<devAcademy>

Introducción práctica a Kubernetes

Juan Vicente Herrera Ruiz de Alejo

DevAcademy es una comunidad compuesta por **profesionales entusiastas** de innovadoras tecnologías en áreas como Big Data, Data Science, DevOps, Ciberseguridad o Fintech.

Este proyecto busca crear “píldoras formativas” que imparten profesionales a través de diferentes rutas formativas con el fin de que los profesionales puedan aprender de forma práctica y en formato ágil.

Los expertos de DevAcademy saben qué se necesita en el mundo laboral y componen sus programas formativos con el fin de disminuir la curva de aprendizaje en base a su experiencia, dando los mejores consejos e indicaciones profesionales.

</ Sobre <devAcademy>

La comunidad de entusiastas de DevAcademy está formada por **más de 40 profesionales** a lo largo de 4 áreas: Big Data, Data Science, DevOps y Fintech.

Formación abierta y formación in-Company. Algunos de los que confían en nosotros:

Más de 50 cursos innovadores en tecnologías específicas: Hadoop, Spark, Blockchain, Jenkins, Puppet, Mongodb, Python, etc.

Más de 350 profesionales formados en 2018 por diferentes especialistas en tecnologías innovadoras

Santander

Devacademy ha sido seleccionada como proyecto innovador en el programa de emprendedores Yuzz patrocinado por el Banco Santander.

</>

Add profile section | More...

Juan Vicente Herrera Ruiz de Alejo

Platform and automation engineer (DevOps culture) at X by Orange

X by Orange

Universidad Pontificia Co...

https://iuta.education/programas/master-en-industria-4-0/
http://madrid.devops.es/
https://www.linkedin.com/in/jvherrera/
http://www.juanvicenteherrera.es
twitter.com/jvicenteherrera
http://www.slideshare.net/juanvicenteherrera

<devAcademy>

</>

Agenda

1)Introducción a la orquestación de contenedores con Kubernetes

2)Descripción de la arquitectura de Kubernetes

3)Pods

4)Etiquetas

5)Controladores

6)Servicios

7)API

8)Instalación de un clúster de Kubernetes (microk8s, minikube)

9)Creación de pods, volúmenes y despliegues de Kubernetes

10)Agrupando y organización de un clúster

11)Administración de servicios

12)Conexión de contenedores

13)Seguridad de Kubernetes

14)Monitorización en Kubernetes

15)Escalabilidad de cluster en Kubernetes

16)Infraestructura de Kubernetes

17)Aprovisionamiento

18)Particionamiento

19)Redes

20)Despliegue de un cluster de alta disponibilidad

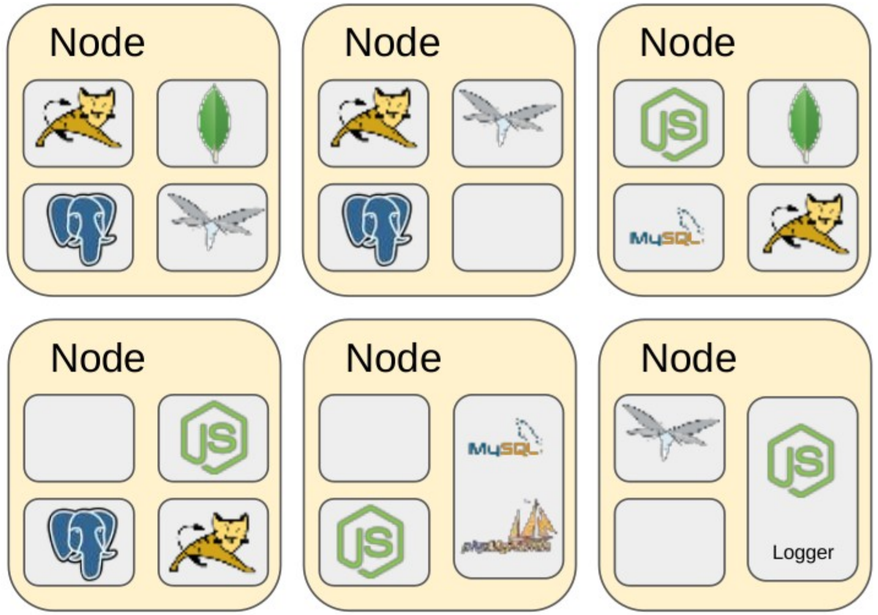
21)Hands on (local y GCP)

<devAcademy>

</>

DevOps challenges for multiple containers

- How to scale?
- How to avoid port conflicts?
- How to manage them in multiple hosts?
- What happens if a host has a trouble?
- How to keep them running?
- How to update them?
- Where are my containers?



<devAcademy>

</>

Introduction to Kubernetes

- What is Kubernetes?
  - Open source platform
  - Automation of application containers
    - Deployment
    - Scaling
    - Operating
  - Started by Google
    - 2014
    - Cloud Native Computing Foundation
  - Abbreviated as K8s
    - **K**ubernete**s** = K8s
    - "Pilot"
    - "Governor"

</>

Meet Kubernetes

Greek for *“Helmsman”*; also the root of the word *“Governor”* *(from latin: gubernator)*

- Container **orchestrator**
- Supports **multiple cloud** and **bare-metal** environments
- Inspired by Google’s experience with containers
- **Open source**, written in **Go**

Manage **applications**, not machines

Introduction to Kubernetes

- Deploy applications using containers
  - Virtualize operating system not hardware
    - More efficient use of resources
    - Small and fast
    - One app per container
- Kubernetes benefits
  - Portable
  - Extensible
  - Self-healing
- Quickly and efficiently respond to demand
  - Deploy applications quickly
  - Roll out new features easily
  - Scale easily
  - Use only required resources

</>

Why Kubernetes?

- Goal
  - Running apps in the Cloud simpler
- Container-centric infrastructure
  - Away from host-centric infrastructure
- Result
  - Take full advantage of benefits fundamental to containers
  - Deployments
    - Simpler
    - Portable
    - Predictable
  - Easier management

Kubernetes Supported Platforms

- Runs on top of Linux
- Platform agnostic
  - Bare metal
    - Servers
    - Desktops
    - Laptops
  - Virtual machines (VMs)
  - Cloud provider
  - OpenStack

</>

Declarative vs imperative

This container orchestrator puts a very strong emphasis on being declarative

Declarative:

I would like a cup of tea.

Imperative:

Boil some water. Pour it in a teapot. Add tea leaves. Steep for a while. Serve in cup.

Declarative seems simpler at first …

… As long as you know how to brew tea

</>

Declarative vs imperative

What declarative would really be:

I want a cup of tea, obtained by pouring an infusion of tea leaves in a cup.

An infusion is obtained by letting the object steep a few minutes in hot water.

Hot liquid is obtained by pouring it in an appropriate container and setting it on a stove.

Ah, finally, containers! Something we know about. Let’s get to work, shall we?

</>

Summary of declarative vs imperative

- Imperative systems:
  - Simpler
  - if a task is interrupted, we have to restart from scratch
- Declarative systems:
  - if a task is interrupted (or if we show up to the party half-way through), we can figure out what's missing and do only what's necessary
  - we need to be able to observe the system
  - … and compute a "diff" between what we have and what we want

</>

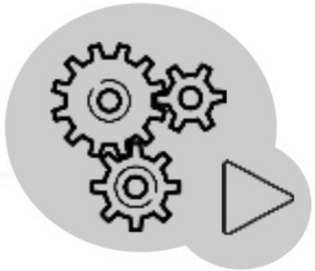
Declarative vs imperative in Kubernetes

- Virtually everything we create in Kubernetes is created from a spec
- Watch for the spec fields in the YAML files later!
- The spec describes how we want the thing to be
- Kubernetes will reconcile the current state with the spec (technically, this is done by a number of controllers)
- When we want to change some resource, we update the spec
- Kubernetes will then converge that resource

</>

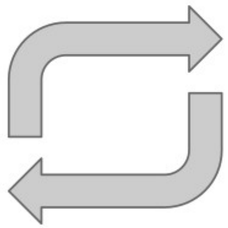
Kubernetes Concepts

Pod



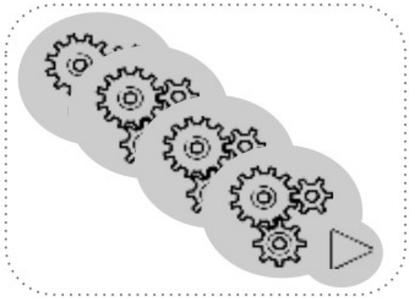
One or More Containers
Shared IP
Shared Storage Volume
Shared Resources
Shared Lifecycle

Replication Controller / Deployment



Ensures that a specified number of pod replicas are running at any one time

Service



Grouping of pods, act as one, has stable virtual IP and DNS name

Label

Key/Value pairs associated with Kubernetes objects (e.g. env=production)

