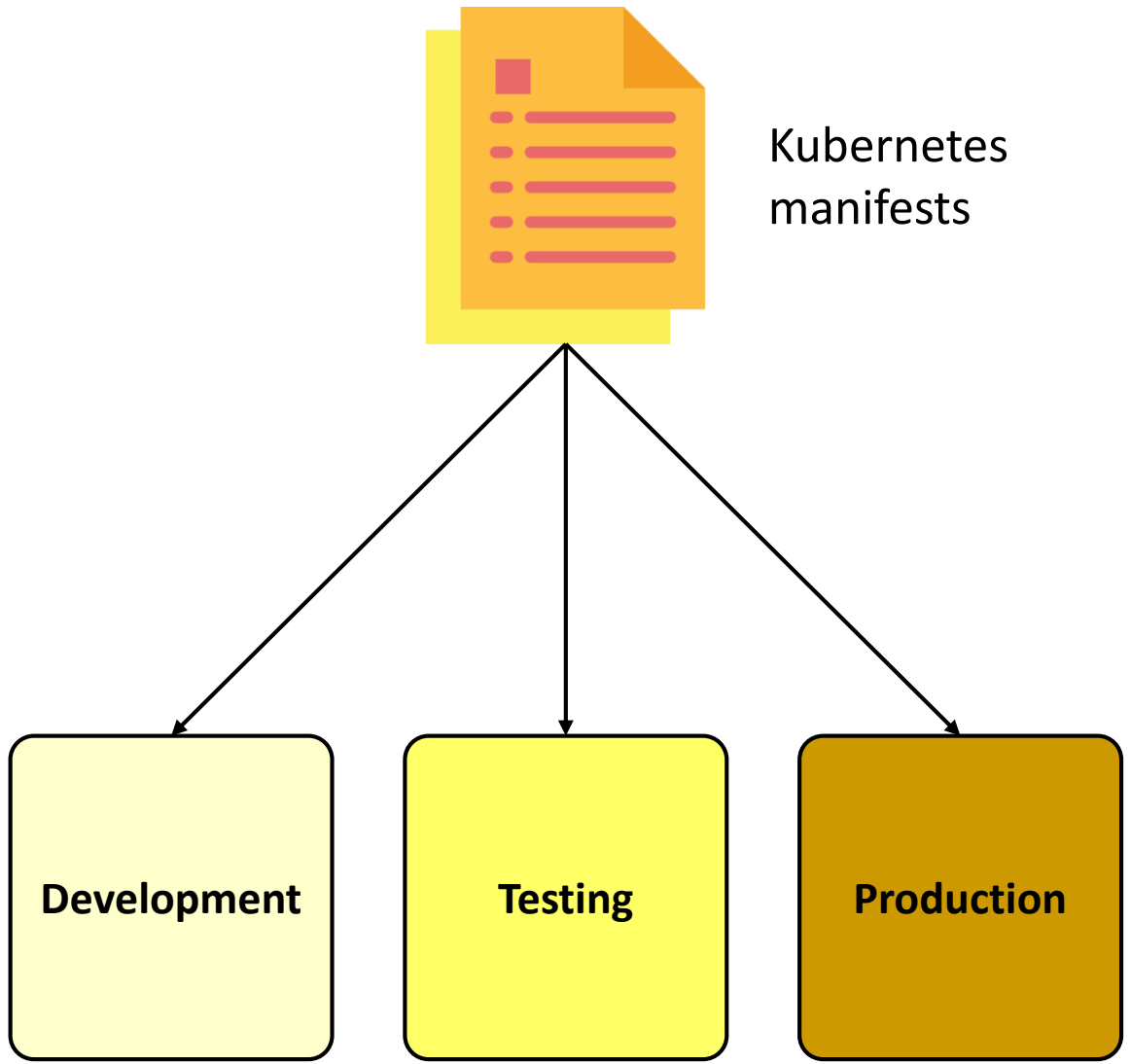
Tools for Working with Kubernetes

Kustomize

Multiple Environments

- Different environments will require different tweaks to the manifests
  - Different configurations
    - Some can be configured with `ConfigMaps` or `Secrets`
    - Some configurations cannot eg. replicas, image name
  - Different attributes
    - Require resource limits in production
    - Additional resources like cache in production



Using Kubernetes In Multiple Environments

- DRY – Don't Repeat Yourself
- Develop a Helm chart
- Nontrivial
  - Need to develop templates that cover all possible scenarios
  - Learn Go templates, template functions, conventions, etc
  - Debugging!
- Significant time investment
- Helm provides
  - Structured way of customizing deployments
  - Deployment management - install, delete , upgrades

- WET – Write Everything Twice
- Duplicate and modify the original Kubernetes manifest
- Need to actively update downstream manifest when upstream changes
  - Eg. bug fixes, improvement, etc.
- Copy and paste nature is
  - Easy to use
  - No new concepts to learn
  - Fast

Kustomize

- Kustomize is a tool for customizing Kubernetes deployments
  - Apply a set of changes to a base Kubernetes manifest
  - Produces a new set of manifest
  - Original manifest (base) is untouched/unchanged
  - Template free viz. logic less
- Kustomize allow you to consume and customize upstream Kubernetes manifest without forking them
- Open source
  - Available as standalone - https://kubectl.docs.kubernetes.io/installation/kustomize/
  - Integrated with kubectl (post 1.14) with '`apply -k`'

Customizing Kubernetes Deployments

- Kustomize transform a set of related Kubernetes manifest by adding or replacing fields and objects
  - Can delete fields but not common
- Modifications are updates can be applied to
  - All resources - eg adding a common label
  - Specific resource by providing patterns that match the resource
- Input Kubernetes manifest are specified with the `resources` field

Example - Common Fields

kustomization.yaml

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- pod.yaml
commonLabels:
  env: stage
namePrefix: eng-
namespace: stage-ns
```

---

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
  labels:
    name: app
spec:
  containers:
  - name: app-container
    image: nwapp:v1
```

```
apiVersion: v1
kind: Pod
metadata:
  name: eng-app-pod
  labels:
    name: app
    env: eng
  namespace: stage-ns
spec:
  containers:
  - name: app-container
    image: nwapp:v1
```

