Nuvolaris Trainings

Developing Kubernetes Operators in Python

Part 1: Kubernetes Operators

https://www.nuvolaris.io

Agenda

- Kubernetes Introduction
- Development Environment
- Kubernetes Operators
- Creating an Operator
- Deploying an Operator
- Contributing to Nuvolaris

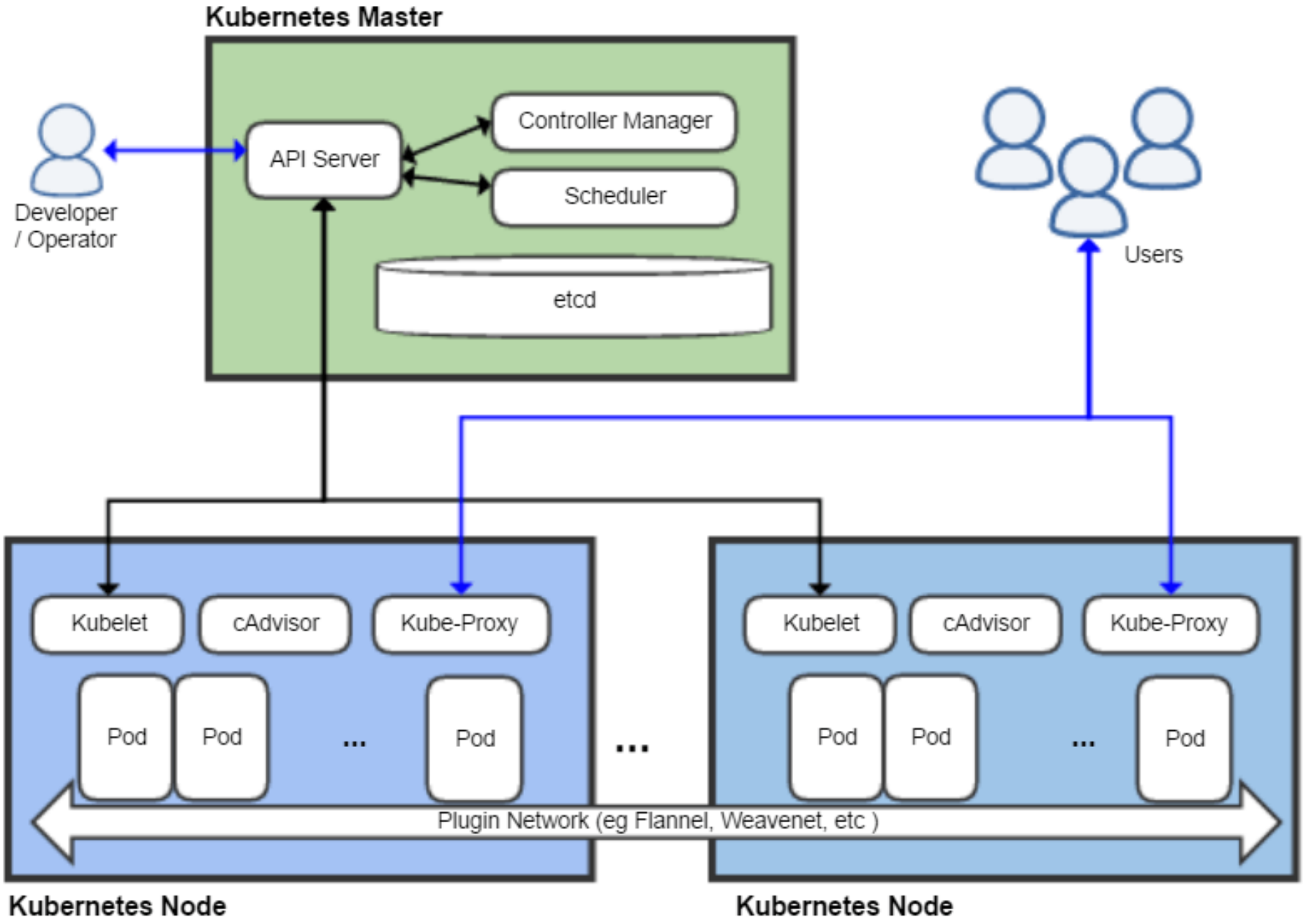
Kubernetes

What is Kubernetes ?