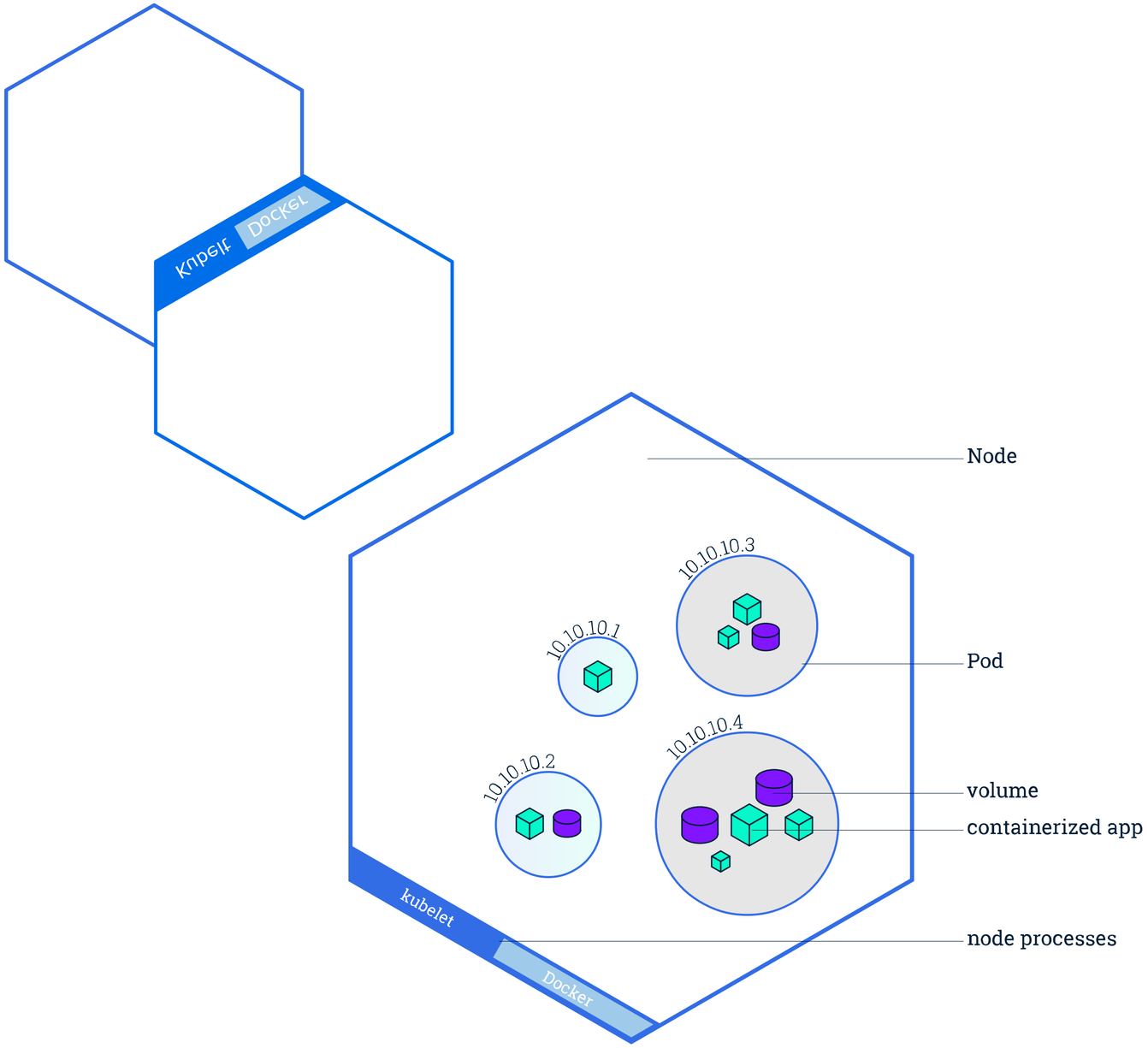
Nodes

- A Pod always runs on a Node.
- A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster.
- Each Node is managed by the Master.
- A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster.
- The Master's automatic scheduling takes into account the available resources on each Node.

Nodes

Every Kubernetes Node runs at least:

- Kubelet, a process responsible for communication between the Kubernetes Master and the Node; it manages the Pods and the containers running on a machine.
- A container runtime (like Docker, rkt) responsible for pulling the container image from a registry, unpacking the container, and running the application.
- Containers should only be scheduled together in a single Pod if they are tightly coupled and need to share resources such as disk.


kubelet
Docker
Node
10.10.10.3
10.10.10.1
Pod
10.10.10.4
10.10.10.2
volume
containerized app
kubelet
Docker
node processes

</>

Nodes

A node’s status contains the following information:

- Addresses
- Condition
- Capacity
- Info

</>

Addresses

The usage of these fields varies depending on your cloud provider or bare metal configuration.

- HostName: The hostname as reported by the node's kernel. Can be overridden via the kubelet --hostname-override parameter.
- ExternalIP: Typically the IP address of the node that is externally routable (available from outside the cluster).
- InternalIP: Typically the IP address of the node that is routable only within the cluster.

</>

Condition

The conditions field describes the status of all Running nodes.

Node Condition Description

OutOfDisk: True if there is insufficient free space on the node for adding new pods, otherwise False

Ready: True if the node is healthy and ready to accept pods, False if the node is not healthy and is not accepting pods, and Unknown if the node controller has not heard from the node in the last node-monitor-grace-period (default is 40 seconds)

MemoryPressure: True if pressure exists on the node memory – that is, if the node memory is low; otherwise False

PIDPressure: True if pressure exists on the processes – that is, if there are too many processes on the node; otherwise False

DiskPressure: True if pressure exists on the disk size – that is, if the disk capacity is low; otherwise False

NetworkUnavailable: True if the network for the node is not correctly configured, otherwise False

The node condition is represented as a JSON object. For example, the following response describes a healthy node.

</>

Nodes

The node condition is represented as a JSON object. For example, the following response describes a healthy node.

```
"conditions": [
  {
    "type": "Ready",
    "status": "True"
  }
]
```

</>

Nodes

- If the Status of the Ready condition remains Unknown or False for longer than the pod-eviction-timeout, an argument is passed to the kube-controller-manager and all the Pods on the node are scheduled for deletion by the Node Controller.

- The default eviction timeout duration is five minutes. In some cases when the node is unreachable, the apiserver is unable to communicate with the kubelet on the node.

- The decision to delete the pods cannot be communicated to the kubelet until communication with the apiserver is re-established. In the meantime, the pods that are scheduled for deletion may continue to run on the partitioned node.

</>

Nodes

In versions of Kubernetes prior to 1.5, the node controller would force delete these unreachable pods from the apiserver.

- However, in 1.5 and higher, the node controller does not force delete pods until it is confirmed that they have stopped running in the cluster.

- You can see the pods that might be running on an unreachable node as being in the Terminating or Unknown state.

- In cases where Kubernetes cannot deduce from the underlying infrastructure if a node has permanently left a cluster, the cluster administrator may need to delete the node object by hand.

- Deleting the node object from Kubernetes causes all the Pod objects running on the node to be deleted from the apiserver, and frees up their names.

</>

Nodes

Capacity

Describes the resources available on the node: CPU, memory and the maximum number of pods that can be scheduled onto the node.

Info

General information about the node, such as kernel version, Kubernetes version (kubelet and kube-proxy version), Docker version (if used), OS name. The information is gathered by Kubelet from the node.

</>

Installing Kubernetes

Local machine solutions

- minikube (https://kubernetes.io/docs/setup/minikube/)
- microk8s (Ubuntu) (https://microk8s.io/)
- kubeadm-dind
- Ubuntu on lxd

Cloud solutions

KOPS (https://github.com/kubernetes/kops)

Hosted solutions

- AWS EKS (https://aws.amazon.com/eks/)
- GCP kubernetes (https://cloud.google.com/kubernetes-engine/)

<devAcademy>

</>

First contact with kubectl

- kubectl is (almost) the only tool we'll need to talk to Kubernetes
- It is a rich CLI tool around the Kubernetes API (Everything you can do with kubectl, you can do directly with the API)
- You can also use the --kubeconfig flag to pass a config file
- Or directly --server, --user, etc.
- kubectl can be pronounced "Cube C T L", "Cube cuttle", "Cube cuddle"…

</>

kubectl get

- Let’s look at our Node resources with kubectl get!

- Look at the composition of our cluster:

kubectl get node

- These commands are equivalent

kubectl get no

kubectl get node

kubectl get nodes

</>

Obtaining machine-readable output

kubectl get can output JSON, YAML, or be directly formatted

Give us more info about the nodes:
kubectl get nodes -o wide

Let’s have some YAML:
kubectl get no -o yaml

</>

(Ab)using kubectl and jq

It’s super easy to build custom reports

Show the capacity of all our nodes as a stream of JSON objects:

```
kubectl get nodes -o json |
    jq ".items[] | {name:.metadata.name} + .status.capacity"
```

</>

Kubectl

What’s available?

kubectl has pretty good introspection facilities

We can list all available resource types by running kubectl get

We can view details about a resource with:

kubectl describe type/name

We can view the definition for a resource type with:

kubectl explain type

Each time, type can be singular, plural, or abbreviated type name.

<devAcademy>

</>

Running our first containers on Kubernetes

- First things first: we cannot run a container
- We are going to run a pod, and in that pod there will be a single container
- In that container in the pod, we are going to run a simple ping command
- Then we are going to start additional copies of the pod

</>

Node Controller

The node controller is a Kubernetes master component which manages various aspects of nodes.

The node controller has multiple roles in a node's life.

- The first is assigning a CIDR block to the node when it is registered (if CIDR assignment is turned on).

- The second is keeping the node controller's internal list of nodes up to date with the cloud provider's list of available machines. When running in a cloud environment, whenever a node is unhealthy, the node controller asks the cloud provider if the VM for that node is still available. If not, the node controller deletes the node from its list of nodes.

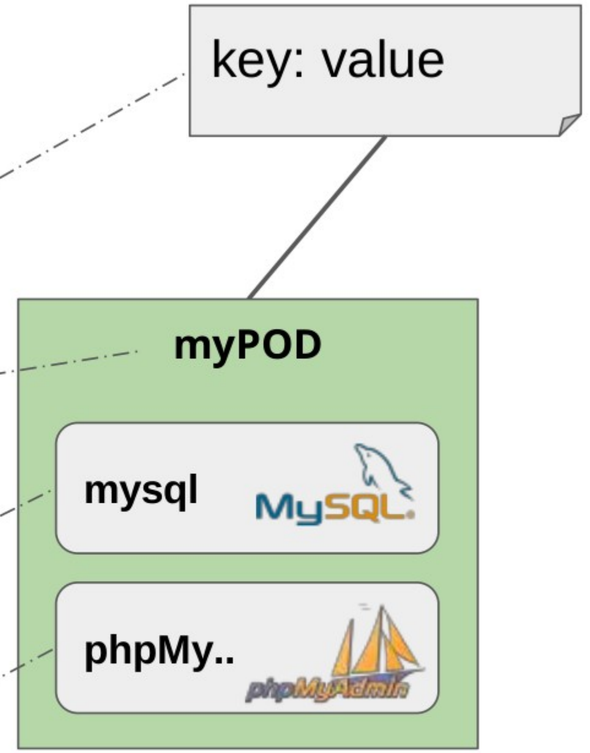
- The third is monitoring the nodes' health. The node controller is responsible for updating the NodeReady condition of NodeStatus to ConditionUnknown when a node becomes unreachable (i.e. the node controller stops receiving heartbeats for some reason, e.g. due to the node being down), and then later evicting all the pods from the node (using graceful termination) if the node continues to be unreachable.