With kustomize

```
kustomize build | kubectl apply -f -
```

With kubectl

```
kubectl apply -k .
```

Directory with `kustomization.yaml`

Example - Deployments

```
kustomization.yaml
apiVersion: ...
kind: Kustomization

replicas:
- name: web-deploy
  count: 1

images:
- name: resilio/sync
  newName: eeacms/rsync
  newTag:2.3
- name: nginx
  newTag: 1.21.1-perl
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-deploy
spec:
  replicas: 2
  selector:
    ...
  template:
    ...
    spec:
      containers:
       - name: file-sync
         image: resilio/sync:2.7.2
       ...
       - name: nginx
         image: nginx:1.20.1
       ...
```

1

eeacms/rsync:2.3

nginx:1.21.1-perl

Patching

- Overrides or add fields to the target resource
- A list of patch files are provided
  - Order of patches applied is the order in which they are listed
- Patch file is similar to the resource to be patched
  - Should have `apiVersion`, `kind` and `name`; used by Kustomize to select the resource to be patched
  - The remainder of the patch file used to overlay on the resource viz. metadata and spec sections
  - Attributes with the same name are replaced
  - Additional attributes are added

Example - Patching

app.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    ...
  template:
    ...
    spec:
      containers:
      - name: nwapp
        image: nwapp:v2
        ports:
        - containerPort: 3000
```

app.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    ...
  template:
    ...
    spec:
      containers:
      - name: nwapp
        image: nwapp:v2
        ports:
        - containerPort: 3000
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
```

Customized resource

Add these

Example - Patching

kustomization.yaml

```
apiVersion:
kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- app.yaml

patchesStrategicMerge:
- patch.yaml
```

patch.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  template:
    spec:
      containers:
      - name: nwapp
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
```

Overlay, provide enough context to patch the original

`ConfigMap` and `Secrets`

- Can customize an existing configuration or secret specific to an environment
  - Modify an existing configuration by merging with new keys or replacing existing keys
  - Generate new configurations and secrets
- **`ConfigMap`** and **`Secrets`** behaviour
  - Pods that references values in these 2 resources are not recreated if the content changes eg. value of a secret key is replaced
    - Referenced with either `envFrom` or `valueFrom`
  - Pods that mount these 2 resources will be recreated

Example - Override Existing Configurations

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

configMapGenerator:
- name: app-cm
  behavior: merge
  literals:
  - DB_USER=barney
  - TRACE_ENABLE="1"

secretGenerator:
- name: app-secret
  behavior: merge
  type: Opaque
  literals:
  - DB_PASSWORD=mypassword
```

Add or update the values in the base

Replaced and added to the base

Replace the base

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-cm
data:
  DB_HOST: localhost
  DB_PORT: 3306
  DB_USER: fred
  DB_NAME: inventory
```

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  DB_PASSWORD: bXlzZWNyZXQ=
```

Example - Generating `ConfigMap` and `Secret`

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Config

resources:
- app.yaml

generatorOptions:
  labels:
    env: stage

configMapGenerator:
- name: app-cm
  behavior: create
  literals:
  - DB_HOST=dbserver
    DB_USER=barney
```

Add labels to all the `ConfigMap` and `Secret`

Create a `ConfigMap` call `app-cm`

One or me configurations

app.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  ...
  containers:
  - name: app-container
    image: nwapp:v1
    envFrom:
     configMapRef
       name: app-cm
```

Reference a non existence `ConfigMap`

Similar for `secretGenerator`

Example - Generating `ConfigMap` and `Secret`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-cm-bfA9Ekh0c5
  labels:
    env: stage
data:
  DB_HOST: dbserver
  DB_USER: barney
```

app.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  ...
  containers:
  - name: app-container
    image: nwapp:v1
    envFrom:
    - app-cm-bfA9Ekh0c5
```

Generated `ConfigMap` with a unique suffix.
A new suffix is generated everytime kustomize runs
Uses this `ConfigMap` when it is referenced from containers
Suffix hash forces Kubernetes to recreate the pods when configurations and secrets are updated

Replacements

- Need to know certain info from resources generated by Kustomize
  - Eg. generated a service for a database with a prefix. Need to pass this service name to a deployment that uses the database
- Used to copy fields from one source to one or more targets
  - Eg copying the service name to environment variables
- Source is the resource that will provide the value; must be a single resource
- Target(s) are the resources that the fields will be replaced
  - Include and reject sets
- Use `fieldPath` to select specific fields as source and targets
  - See https://kubectl.docs.kubernetes.io/references/kustomize/kustomization/replacements/#delimiter