- In theory, an **orchestrator**
  - also Windows, originally, was just a **GUI** on top of DOS
- In practice, an **Operating System** for the cloud

What is Nuvolaris?

- a Serverless **distribution** for Kubernetes
- *Linux* : **RedHat** = *Kubernetes* : **Nuvolaris**

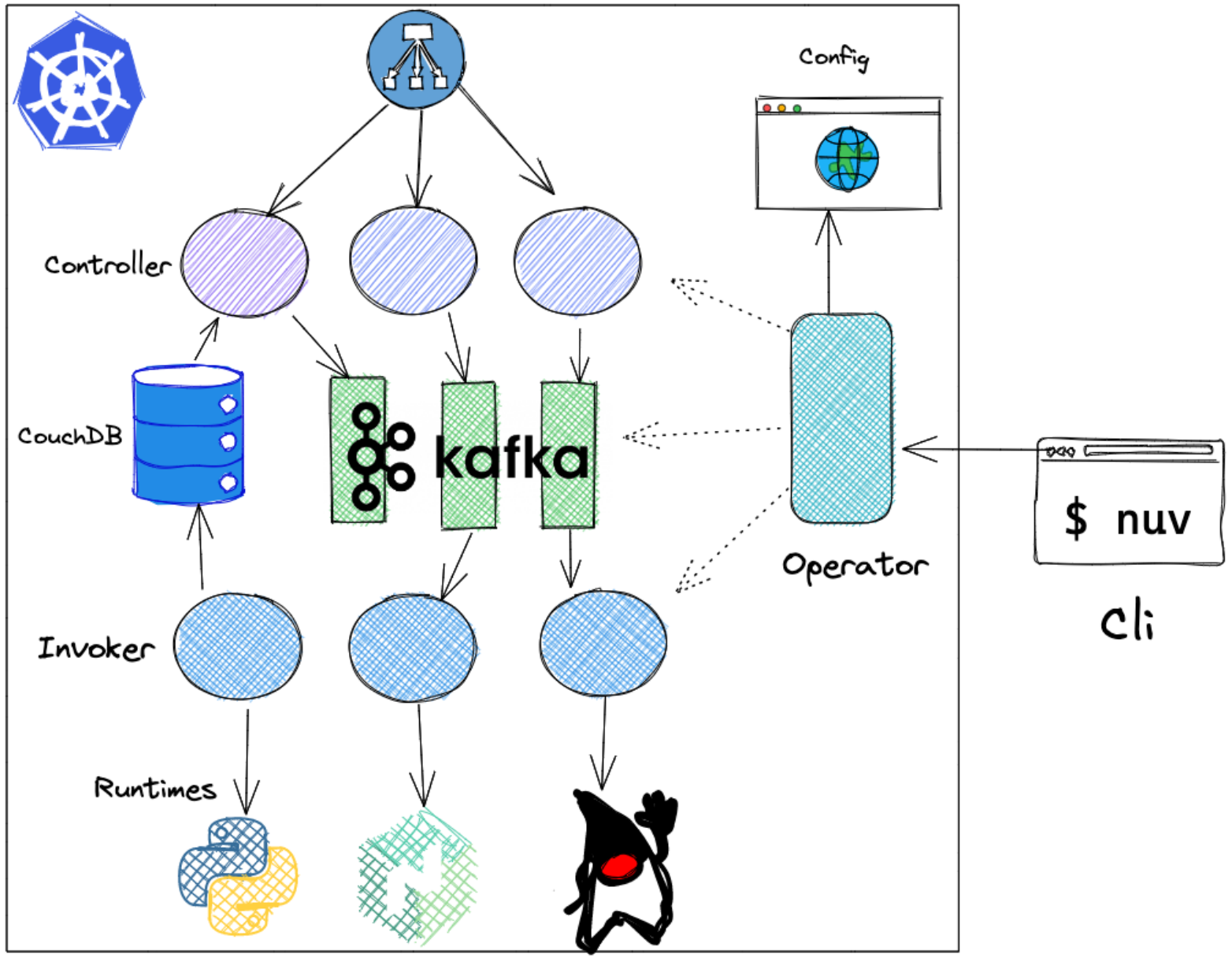

Kubernetes Master
API Server
Controller Manager
Scheduler
etcd
Developer
/ Operator
Users
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
...
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
Plugin Network (eg Flannel, Weavenet, etc)
Kubernetes Node
Kubernetes Node