</>

Concept: POD

Defining a POD as YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: myPod
  labels:
    key: value
spec:
  containers:
    - name: mysql
      image: username/image
    - name: phpMyAdmin
      image: username/image2
```

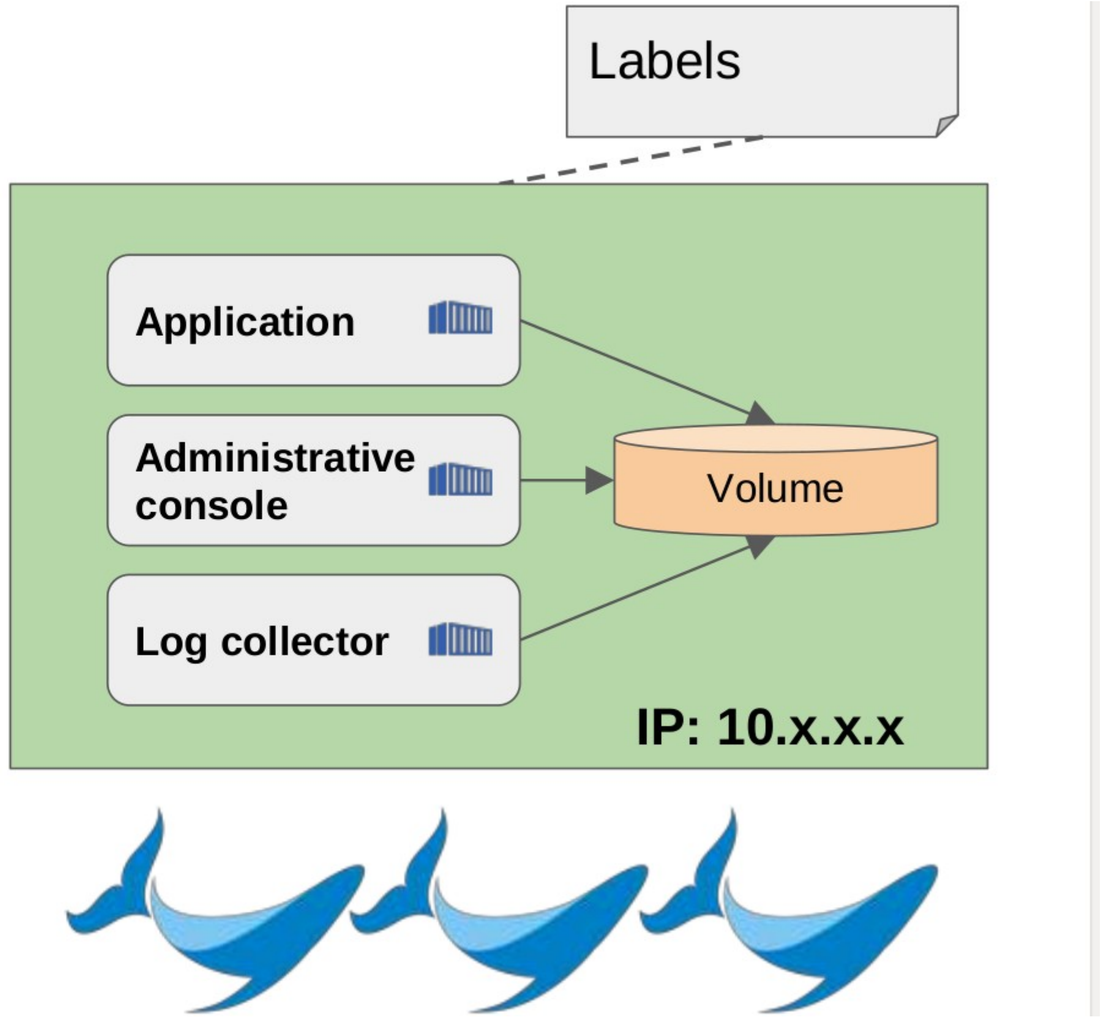


<devAcademy>

</>

Concept: Pod

- Group of containers
- Live and die together
- Share:
  - IP
  - Secrets
  - Labels *
  - Volumes *



** we will talk about these concepts later*

</>

Pods

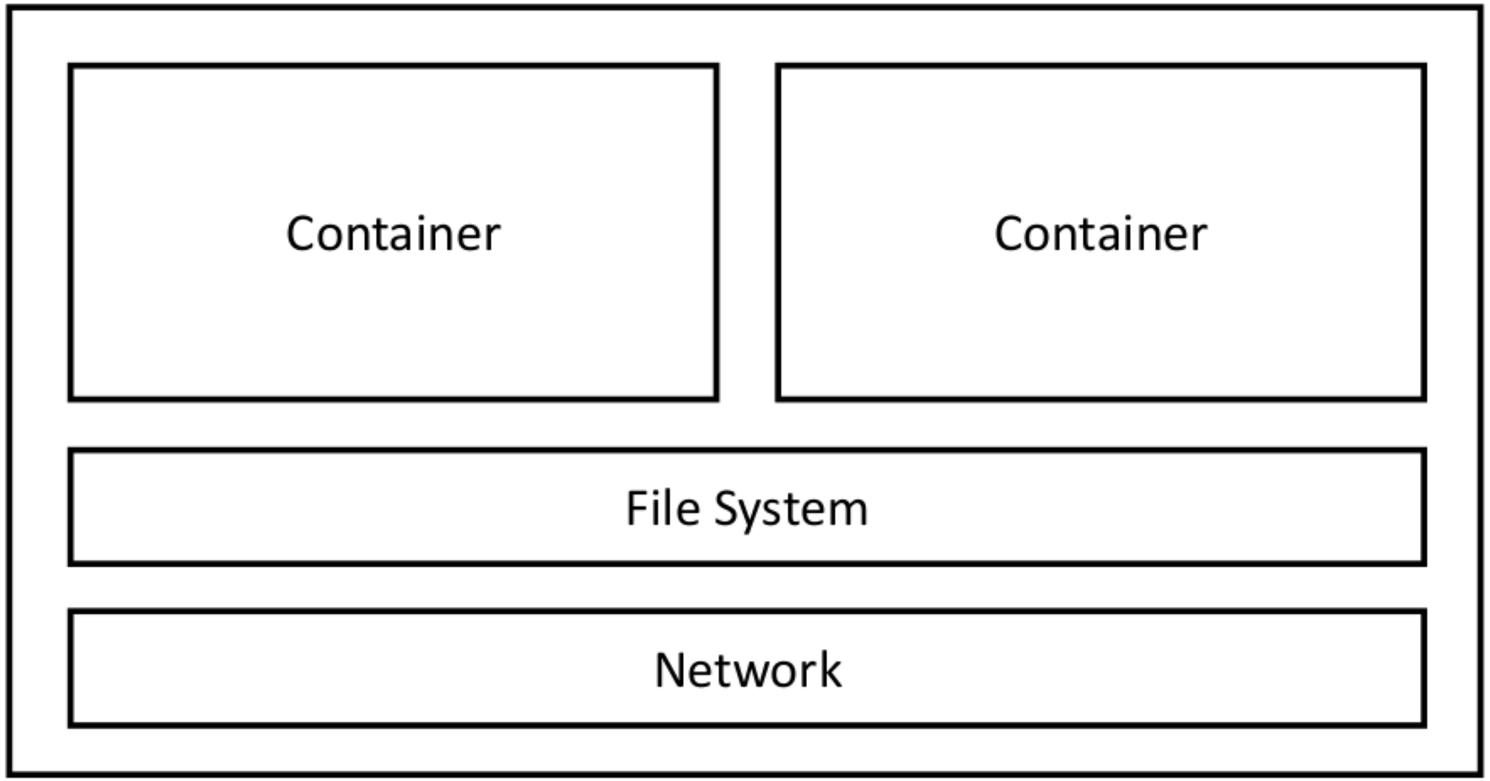
The primary primitive for running containerized workload.

- Unit of scheduling / resource envelope
- May contain one or more containers
- Containers share network, volumes, isolation components

</>

Pods Cont.

What does a pod look like?



</>

What are all these pods?

- **etcd is** our etcd server
- **kube-apiserver** is the API server
- **kube-controller-manager** and kube-scheduler are other master components
- **kube-dns** is an additional component (not mandatory but super useful, so it's there)
- **kube-proxy** is the (per-node) component managing port mappings and such
- **weave is** the (per-node) component managing the network overlay

the READY column indicates the number of containers in each pod

the pods with a name ending with -node1 are the master components (they have been specifically "pinned" to the master node)

</>

Kubernetes Manifest

YAML file to declare desired state of Kubernetes object types.

- Define Kubernetes type
- Define type specification
- Labels / Annotations
- Metadata

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      containers:
      - name: azure-vote-front
        image: microsoft/azure-vote-front:redis-v1
        ports:
        - containerPort: 80
        env:
        - name: REDIS
          value: "azure-vote-back"
```

</>

Replication Controller

- A ReplicationController ensures that a specified number of pod replicas are running at any one time.

- In other words, a ReplicationController makes sure that a pod or a homogeneous set of pods is always up and available.

</>

Replication Controller

Run the example job by downloading the example file and then running this command:

$ kubectl apply -f https://k8s.io/examples/controllers/replication.yaml

replicationcontroller/nginx created

$ kubectl describe replicationcontrollers/nginx

To list all the pods that belong to the ReplicationController in a machine readable form, you can use a command like this:

pods=$(kubectl get pods --selector=app=nginx --output=jsonpath={.items..metadata.name})

echo $pods

<devAcademy>

</>

Replication Controller

- How a ReplicationController Works
- If there are too many pods, the ReplicationController terminates the extra pods.
- If there are too few, the ReplicationController starts more pods.
- Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated.

</>

Replication Controller

- For example, your pods are re-created on a node after disruptive maintenance such as a kernel upgrade.

- For this reason, you should use a ReplicationController even if your application requires only a single pod.

- A ReplicationController is similar to a process supervisor, but instead of supervising individual processes on a single node, the ReplicationController supervises multiple pods across multiple nodes.

- ReplicationController is often abbreviated to “rc” or “rcs” in discussion, and as a shortcut in kubectl commands.

- A simple case is to create one ReplicationController object to reliably run one instance of a Pod indefinitely. A more complex use case is to run several identical replicas of a replicated service, such as web servers.

</>

Concept: Replication Controllers / Deployment

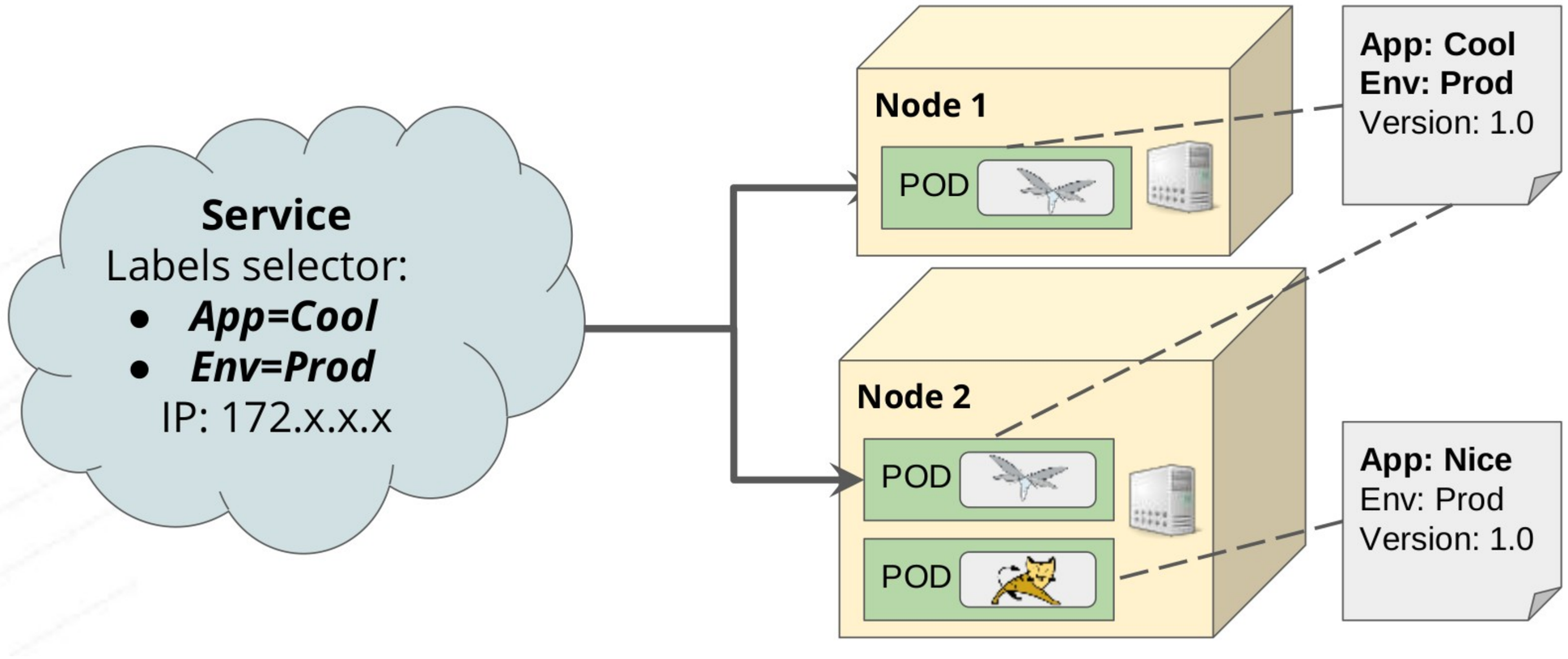
Defining a Deployment as YAML:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: myDeployment
spec:
  replicas: 4
  template:
    metadata:
    spec:
```