Example - Replacement

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

replacements:
- source:
    kind: Service
    fieldPath: metadata.name

  targets:
  - select:
      kind: Deployment
      name: app
    fieldPaths:
    - spec.template.spec.containers.[name=myapp].env.[name=DB_HOST].value
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
spec:
  template:
    spec:
      containers:
      - name: myapp
        env:
        - name: DB_HOST
          value: old_value
```

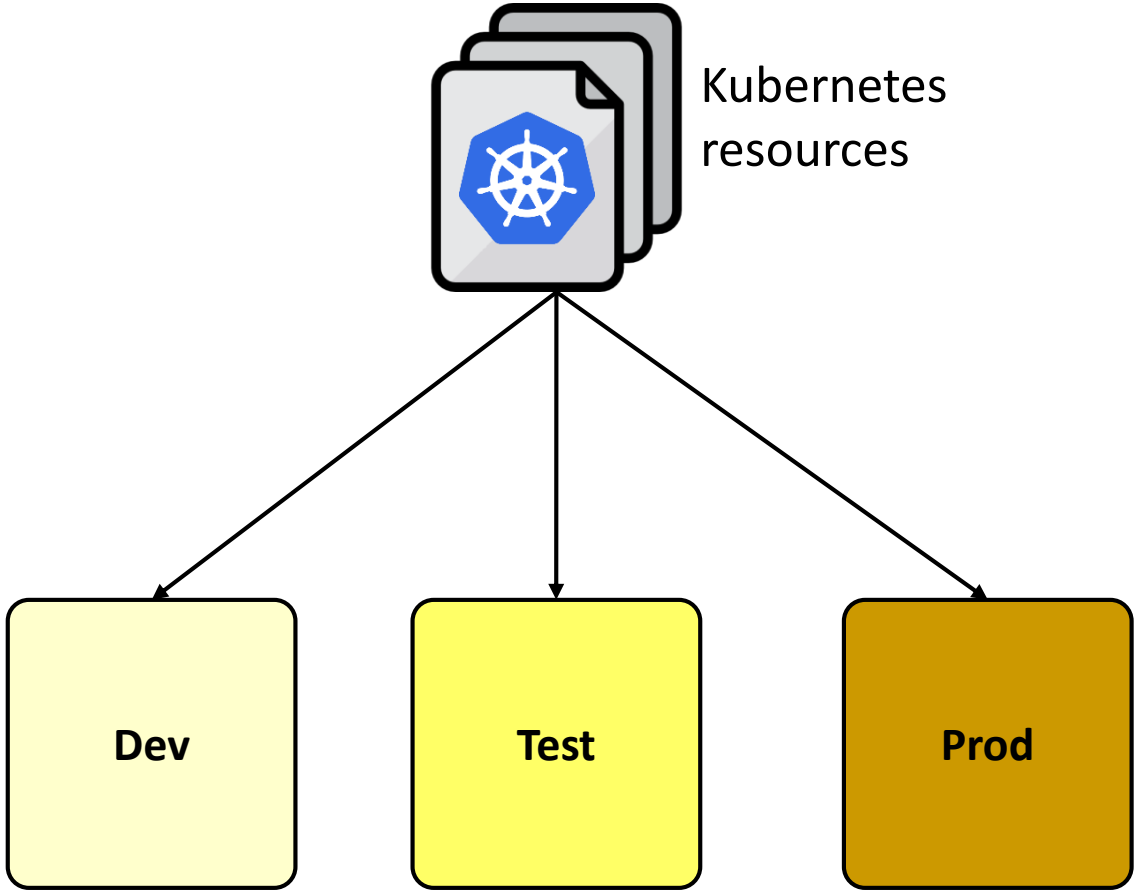
Selects this resource

Update the value of this field

Path to the value to be replaced

Directory Layout

- Create a structure to support customizing a set of Kubernetes manifest
  - Support multiple environments (eg dev, test, prod) without interfering with another environment
  - Any changes will be picked up by all upstreams viz. dev, test, prod, etc
- Base manifest
  - Generic Kubernetes manifest applicable to all environments
- Overlays
  - Kustomize configurations to be applied to the base manifest
  - One per environment



Directory Structure

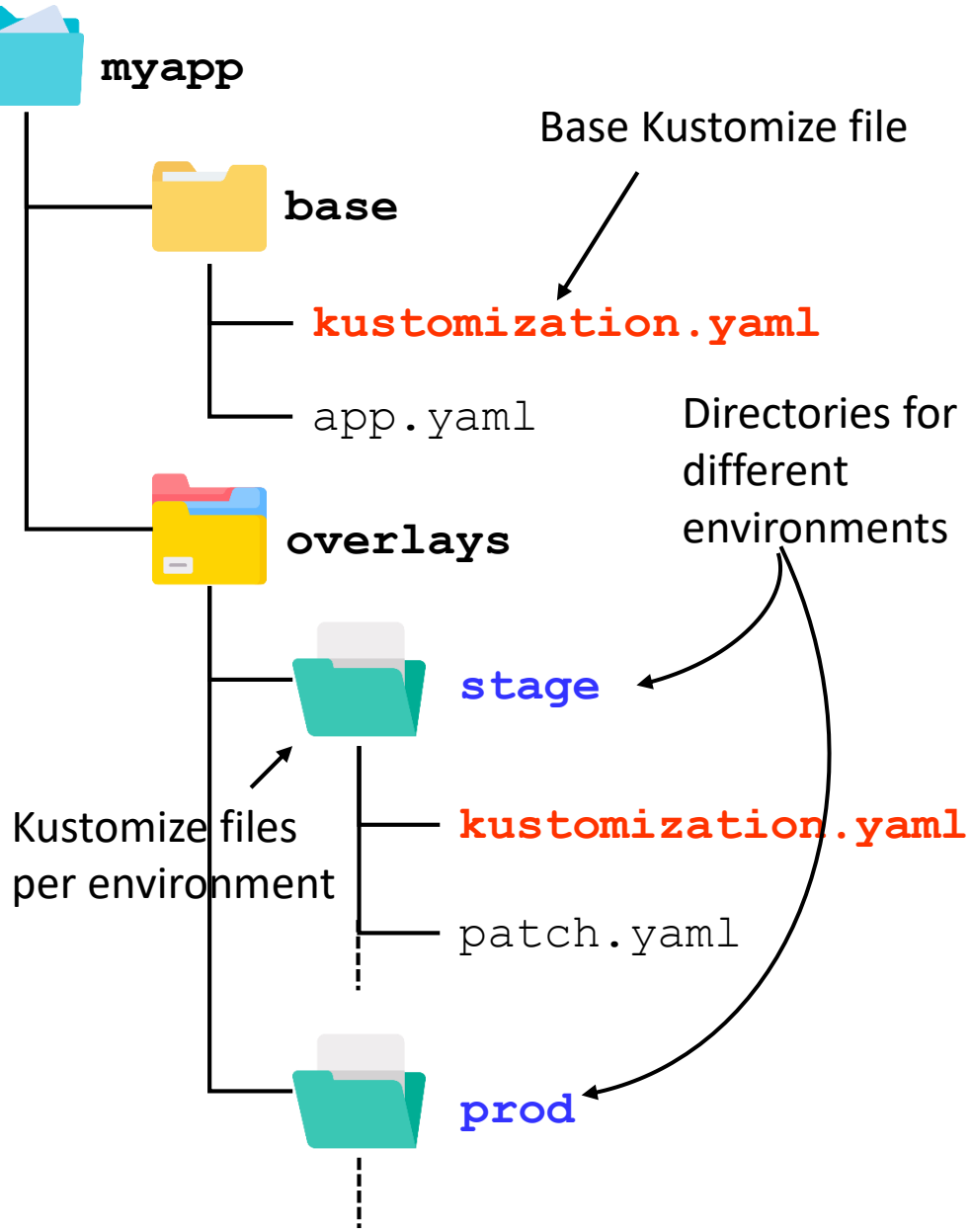
base/kustomization.yaml

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- app.yaml
```