Kubernetes Operators

- It is a **pattern** that is becoming commonplace
  - There is *NOT* a specific API that you implement
  - You have to use the *Kubernetes API* anyway
- You define your own Resource
  - Defining new resources as **CRD** `Custom Resource Definitions`
  - Creating instances conforming to the CRD
    - that describes the *desidered state*
  - **Writing code that brings the system to this state**

Kubernetes

```
kubectl get nodes
kubectl get ns
kubectl create ns demo
kubectl get ns
```



Nuvolaris Architecture

Operator Frameworks

- **Operator Framework**: ansible/helm/go
- **Kudo**: a declarative, yaml based framework
- **metacontroller**: generic, with hooks in any languages
- **shell-operator**: write operators in bash
- **kubebuilder**: Go based operator
- **kopf**: Python based Operator

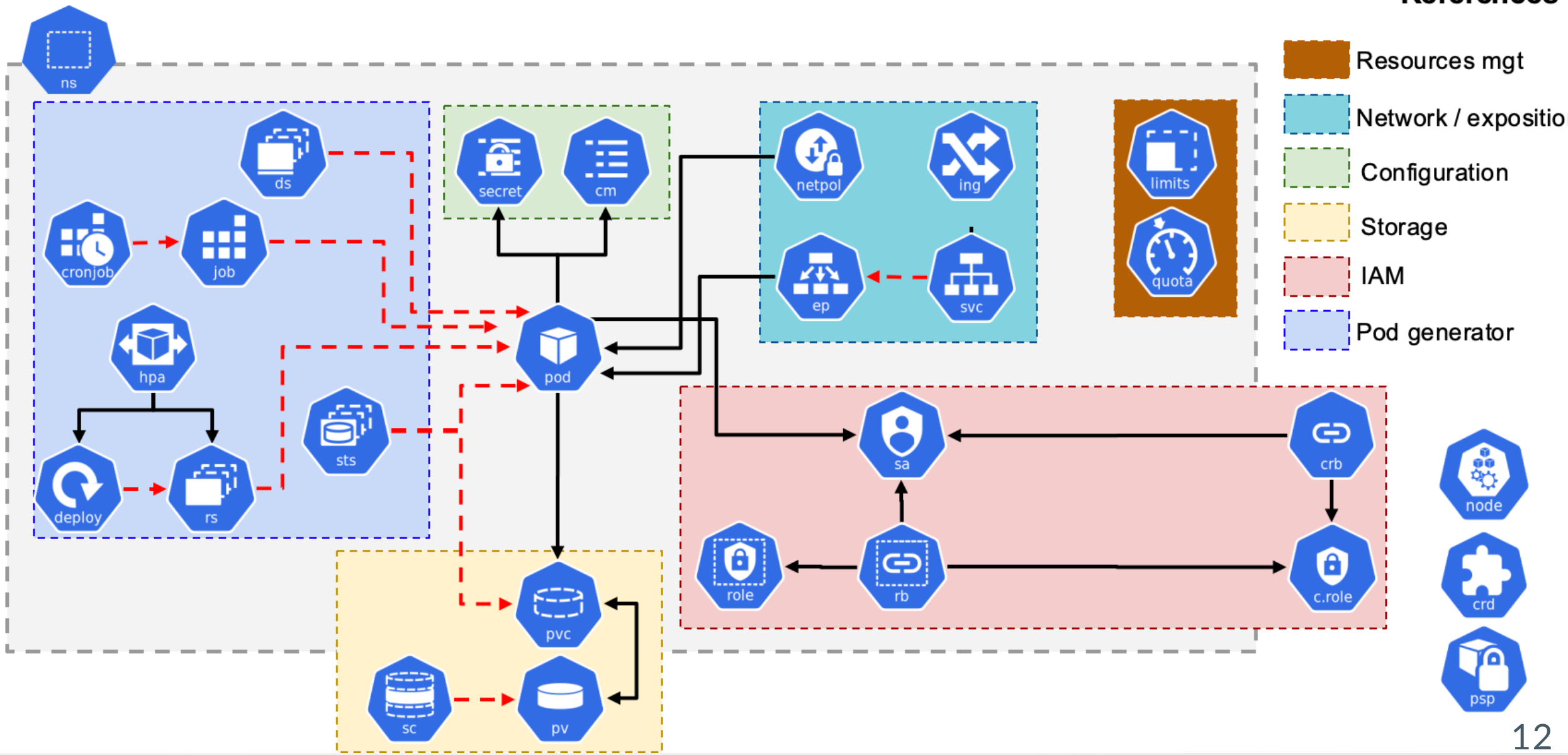
also exists Java, Rust, Elixir, Javascript based operator frameworks