```
apiVersion: v1
kind: Pod
metadata:
  name: myPod
  labels:
    key: value
spec:
  containers:
    - name: myPod
      image: username/image
  ports:
      - name: http
        containerPort: 8080
```

</>

Concept: Services



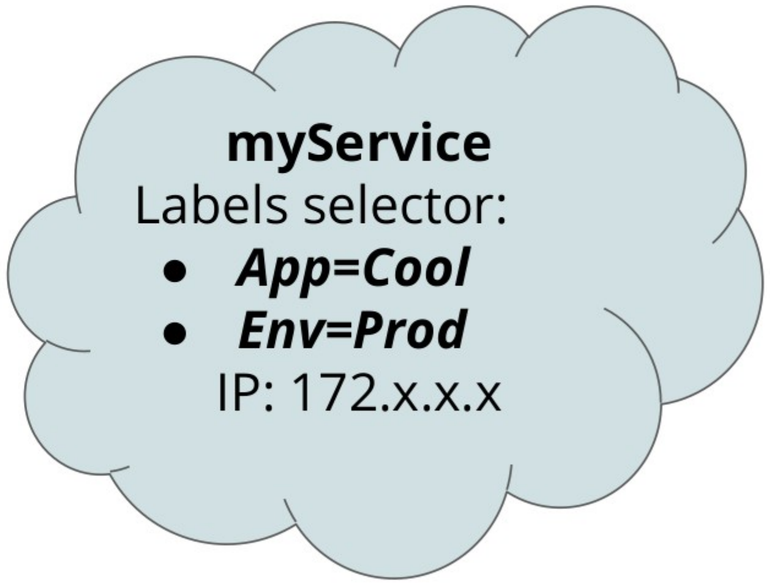
<devAcademy>

</>

Concept: Services

Defining a Service as YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: myService
  labels:
        ...
spec:
  ports:
    - port: 80
      targetPort: 80
  selector:
    App: Cool
    Env: Prod
```



</>

Kubernetes Services

Kubernetes object expressing pod networking endpoint (internal / external IP address).

- Service is associate with pod through label selector
- ClusterIP – exposed on cluster-internal IP
- NodePort – exposed on nodes IP .via static port
- LoadBalancer – exposed externally on providers NLB
- ExternalName – map service to DNS name

</>

ClusterIP services

A ClusterIP service is internal, available from the cluster only

This is useful for introspection from within containers

Try to connect to the API.

-k is used to skip certificate verification

Make sure to replace 10.96.0.1 with the CLUSTER-IP shown by $ kubectl get svc

```
curl -k https://10.96.0.1
The error that we see is expected: the Kubernetes API requires authentication.
```

<devAcademy>

</>

Exposing containers

- kubectl expose creates a service for existing pods
- A service is a stable address for a pod (or a bunch of pods)
- If we want to connect to our pod(s), we need to create a service
- Once a service is created, kube-dns will allow us to resolve it by name (i.e. after creating service hello, the name hello will resolve to something)
- There are different types of services, detailed on the following slides:
- ClusterIP, NodePort, LoadBalancer, ExternalName

</>

Basic service types

- **ClusterIP (default type)**

a virtual IP address is allocated for the service (in an internal, private range) this IP address is reachable only from within the cluster (nodes and pods) our code can connect to the service using the original port number

- **NodePort**

a port is allocated for the service (by default, in the 30000-32768 range) that port is made available on all our nodes and anybody can connect to it our code must be changed to connect to that new port number

These service types are always available.

Under the hood: kube-proxy is using a userland proxy and a bunch of iptables rules.

</>

More service types

LoadBalancer

an external load balancer is allocated for the service

the load balancer is configured accordingly (e.g.: a NodePort service is created, and the load balancer sends traffic to that port)

ExternalName

the DNS entry managed by kube-dns will just be a CNAME to a provided record

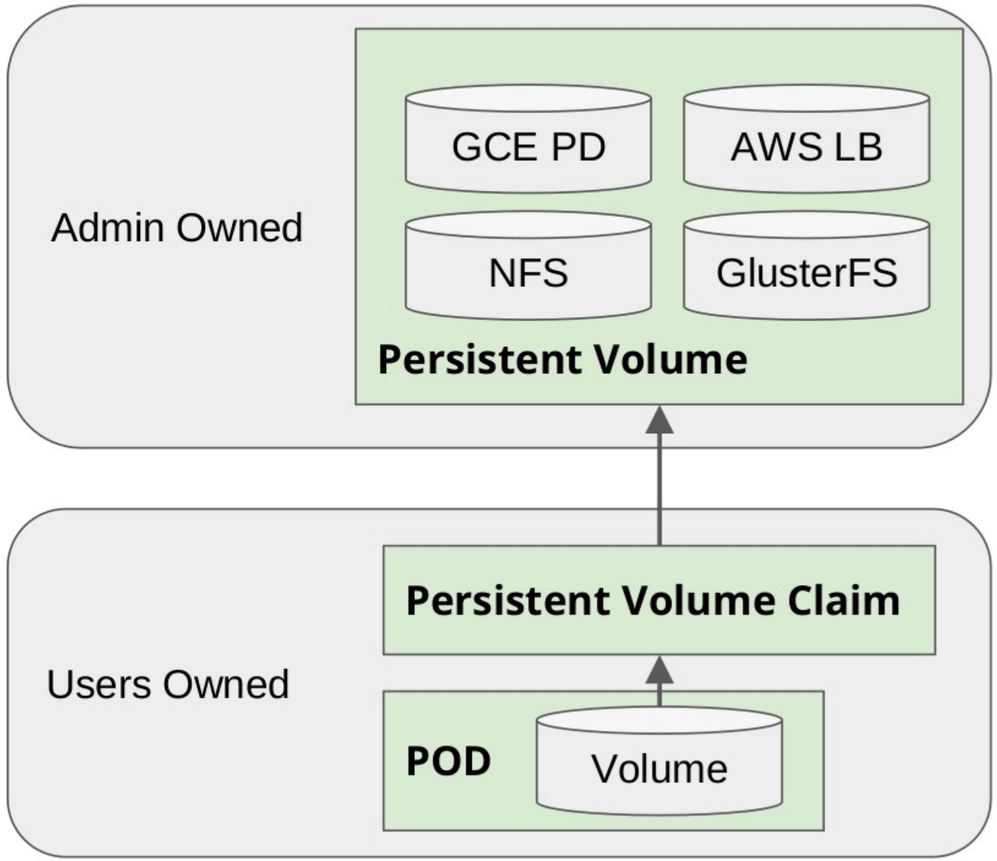
no port, no IP address, no nothing else is allocated

</>

Concept: Persistent Volumes

- Admin provisions them, Users claim them
- High-level abstraction
- Pods can mount PVCs as Volumes

```
  volumeMounts:
    # name must match the volume name below
  - name: mysql-persistent-volume
    # mount path within the container
    mountPath: /var/lib/mysql/data
volumes:
- name: mysql-persistent-volume
  persistentVolumeClaim:
    claimName: mysql-pvc
```



</>

ConfigMaps and secrets

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. This page provides a series of usage examples demonstrating how to create ConfigMaps and configure Pods using data stored in ConfigMaps.

There are many ways to create config files, such as

- Create ConfigMaps from directories
- Create ConfigMaps from files
- Create ConfigMaps from literal values

</>

ConfigMaps and secrets

There are a couple of things to note here:

That the ConfigMap has a Data section which is really just a key-value store

What's really interesting is that the original file name (and we could have used multiple files so would have ended up with multiple keys in the Data section) that we used is used as the name of the key inside this ConfigMap Data section.

The other interesting thing is what happens with the original file(s) content. The content from the file/files is used as the value for the key (representing the file) inside of the Data section.

</>

ConfigMaps and secrets

Well now that we have a ConfigMap we need to think about how to use it inside our own pods. Again there are several different ways, where I will concentrate on one of them

- Define pod ENV variables using ConfigMap data
- Configure all key-value pairs in a ConfigMap as Pod environment variables
- Use ConfigMap-defined environment variables in Pod commands
- Add ConfigMap data to a Volume

</>

ConfigMaps and secrets

What is a Secret?

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure.

Users can create secrets, and the system also creates some secrets.

To use a secret, a pod needs to reference the secret. A secret can be used with a pod in two ways: as files in a volume mounted on one or more of its containers, or used by kubelet when pulling images for the pod.

</>

ConfigMaps and secrets

Kubernetes secret objects let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Putting this information in a secret is safer and more flexible than putting it verbatim in a Pod definition or in a container image

</>

ConfigMaps and secrets

Right now there is not much difference between secretes and ConfigMaps in Kubernetes. The only real difference is how you create the data that you want to be stored in the first place, where the recommendation is to use Base64 encoded values for your secrets

There is a slightly different command like to run, and the way you mount the volume in your pod is also slightly different, but conceptually its not that different (right now, but I would imagine this might change to use some other mechanism over time).

</>

ConfigMaps and secrets

Base64 Encoding

So the recommendation is to base64 encode our secret value

Or you could just use one of the many base64 encoding/decoding sites on line such as : https://www.base64decode.org/

</>

Annotations

You can use Kubernetes annotations to attach arbitrary non-identifying metadata to objects. Clients such as tools and libraries can retrieve this metadata.

You can use either labels or annotations to attach metadata to Kubernetes objects. Labels can be used to select objects and to find collections of objects that satisfy certain conditions. In contrast, annotations are not used to identify and select objects. The metadata in an annotation can be small or large, structured or unstructured, and can include characters not permitted by labels.