---

overlays/stage/kustomization.yaml

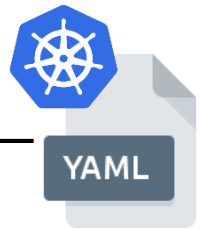
```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- ../../base

# site specific customization
namePrefix: ...
```



Skaffold

Kubernetes Application Setup

myapp



app.yaml

Deploy the image with one or more Kubernetes resources
Depends on application image

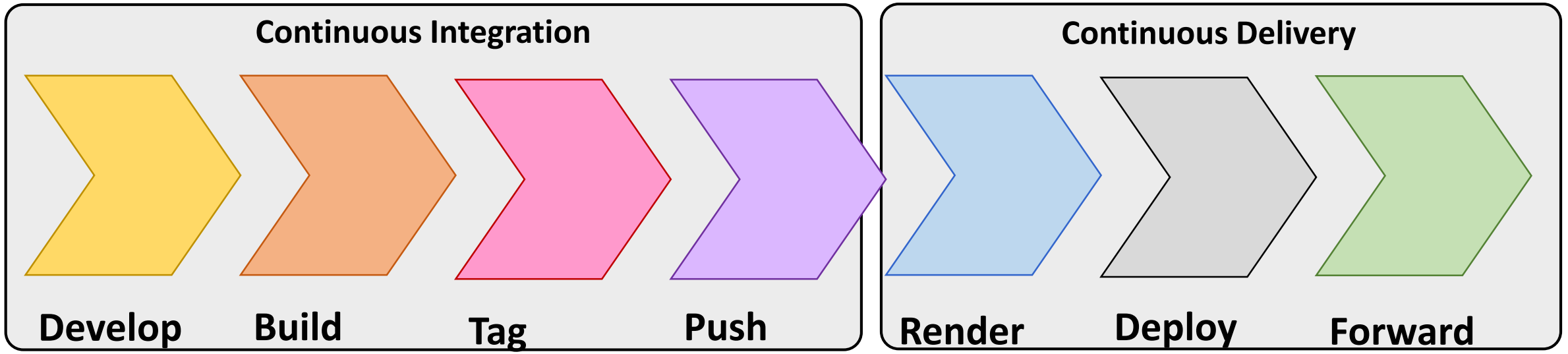
Dockerfile

Build the application image with the application files and artefacts.
May require compilation
Depends on sources

sources

Application files and artefacts

Developing, Debugging and Deploying



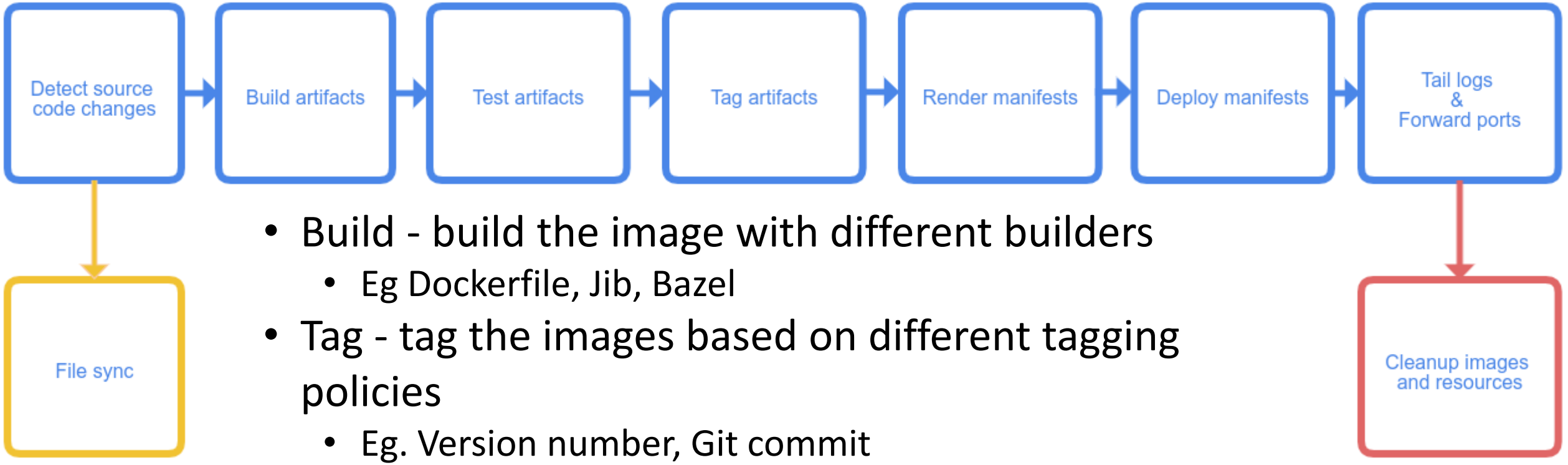
Repeat the process if there is any errors or bugs in any of the steps

Image adapted from https://www.youtube.com/watch?v=U7ybnPmwKik

Skaffold

- Skaffold is a tool to automate developing, testing and deploying applications to Kubernetes
- Watches for file changes in applications, Dockerfile and Kubernetes manifests
- Automate the development process
  - Build, Deploy, Access
  - Can use Skaffold to just execute one of the stage eg. Build
- skaffold CLI is used to automate the flow
  - Reads a default file called `scaffold.yaml`
  - Download from https://skaffold.dev/docs/install/

Skaffold Pipeline



- Build - build the image with different builders
  - Eg Dockerfile, Jib, Bazel
- Tag - tag the images based on different tagging policies
  - Eg. Version number, Git commit
- Test - run tests
- Deploy - deploy the application
  - With kubectl, Helm or Kustomize
- Tail logs and forward ports

Image from https://skaffold.dev/docs/pipeline-stages/

Skaffold Manifest

```
apiVersion: skaffold/v2beta29
kind: Config
metadata:
  name: myapp

build:

deploy:

test:

portForward:
```