Kubernetes Descriptors

Kubernetes Descriptors Concepts

- It is declarative:
  - You **describe** what you want to get by the system
  - Kubernetes will bring the system to the desidered state
- You declare what you want with *descriptors* in YAML
  - those descriptors are in YAML format
    - actually, they are internally JSON files
    - YAML is really syntax sugar for JSON
- Kubernetes brings the system to what you asked
  - ... **if it is possible** ...

Kubernetes Ressources Map



Structure of a descriptor

- Common: Header and Metadata

```
apiVersion: v1
kind: Namespace
metadata:
  name: nuvolaris
```

- `spec` : changes according tothe kind
- `status` : maintained by the system

Simple Descriptor: a Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: demo-pod
  namespace: demo
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

Nested Descriptor: a Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deploy
```

Templatized, repeat the template using labels

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

Deployment template

It creates `replica` times the pods specified in the template

```
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

Dev Environment

VSCode-based Development Environment

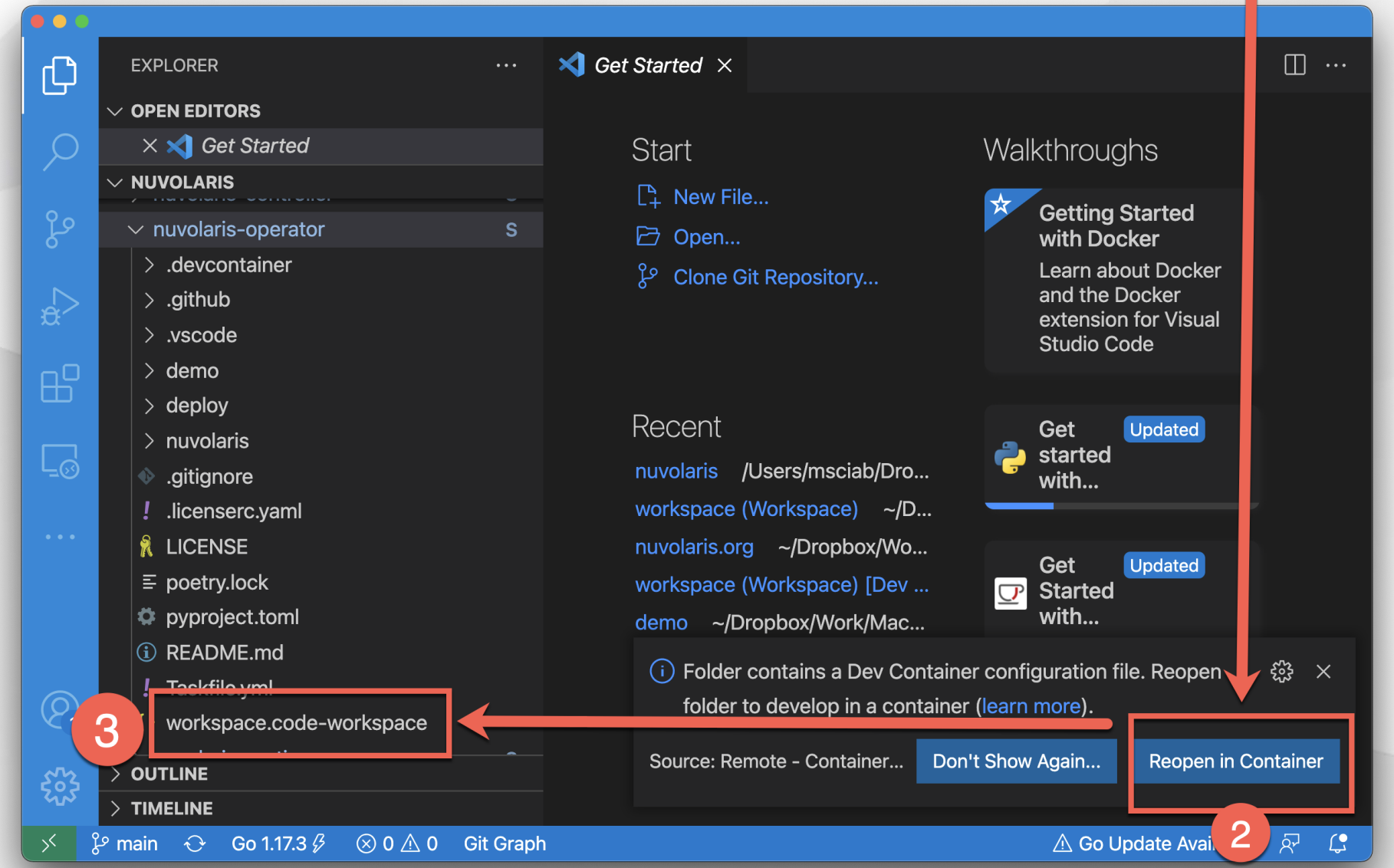
- Clone the repositories (multiple and linked)

```
git clone https://github.com/nuvolaris/nuvolaris
--recurse-submodules
```