Annotations, like labels, are key/value maps:

```
"metadata": {
  "annotations": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

</>

Annotations

Here are some examples of information that could be recorded in annotations:

- Fields managed by a declarative configuration layer. Attaching these fields as annotations distinguishes them from default values set by clients or servers, and from auto-generated fields and fields set by auto-sizing or auto-scaling systems.
- Build, release, or image information like timestamps, release IDs, git branch, PR numbers, image hashes, and registry address.
- Pointers to logging, monitoring, analytics, or audit repositories.
- Client library or tool information that can be used for debugging purposes: for example, name, version, and build information.

</>

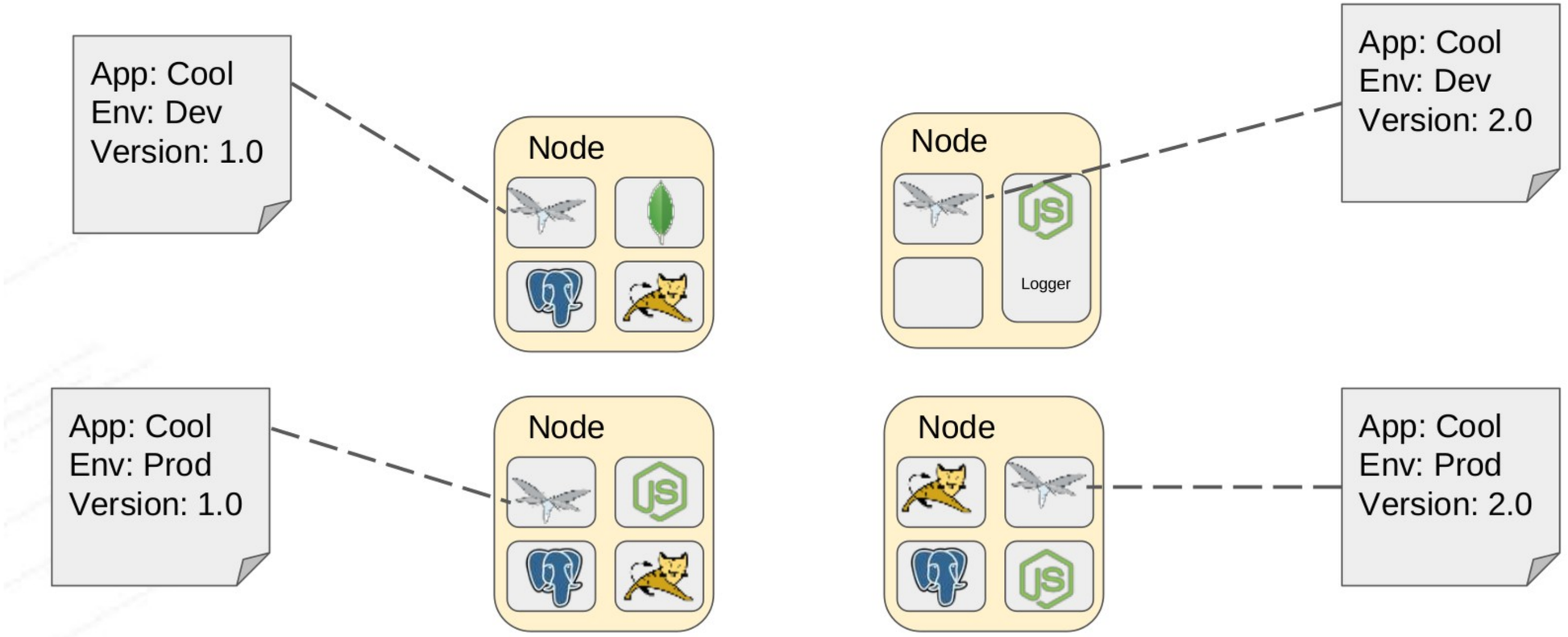
Annotations

- User or tool/system provenance information, such as URLs of related objects from other ecosystem components.
- Lightweight rollout tool metadata: for example, config or checkpoints.
- Phone or pager numbers of persons responsible, or directory entries that specify where that information can be found, such as a team web site.
- Directives from the end-user to the implementations to modify behavior or engage non-standard features.
- Instead of using annotations, you could store this type of information in an external database or directory, but that would make it much harder to produce shared client libraries and tools for deployment, management, introspection, and the like.

</>

Concept: Labels

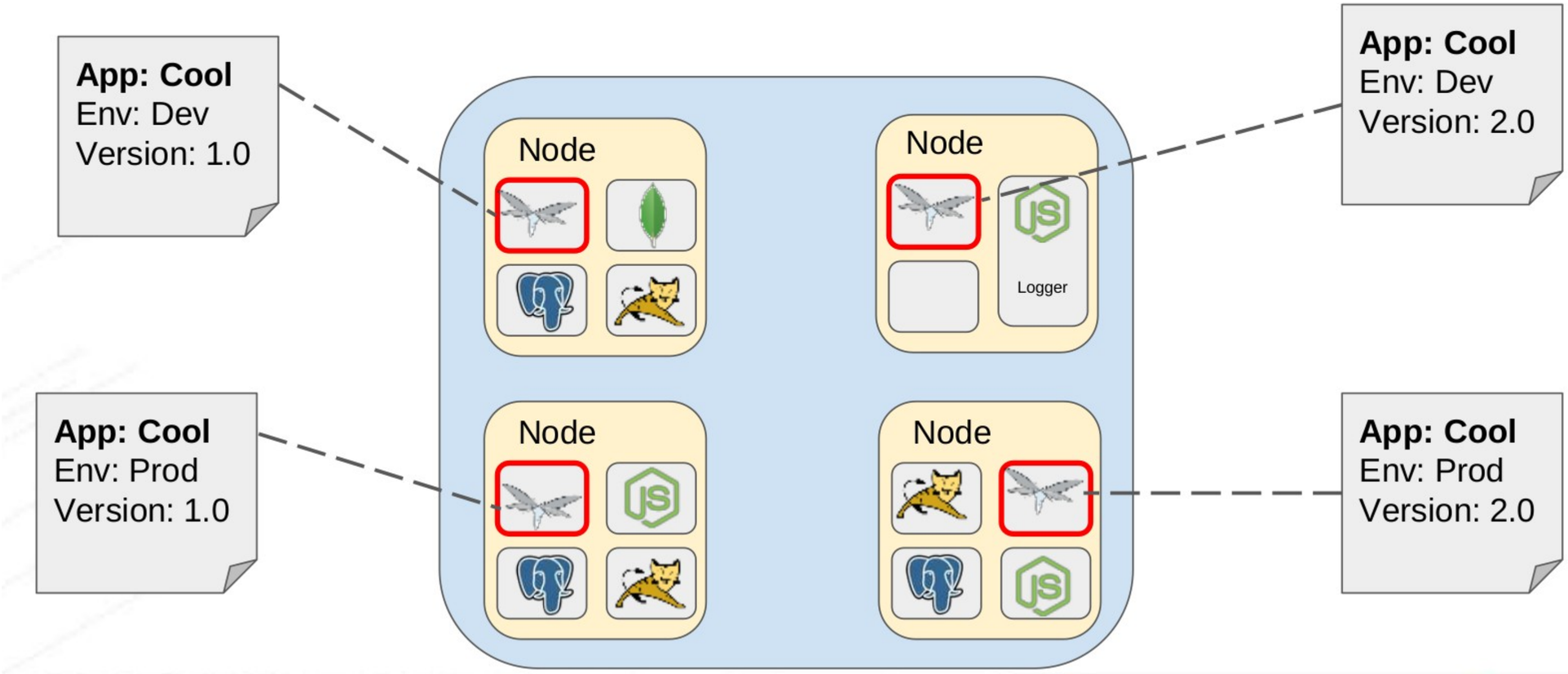
Everything in Kubernetes can have a label



<devAcademy>

</>

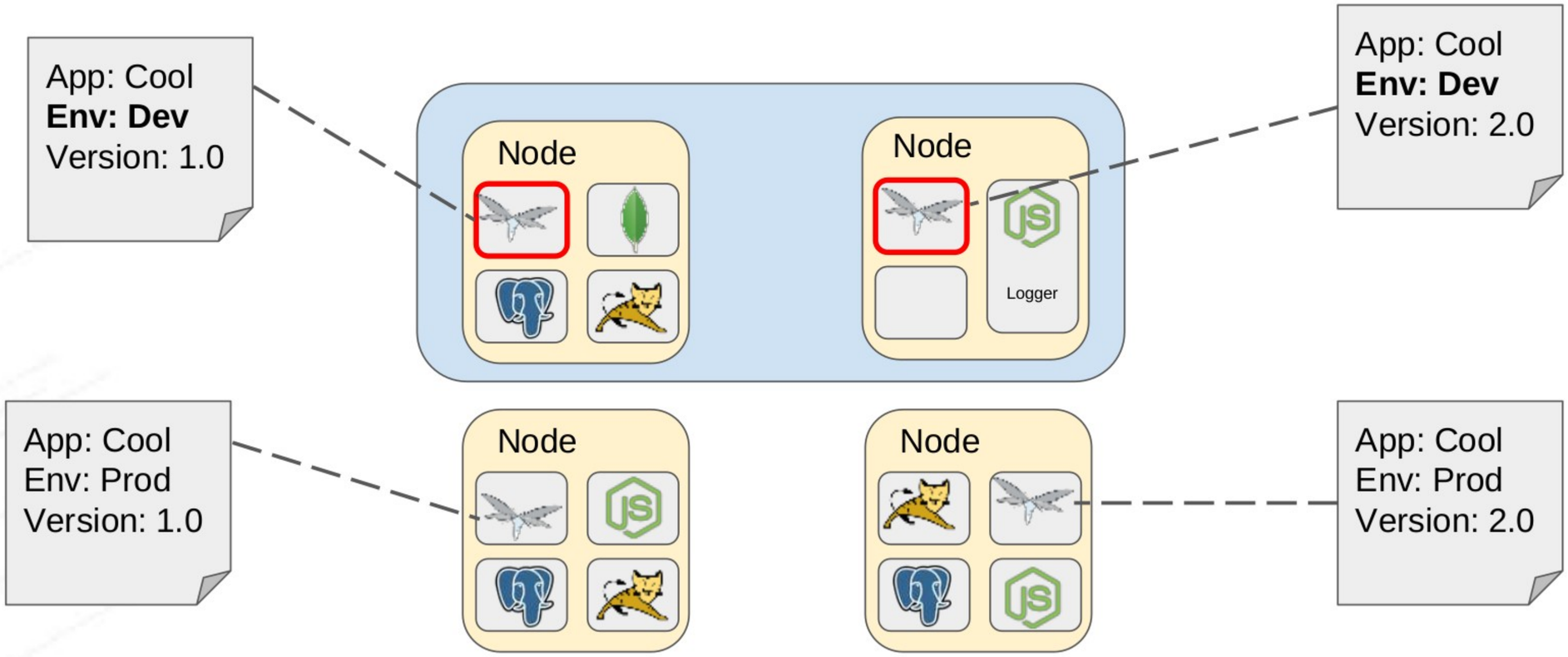
Concept: Labels



<devAcademy>

</>

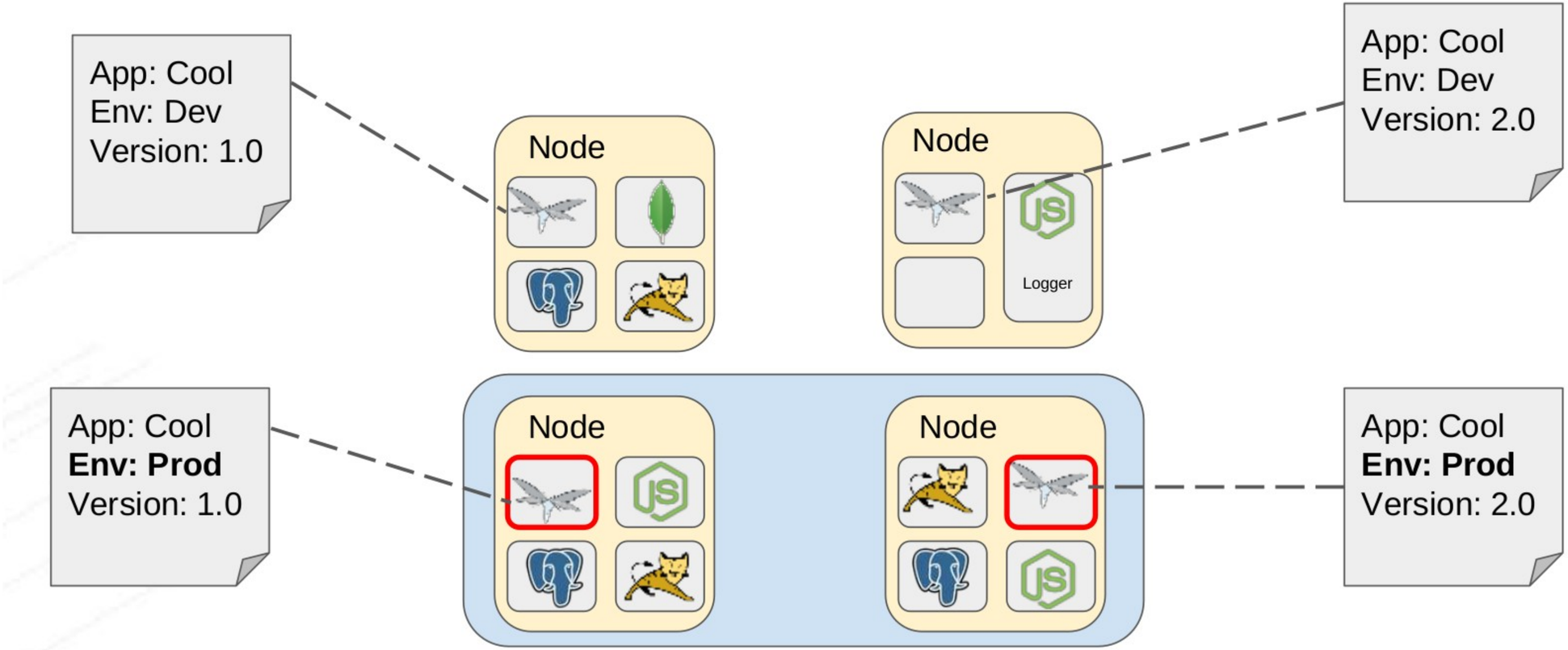
Concept: Labels



<devAcademy>

</>

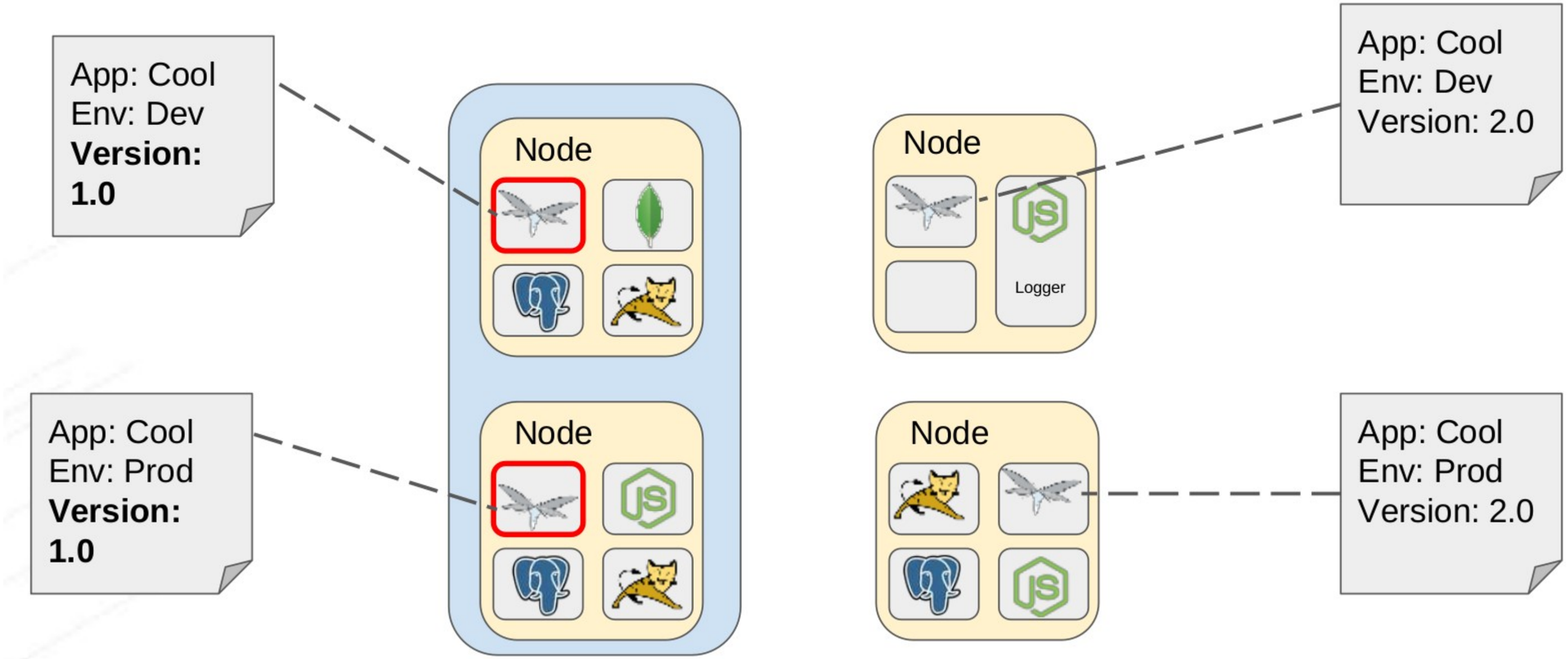
Concept: Labels



<devAcademy>

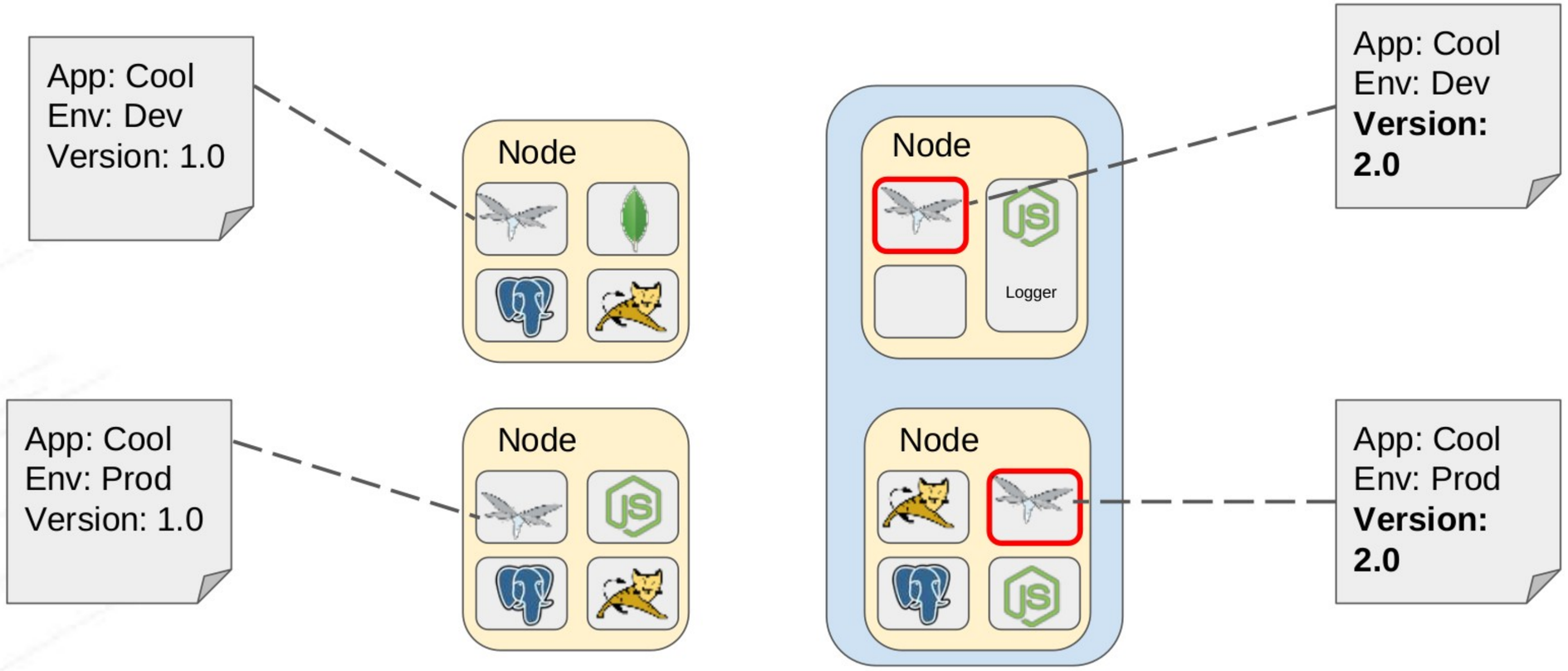
</>

Concept: Labels



<devAcademy>

Concept: Labels



</>

Concept: Labels

Defining Labels as YAML:
(can be placed in any object metadata)