Version from https://skaffold.dev/docs/references/yaml/

Config kind

Config name

Build specification

Deployment strategy

Run test

Ports to expose

Building Images

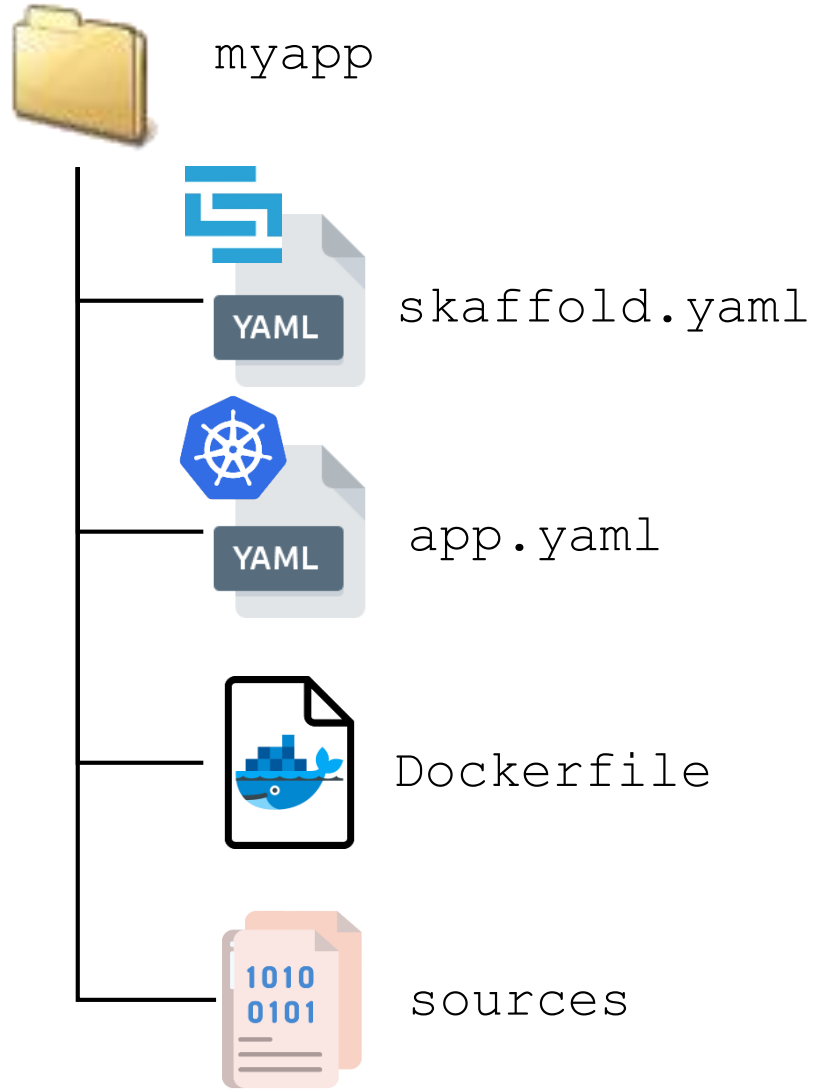
- Supports multiple tools for building images
  - Dockerfile
  - Jib (Java image builder) for Maven and Gradle
  - Buildpacks - see https://buildpacks.io
  - Custom script
- Docker build
  - Can build images locally or in-cluster
  - Images can be optionally pushed to registry
  - Tagging policy based on SHA, Git commit, version,

Example - build

```
build:
  artifacts:
  - image: myapp
    context: .
    docker:
      dockerfile: Dockerfile

  local:
    tryImportMissing: true

  tagPolicy:
    envTemplate:
      template: "{{ .APP_VERSION }}"
```

Import images that are not present

Tag the image with the environment variable



With skaffold

```
APP_VERSION="v1" skaffold build
```

With Docker

```
docker build -t fred/myapp:v1
docker push fred/myapp:v1
```

Deployment

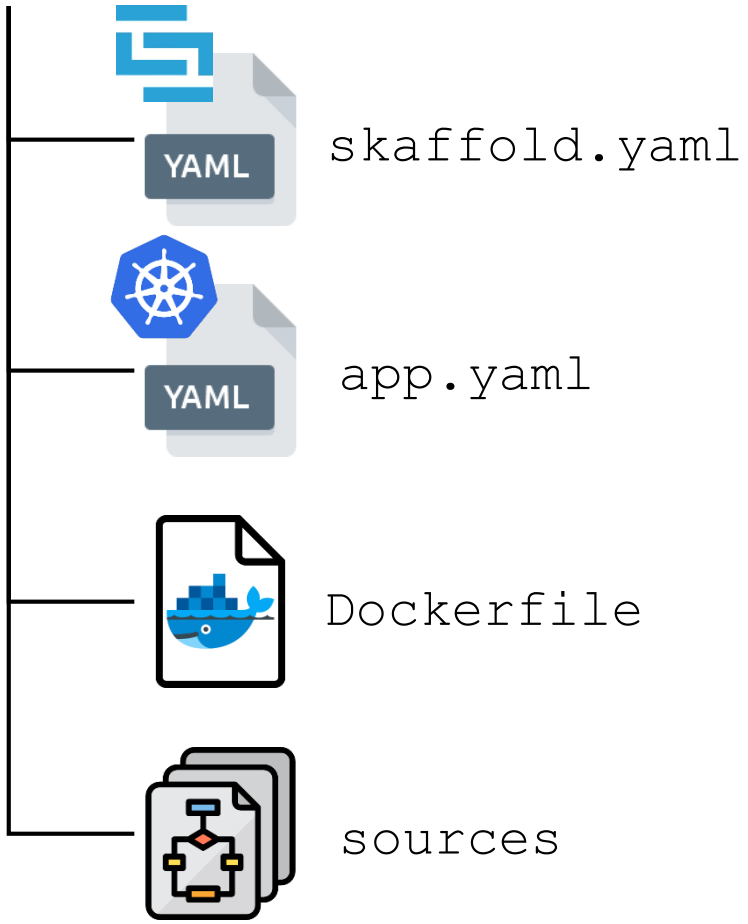
- Supports the following ways to deploy the application
  - kubectl
  - helm
  - kustomize
- CLI for all of the above must be installed on the machine that runs the skaffold command