  - do not forget **`--recurse-submodules`**
- Open the folder `nuvolaris` with VSCode:
  - Command Line: `code nuvolaris`
- Open the workspaces in subfolders: `workspace.code-workspace`

1

```
$ git clone https://github.com/nuvolaris/nuvolaris --recurse-submodules
$ code nuvolaris
```



Kubernetes `kubectl`

- $ `kubectl get nodes`

```
NAME                      STATUS   ROLES                  AGE   VERSION
nuvolaris-control-plane   Ready    control-plane,master   41m   v1.21.1
nuvolaris-worker          Ready    <none>                 41m   v1.21.1
```

Demo: `nuvolaris-controller/training/transcript1.txt`

Kubernetes CRD

Kubernetes Controllers

- Deployment, DaemonSet, StatefulSet

```
[~]$ kubectl get deploy
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           21m
[~]$ kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-66b6c48dd5-4dpl2   1/1     Running   1          21m
nginx-deployment-66b6c48dd5-5c4q6   1/1     Running   1          21m
nginx-deployment-66b6c48dd5-xs8nd   1/1     Running   1          21m
```

What they do?

- create a set of resources
- control them as an unit

Custom Resources Definitions

- Define your own Kubernetes Resources
  - create new Kinds of resources
  - Handled as other resources

Resource Handlers

- You need to write your own resource handler!
  - It responds to Kubernetes events
  - It interacts with Kubernetes APIs to perform operations

Components of a CRD

- Group, Kind and short names:
  - Example: `nuvolaris.org`, `Sample`, `sam`
- Spec and Status
  - Versioned
  - defined as an OpenApi Schema:

```
type: object
properties:
  spec:
    type: object
```

Defining a CRD (1/2)

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: samples.nuvolaris.org
spec:
  scope: Namespaced
  group: nuvolaris.org
  names:
    kind: Sample
    plural: samples
    singular: sample
    shortNames:
      - sam
```

Defining a CRD (2/2)