```
metadata:
    name: objectName
    labels:
        App: Cool
        Env: Dev
        Version: 1.0
```

App: Cool
Env: Dev
Version: 1.0

<devAcademy>

</>

Label selectors

- Unlike names and UIDs, labels do not provide uniqueness. In general, we expect many objects to carry the same label(s).
- Via a label selector, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.
- The API currently supports two types of selectors: equality-based and set-based. A label selector can be made of multiple requirements which are comma-separated. In the case of multiple requirements, all must be satisfied so the comma separator acts as a logical AND (&&) operator.
- The semantics of empty or non-specified selectors are dependent on the context, and API types that use selectors should document the validity and meaning of them.

</>

```
apiVersion: v1
kind: Pod
metadata:
  name: cuda-test
spec:
  containers:
    - name: cuda-test
      image: "k8s.gcr.io/cuda-vector-add:v0.1"
      resources:
        limits:
          nvidia.com/gpu: 1
  nodeSelector:
    accelerator: nvidia-tesla-p100
```

</>

Kubernetes Governance

Like all systems, security and governance should be a concern.

- Limit access to resources
- Provide isolated compute environments
- Ensure cluster safety

</>

Namespaces

Unit of multi-tenancy cluster isolation.

- Provide isolated environments for teams / projects
- Can be accessed controlled
- Not necessary for running multiple applications
- Not every Kubernetes object can be added to a namespace

</>

Namespaces

- Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

- Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide.

- Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces can not be nested inside one another and each Kubernetes resource can only be in one namespace.

</>

Namespaces

- Namespaces are a way to divide cluster resources between multiple users (via resource quota).

- In future versions of Kubernetes, objects in the same namespace will have the same access control policies by default.

- It is not necessary to use multiple namespaces just to separate slightly different resources, such as different versions of the same software: use labels to distinguish resources within the same namespace.

</>