Example - deploy with kubectl

myapp

One or more Kubernetes resource files

```
deploy:
  kubectl:
    manifests:
    - app.yaml
    defaultNamespace: myapp-ns
```



`manifests` supports patterns for. Example:

all the files with k8s prefix and .yaml suffix

```
k8s-*.yaml
```

all the files in the k8s directory

```
k8s/*.yaml
```

With skaffold

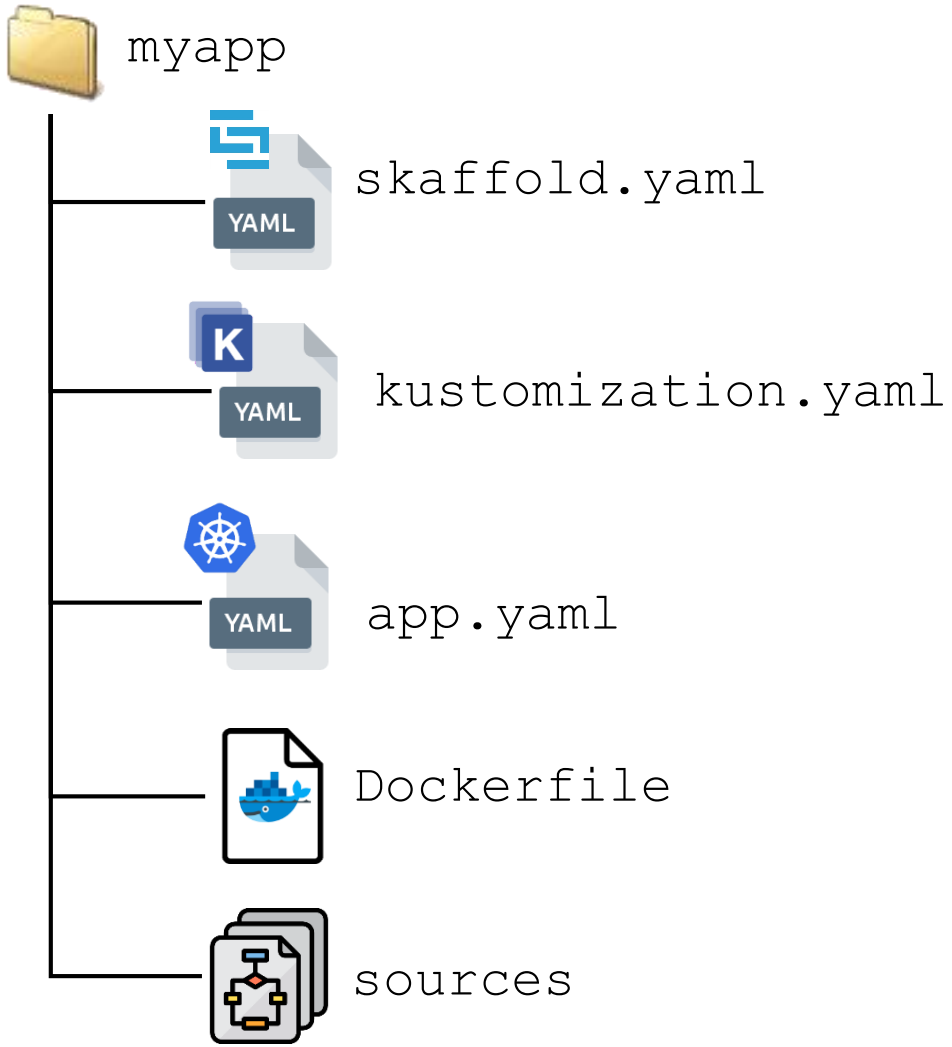
```
APP_VERSION="v1" skaffold run
```

With kubectl

```
kubectl apply -f app.yaml
```

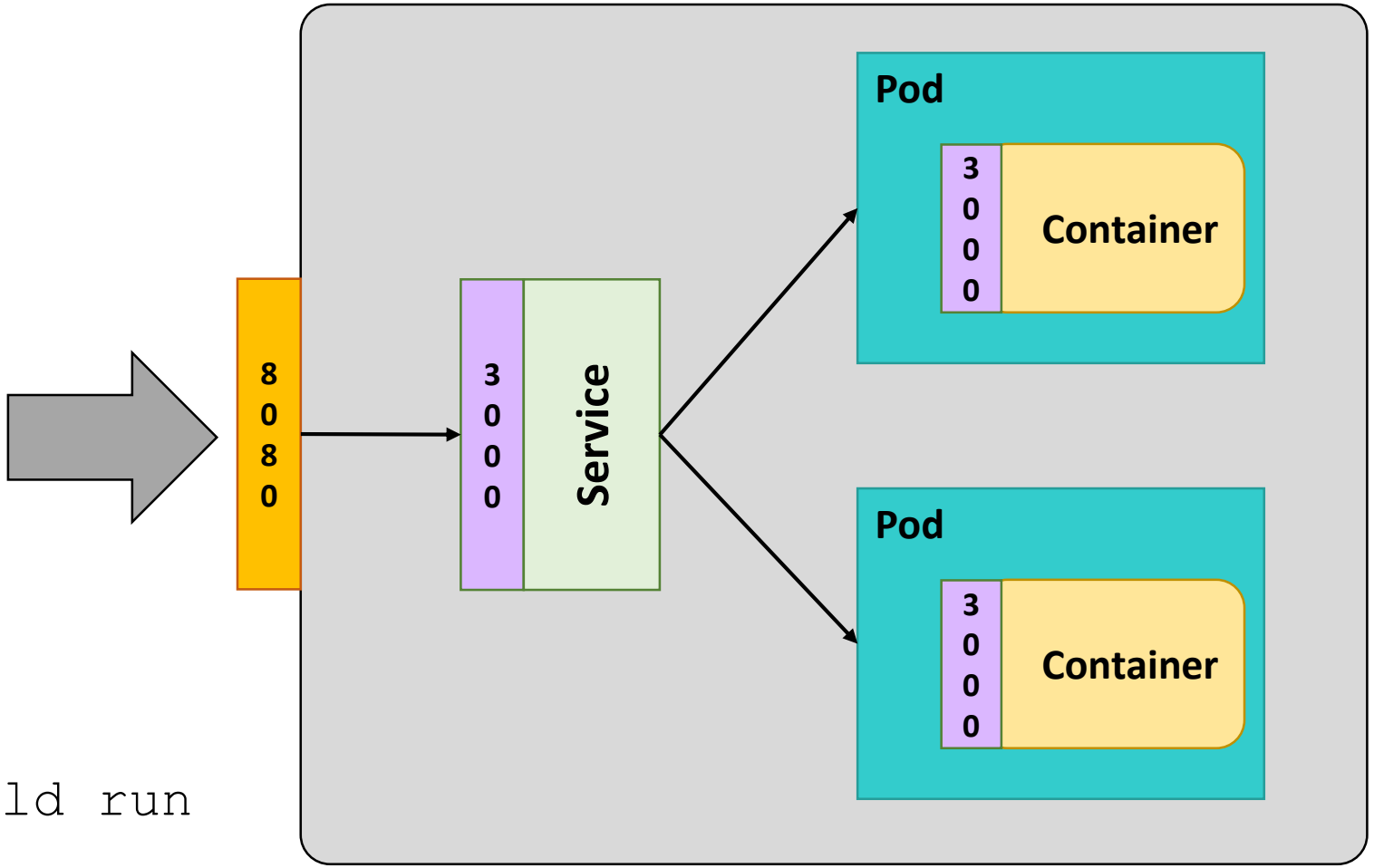
Example - deploy with Kustomize

```
deploy:
  kustomize:
    paths:
    - ./kustomization
    defaultNamespace: myapp-ns
```