```
versions:
  - name: v1
    served: true
    storage: true
    subresources: { status: { } }
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            x-kubernetes-preserve-unknown-fields: true
          status:
            type: object
            x-kubernetes-preserve-unknown-fields: true
```

Instance

```yaml
apiVersion: nuvolaris.org/v1
kind: Sample
metadata:
  name: obj
spec:
  count: 2
```

- Demo: `nuvolaris/nuvolaris-controller/training/transcript2.txt`

Kubernetes Operator

About `kopf`

- See kopf.readthedocs.io
- Python based
  - provied an handy `kopf` cli runner
- Handlers for the various Kubernetes events:
  - `@kopf.on.login`
  - `@kopf.on.create`
  - `@kopf.on.delete`
- It does not manage Kubernetes API

Login

- Kopf supports various autentication
  - Code to support either your `~/.kube/config` or the service token

```
@kopf.on.login()
def sample_login(**kwargs):
    token = '/var/run/secrets/kubernetes.io/serviceaccount/token'
    if os.path.isfile(token):
        logging.debug("found serviceaccount token: login via pykube in kubernetes")
        return kopf.login_via_pykube(**kwargs)
    logging.debug("login via client")
    return kopf.login_via_client(**kwargs)
```

Handling object creation and deletion

```
@kopf.on.create('nuvolaris.org', 'v1', 'samples')
def sample_create(spec, **kwargs):
    print(spec)
    return { "message": "created" }
```

```
@kopf.on.delete('nuvolaris.org', 'v1', 'samples')
def sample_delete(spec, **kwargs):
    print(spec)
    return { "message": "delete" }
```

Kustomize

Interacting with Kubernetes

- `kopf` does *not* provide how to interact with Kubernetes
  - You can use any other api like `pykube` or others
- We use... `kubectl` and `kustomize`
  - It may look "odd" to use an external command line tool
  - However, this allows compatibility with command line tools
    - avoding "strange" templating
    - easier development and debug

About `kustomize`

- Originally a separate tool, now part of `kubectl`
  - It works "customizing" sets of descriptors with rules
  - support many ways of *patching* the JSON/YAML
  - **NO TEMPLATING** (huge win over `helm`!)
- You simply do `kubectl apply -k <folder>`
  - It will search for `kustomization.yaml`
  - It will produce the output sent to Kubernetes
- Debug the output without applying with:
  `kubectl kustomize <folder>`

Simple `kustomizationl.yaml` with patch

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- demo-deployment.yaml
patches:
 - path: patch.yaml
```

- put it in a folder `deploy` and `apply -k deploy`

Sample patch of a `Deployment`

- We want to change the replica count

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deploy
spec:
  replicas: 2
```

- Intuitively, provide enough context to locate the descriptor
- Provide the replaced fields

Implementing Operator

Using `kubectl` from the operator