Example - Port Forward

```
portForward:
- resourceType: Service
  resourceName: app-svc
  namespace: myapp-ns
  port: 3000
  localPort: 8080
```



With skaffold

```
APP_VERSION="v1" skaffold run
```

With kubectl

```
kubectl port-forward svc/myapp-svc 8080:3000 -n myapp-ns
```

Skaffold Pipeline

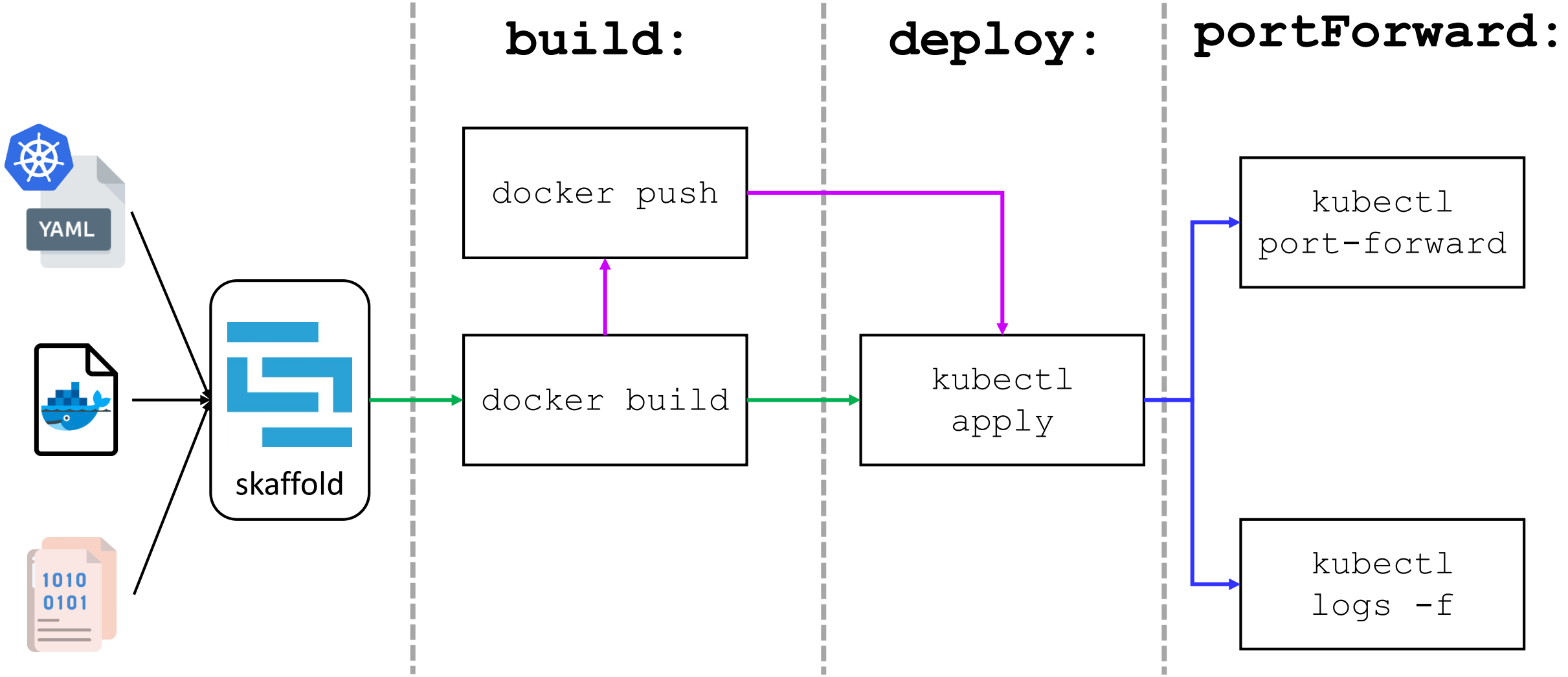
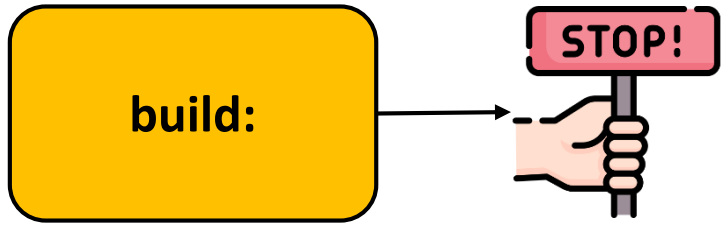


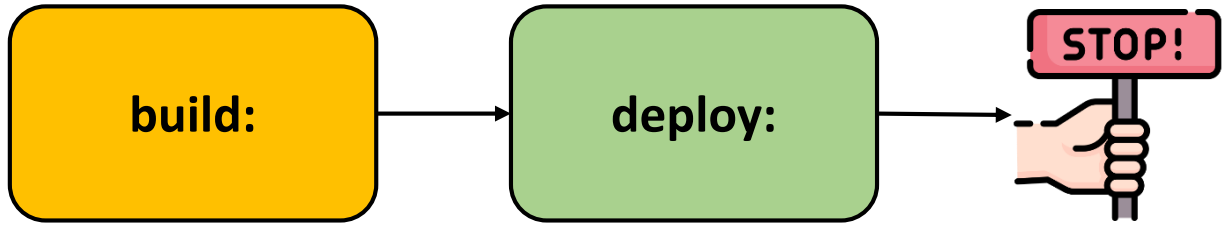
Image adapted from https://ahmet.im/blog/skaffold/

Skaffold Commands

`skaffold build`



`skaffold run`



`skaffold dev`

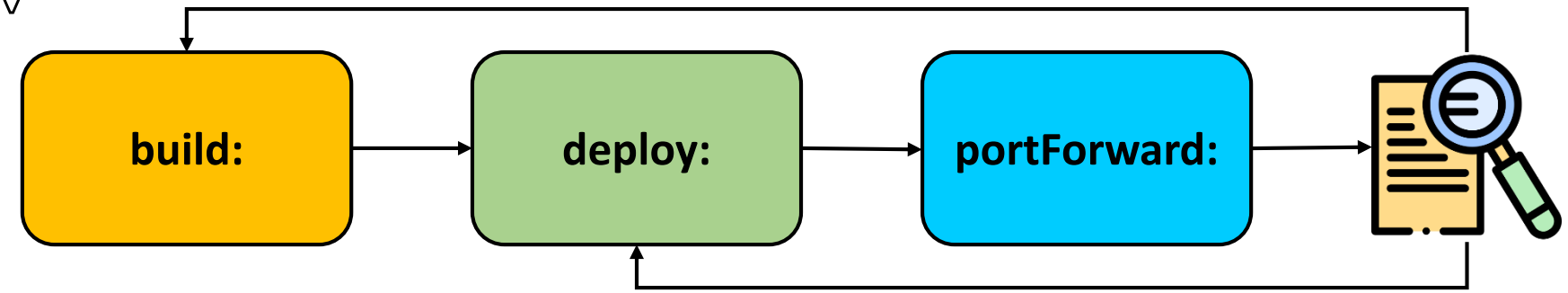
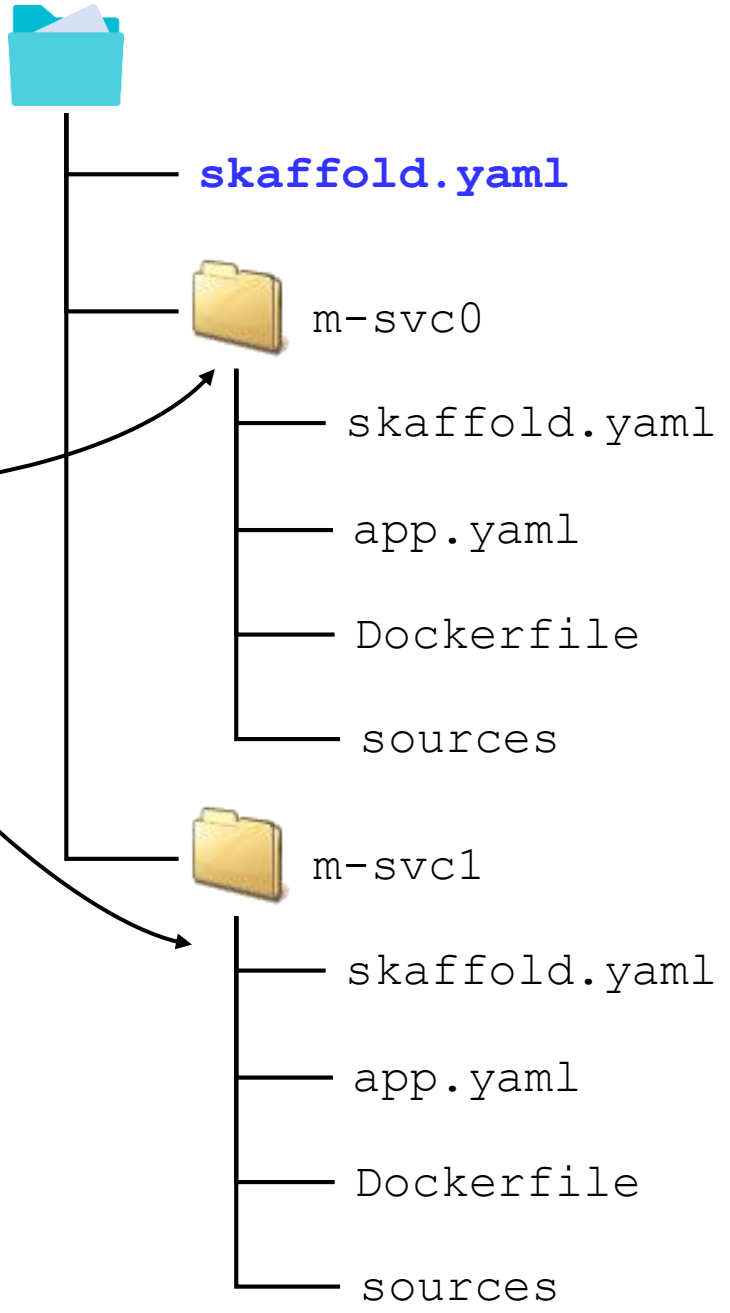


Image adapted from https://ahmet.im/blog/skaffold/

Working with Monorepos

Each service has its own `Dockerfile`, Kubernetes manifests and `skaffold.yaml`



skaffold.yaml

```
apiVersion: skaffold/v2beta29
kind: Config
metadata:
  name: m-services
requires:
- path: ./m-svc0
- path: ./m-svc1
```

Use the `skaffold.yaml` file in the respective directories