```
# generate patch
def patch(n):
  return f"""apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-deploy
spec:
  replicas: {n}
"""
# run kubectl
def kubectl(cmd, patch):
  with open(f"deploy/patch.yaml", "w") as f:
    f.write(patch)
  res = subprocess.run(["kubectl", cmd, "-k", "deploy"], capture_output=True)
  return res.stdout.decode()
```

Implementing the operator

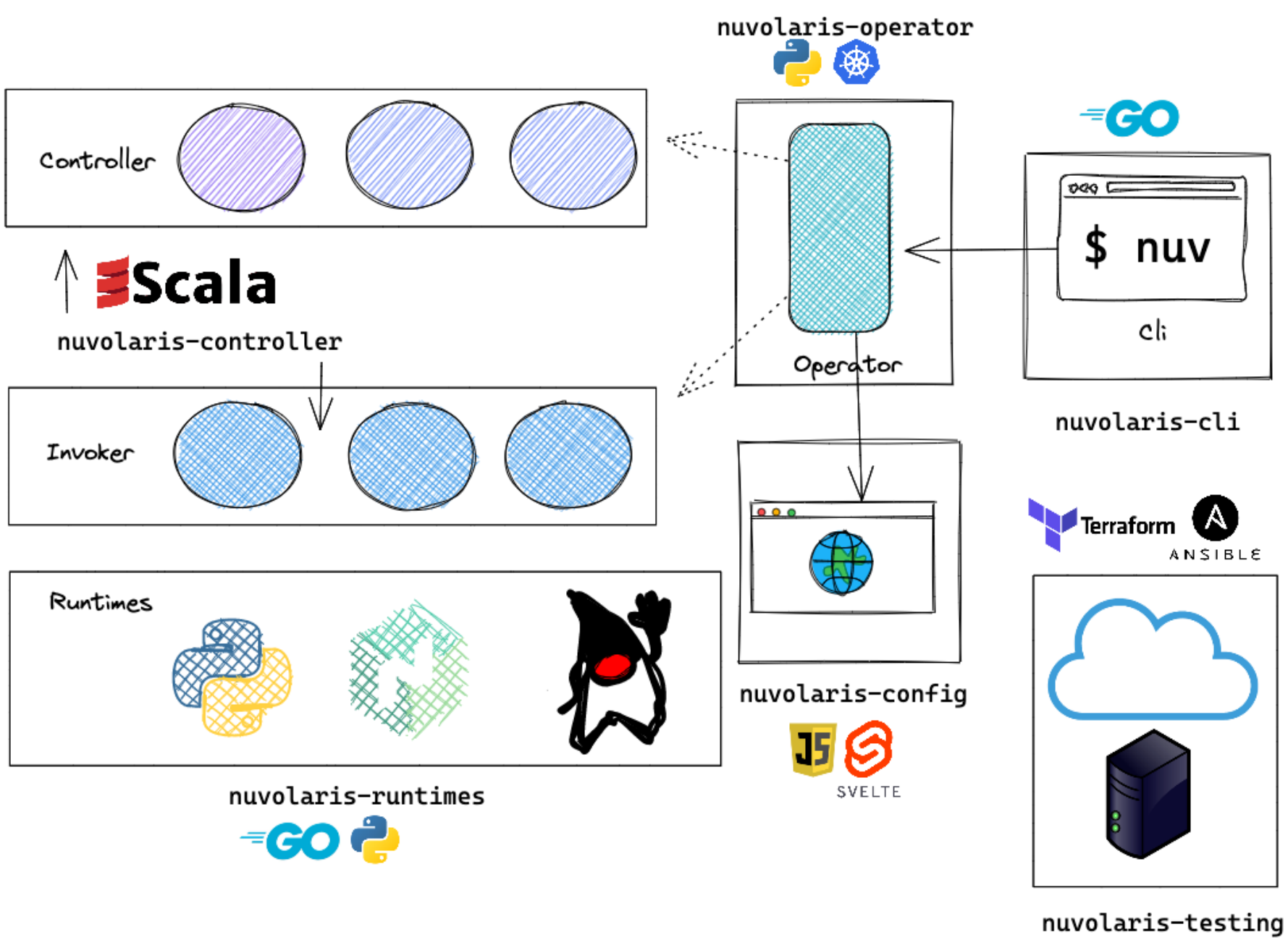
```
@kopf.on.create('nuvolaris.org', 'v1', 'samples')
def sample_create(spec, **kwargs):
    count = spec["count"]
    message = kubectl("apply", patch(count))
    return { "message": message }

@kopf.on.delete('nuvolaris.org', 'v1', 'samples')
def sample_delete(spec, **kwargs):
    count = spec["count"]
    message = kubectl("delete", patch(count))
    return { "message": "delete" }
```

Packaging

- Create a Dockerfile embedding the operator
  - You need `poetry`, `kopf` an `kubectl` in the image
- Deploy the POD with the right permissions
  - You need to setup *Kuberbetes RBAC*
  - `ServiceAccount` and `ClusterRoleBinding`
- See `nuvolaris/nuvolaris-operator` for an example

Contributing



Nuvolaris Components and Technologies

Required OpenSource Paperwork

Before sending a PR

- Add **Apache License** headers to each file:
  quick way: `license-eye header fix`
- Download the ICLA and sign it:
  `bit.ly/apache-icla`
- Send to
  `To: secretary@apache.org`
  `Cc: secretary@nuvolaris.io`