```
jvherrera@juanvi-HP-ZBook-14u-G5:~$ k get namespaces
NAME              STATUS   AGE
default           Active   17d
development       Active   14d
kube-node-lease   Active   17d
kube-public       Active   17d
kube-system       Active   17d
```

</>

Namespaces

Not All Objects are in a Namespace

Most Kubernetes resources (e.g. pods, services, replication controllers, and others) are in some namespaces. However namespace resources are not themselves in a namespace. And low-level resources, such as nodes and persistentVolumes, are not in any namespace.

To see which Kubernetes resources are and aren’t in a namespace:

```
# In a namespace
kubectl api-resources --namespaced=true

# Not in a namespace
kubectl api-resources --namespaced=false
```

</>

Ingress

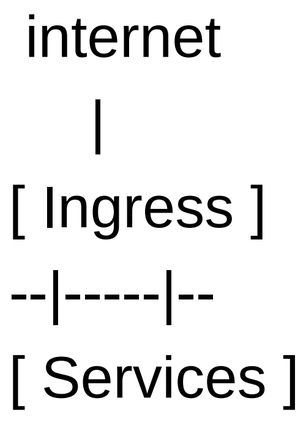
An API object that manages external access to the services in a cluster, typically HTTP.

Ingress can provide load balancing, SSL termination and name-based virtual hosting.

</>

What is Ingress?

Ingress, added in Kubernetes v1.1, exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.



</>

What is Ingress?

An Ingress can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, and offer name based virtual hosting. An Ingress controller is responsible for fulfilling the Ingress, usually with a loadbalancer, though it may also configure your edge router or additional frontends to help handle the traffic.

An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.

</>

The Ingress Resource

A minimal ingress resource example:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
      paths:
      - path: /testpath
        backend:
          serviceName: test
          servicePort: 80
```

Ingress rules

Each http rule contains the following information:

- An optional host. In this example, no host is specified, so the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided (for example, foo.bar.com), the rules apply to that host.
- a list of paths (for example, /testpath), each of which has an associated backend defined with a serviceName and servicePort. Both the host and path must match the content of an incoming request before the loadbalancer will direct traffic to the referenced service.
- A backend is a combination of service and port names as described in the services doc. HTTP (and HTTPS) requests to the Ingress matching the host and path of the rule will be sent to the listed backend.
- A default backend is often configured in an Ingress controller that will service any requests that do not match a path in the spec.

</>

Default Backend

- An Ingress with no rules sends all traffic to a single default backend. The default backend is typically a configuration option of the Ingress controller and is not specified in your Ingress resources.

- If none of the hosts or paths match the HTTP request in the Ingress objects, the traffic is routed to your default backend.

Single Service Ingress

There are existing Kubernetes concepts that allow you to expose a single Service (see alternatives). You can also do this with an Ingress by specifying a default backend with no rules.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

Ingress

If you create it using kubectl apply -f you should see:

kubectl get ingress test-ingress

NAME	HOSTS	ADDRESS	PORTS	AGE
test-ingress	*	107.178.254.228	80	59s

Where 107.178.254.228 is the IP allocated by the Ingress controller to satisfy this Ingress.

</>

Ingress Example

A **fanout** configuration routes traffic from a single IP address to more than one service, based on the HTTP URI being requested.

An Ingress allows you to keep the number of loadbalancers down to a minimum. For example, a setup like:

foo.bar.com -> 178.91.123.132 -> / foo service1:4200
/ bar service2:8080

</>

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

When you create the ingress with kubectl apply -f:

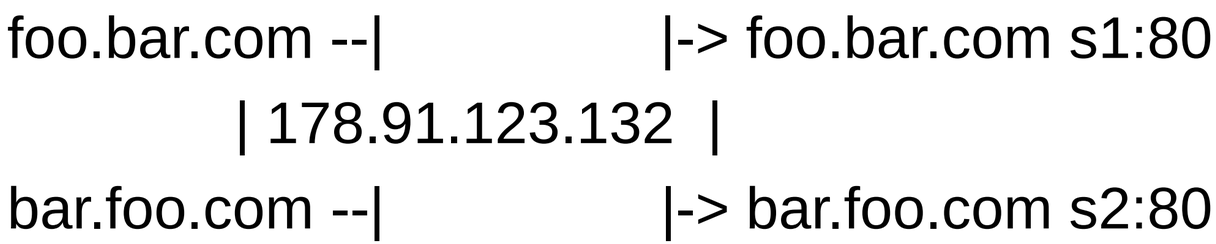
```
kubectl describe ingress simple-fanout-example
Name:             simple-fanout-example
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host         Path  Backends
  ----         ----  --------
  foo.bar.com
               /foo   service1:4200 (10.8.0.90:4200)
               /bar   service2:8080 (10.8.0.91:8080)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target:  /
Events:
  Type     Reason  Age              From                     Message
  ----     ------  ----             ----                     -------
  Normal   ADD     22s              loadbalancer-controller  default/test
```

</>

Ingress Example

Name based virtual hosting

Name-based virtual hosts support routing HTTP traffic to multiple host names at the same IP address.



The following Ingress tells the backing loadbalancer to route requests based on the Host header.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - backend:
          serviceName: service1
          servicePort: 80
  - host: bar.foo.com
    http:
      paths:
      - backend:
          serviceName: service2
          servicePort: 80
```

TLS

- You can secure an Ingress by specifying a secret that contains a TLS private key and certificate.
- Currently the Ingress only supports a single TLS port, 443, and assumes TLS termination.
- If the TLS configuration section in an Ingress specifies different hosts, they will be multiplexed on the same port according to the hostname specified through the SNI TLS extension (provided the Ingress controller supports SNI).
- The TLS secret must contain keys named tls.crt and tls.key that contain the certificate and private key to use for TLS, e.g.:

```
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

</>

Referencing this secret in an Ingress will tell the Ingress controller to secure the channel from the client to the loadbalancer using TLS. You need to make sure the TLS secret you created came from a certificate that contains a CN for sslexample.foo.com.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
  - hosts:
    - sslexample.foo.com
    secretName: testsecret-tls
  rules:
    - host: sslexample.foo.com
      http:
        paths:
        - path: /
          backend:
            serviceName: service1
            servicePort: 80
```

<devAcademy>

</>

Node Scaling

Provide more or less computer infrastructure for the cluster.

- Manually scale
- Auto scale (depending on provider)
- Take care to provide stateless workload

</>

Pod Scaling

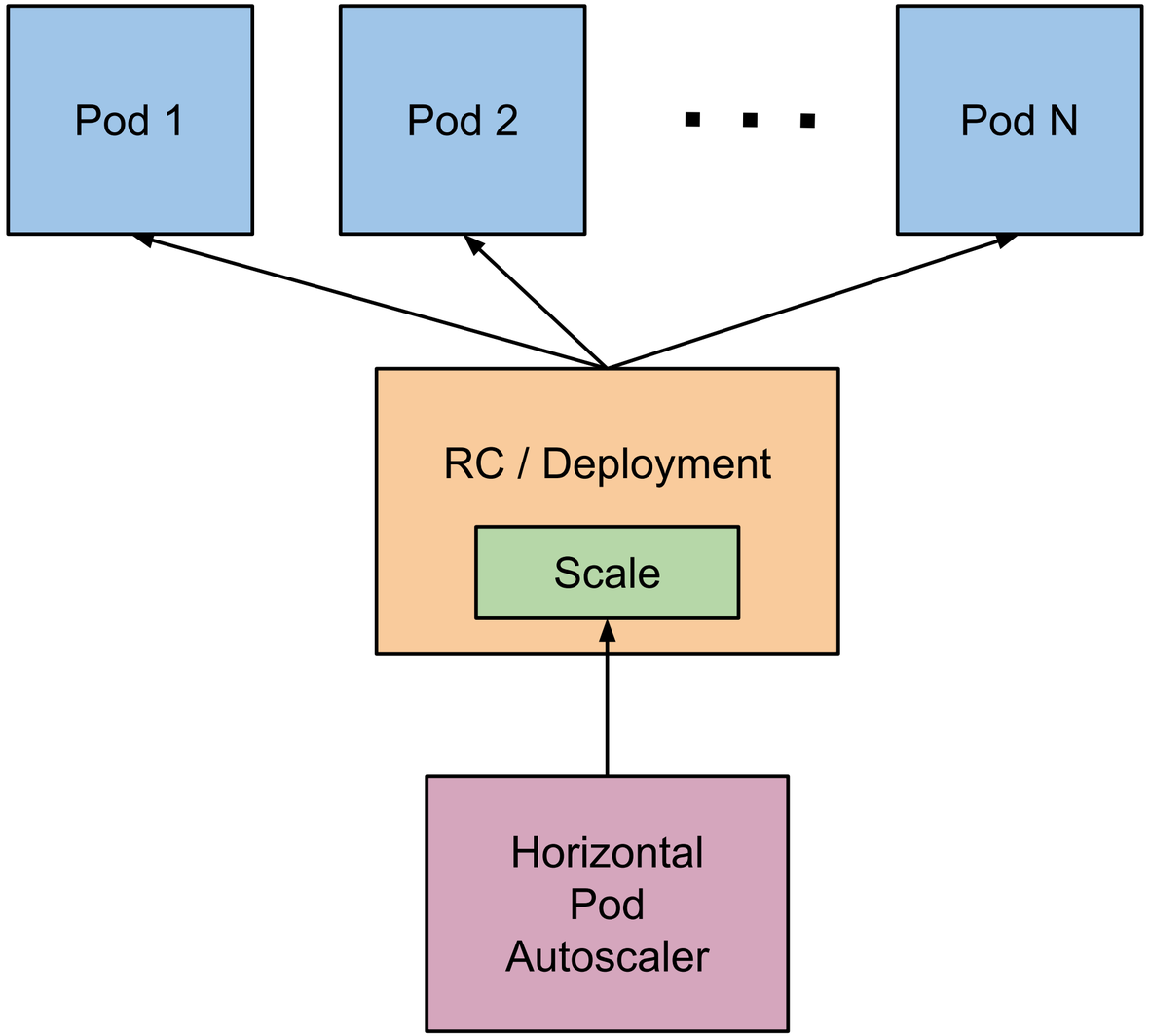
Provide more or less processing power for an application.

- Manually scale by increasing pod replica set count
- Auto scale using Horizontal Pod Autoscaling (HPA)
- Provide custom metrics for scale operations

</>

Horizontal Pod Autoscaler

- The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU utilization (or, with custom metrics support, on some other application-provided metrics).
- Note that Horizontal Pod Autoscaling does not apply to objects that can be scaled, for example, DaemonSets.
- The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller.
- The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.


Pod 1
Pod 2
Pod N
RC / Deployment
Scale
Horizontal
Pod
Autoscaler

</>

Support for Horizontal Pod Autoscaler in kubectl

Horizontal Pod Autoscaler, like every API resource, is supported in a standard way by kubectl. We can create a new autoscaler using kubectl create command. We can list autoscalers by kubectl get hpa and get detailed description by kubectl describe hpa. Finally, we can delete an autoscaler using kubectl delete hpa.

In addition, there is a special kubectl autoscale command for easy creation of a Horizontal Pod Autoscaler. For instance, executing kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80 will create an autoscaler for replication set foo, with target CPU utilization set to 80% and the number of replicas between 2 and 5.

</>

Autoscaling during rolling update

Currently in Kubernetes, it is possible to perform a rolling update by managing replication controllers directly, or by using the deployment object, which manages the underlying replica sets for you. Horizontal Pod Autoscaler only supports the latter approach: the Horizontal Pod Autoscaler is bound to the deployment object, it sets the size for the deployment object, and the deployment is responsible for setting sizes of underlying replica sets.

Horizontal Pod Autoscaler does not work with rolling update using direct manipulation of replication controllers, i.e. you cannot bind a Horizontal Pod Autoscaler to a replication controller and do rolling update (e.g. using kubectl rolling-update). The reason this doesn’t work is that when rolling update creates a new replication controller, the Horizontal Pod Autoscaler will not be bound to the new replication controller.

</>

Extending Kubernetes

Kubernetes is extensible to provide additional capability.

- Custom Resource Definitions (CRD)
- Operators
- Aggregated API extensions

</>

Extending Kubernetes (CRD)

Custom Resource Definitions define a new Kuberntes object type (API Object).

- No code necessary *
- Instances of a CRD contain structured data
- Provides identical kubectl operational capibilities

ReplicaSet

- A ReplicaSet’s purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.
- A ReplicaSet is defined with fields, including a selector that specifies how to identify Pods it can acquire, a number of replicas indicating how many Pods it should be maintaining, and a pod template specifying the data of new Pods it should create to meet the number of replicas criteria. A ReplicaSet then fulfills its purpose by creating and deleting Pods as needed to reach the desired number. When a ReplicaSet needs to create new Pods, it uses its Pod template.
- A ReplicaSet identifies new Pods to acquire by using its selector. If there is a Pod that has no OwnerReference or the OwnerReference is not a controller and it matches a ReplicaSet’s selector, it will be immediately acquired by said ReplicaSet.

ReplicaSet

When to use a ReplicaSet

- A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don’t require updates at all.

- This actually means that you may never need to manipulate ReplicaSet objects: use a Deployment instead, and define your application in the spec section.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

Saving this manifest into frontend.yaml and submitting it to a Kubernetes cluster will create the defined ReplicaSet and the Pods that it manages.

kubectl apply -f https://kubernetes.io/examples/controllers/frontend.yaml

You can then get the current ReplicaSets deployed:

kubectl get rs

And see the frontend one you created:

```
NAME       DESIRED   CURRENT   READY   AGE
frontend   3         3         3       6s
```

You can also check on the state of the replicaset:

kubectl describe rs/frontend

And lastly you can check for the Pods brought up:

kubectl get Pods

ReplicaSet as a Horizontal Pod Autoscaler Target

</>

A ReplicaSet can also be a target for Horizontal Pod Autoscalers (HPA). That is, a ReplicaSet can be auto-scaled by an HPA. Here is an example HPA targeting the ReplicaSet we created in the previous example.

controllers/hpa-rs.yaml Copy controllers/hpa-rs.yaml to clipboard

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-scaler
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: frontend
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Saving this manifest into hpa-rs.yaml and submitting it to a Kubernetes cluster should create the defined HPA that autoscales the target ReplicaSet depending on the CPU usage of the replicated Pods.

kubectl apply -f https://k8s.io/examples/controllers/hpa-rs.yaml

Alternatively, you can use the kubectl autoscale command to accomplish the same (and it’s easier!)

```
kubectl autoscale rs frontend --max=10
```

</>

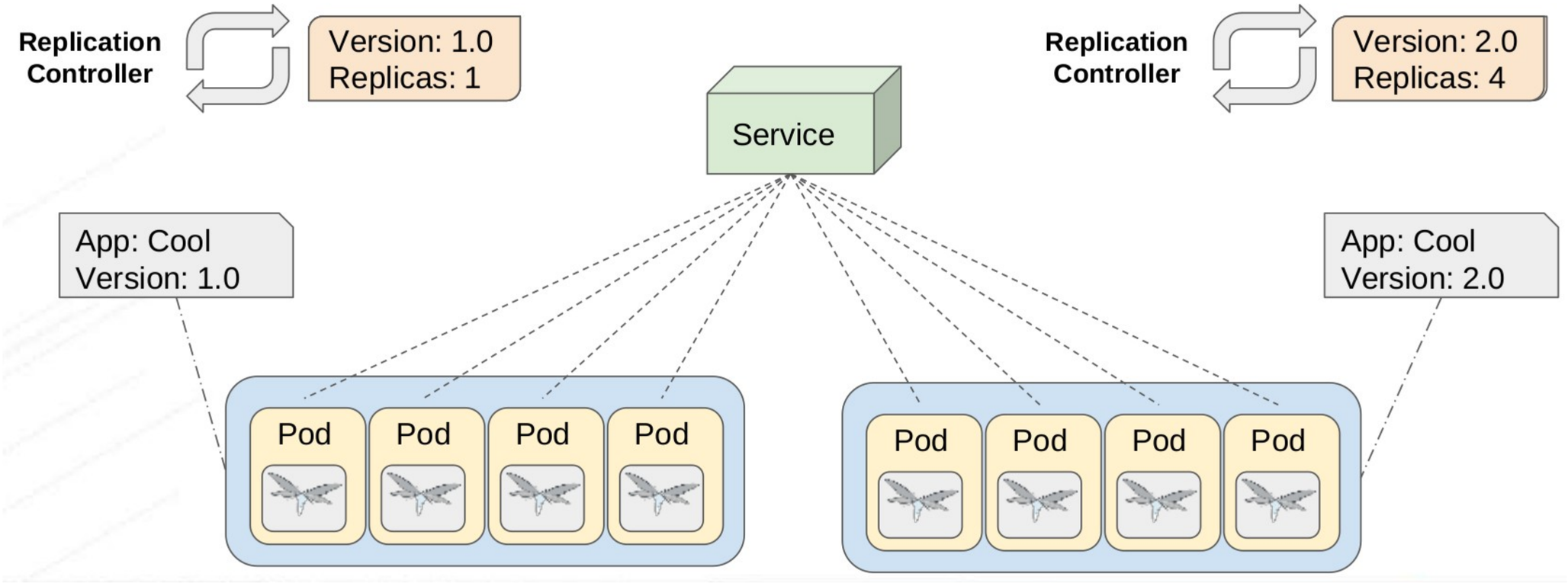
Kubernetes Deployments

Pod scheduling abstract.

- Contains, POD, Container, and replica spec
- Replica set – how many instances of a pod

</>

Deployment Concept: Rolling Updates



<devAcademy>

</>

Rolling Deployments

Zero downtime application updates.

- Deployment must contain more than one replica
- Pod are incrementally updated
- Configurable update schema (max unavailable / available)
- Rollback to previous version

<devAcademy>

</>

Rolling updates

By default (without rolling updates), when a scaled resource is updated:

new pods are created

old pods are terminated

… all at the same time

if something goes wrong, ¯_(ツ)_/¯

With rolling updates, when a resource is updated, it happens progressively

Two parameters determine the pace of the rollout: maxUnavailable and maxSurge

They can be specified in absolute number of pods, or percentage of the replicas count

At any given time …

there will always be at least replicas-maxUnavailable pods available

there will never be more than replicas+maxSurge pods in total

there will therefore be up to maxUnavailable+maxSurge pods being updated

We have the possibility to rollback to the previous version (if the update fails or is unsatisfactory in any way)

</>

Rolling updates in practice

As of Kubernetes 1.8, we can do rolling updates with:

deployments, daemonsets, statefulsets

Editing one of these resources will automatically result in a rolling update

Rolling updates can be monitored with the kubectl rollout subcommand

Deployments

A Deployment controller provides declarative updates for Pods and ReplicaSets.

- You describe a desired state in a Deployment object, and the Deployment controller changes the actual state to the desired state at a controlled rate.
- You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Creating a Deployment

The following is an example of a Deployment. It creates a ReplicaSet to bring up three nginx Pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

In this example:

- A Deployment named nginx-deployment is created, indicated by the .metadata.name field.
- The Deployment creates three replicated Pods, indicated by the replicas field.
- The selector field defines how the Deployment finds which Pods to manage. In this case, you simply select a label that is defined in the Pod template (app: nginx). However, more sophisticated selection rules are possible, as long as the Pod template itself satisfies the rule
- The template field contains the following sub-fields:
- The Pods are labeled app: nginxusing the labels field.
- The Pod template’s specification, or .template.spec field, indicates that the Pods run one container, nginx, which runs the nginx Docker Hub image at version 1.7.9.
- Create one container and name it nginx using the name field.
- Open port 80 so that the container can send and accept traffic.

To create this Deployment, run the following command:

kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml
Next, run kubectl get deployments. The output is similar to the following:

```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         0         0            0           1s
```

Run the kubectl get deployments again a few seconds later:

```
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3         3         3            3           18s
```

Statefulset

- StatefulSet is the workload API object used to manage stateful applications.
- StatefulSets are intended to be used with stateful applications and distributed systems. However, the administration of stateful applications and distributed systems on Kubernetes is a broad, complex topic. In order to demonstrate the basic features of a StatefulSet, and not to conflate the former topic with the latter, you will deploy a simple web application using a StatefulSet.
- Manages the deployment and scaling of a set of Pods , and provides guarantees about the ordering and uniqueness of these Pods.

Statefulset

- Like a Deployment , a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.

- A StatefulSet operates under the same pattern as any other Controller. You define your desired state in a StatefulSet object, and the StatefulSet controller makes any necessary updates to get there from the current state..

Statefulset

StatefulSets are valuable for applications that require one or more of the following.

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered, graceful deployment and scaling.
- Ordered, automated rolling updates.

In the above, stable is synonymous with persistence across Pod (re)scheduling. If an application doesn’t require any stable identifiers or ordered deployment, deletion, or scaling, you should deploy your application with a controller that provides a set of stateless replicas. Controllers such as Deployment or ReplicaSet may be better suited to your stateless needs.

Stateful’s Limitations

- StatefulSet was a beta resource prior to 1.9 and not available in any Kubernetes release prior to 1.5.
- The storage for a given Pod must either be provisioned by a PersistentVolume Provisioner based on the requested storage class, or pre-provisioned by an admin.
- Deleting and/or scaling a StatefulSet down will not delete the volumes associated with the StatefulSet. This is done to ensure data safety, which is generally more valuable than an automatic purge of all related StatefulSet resources.
- StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. You are responsible for creating this Service.
- StatefulSets do not provide any guarantees on the termination of pods when a StatefulSet is deleted. To achieve ordered and graceful termination of the pods in the StatefulSet, it is possible to scale the StatefulSet down to 0 prior to deletion.
- When using Rolling Updates with the default Pod Management Policy (OrderedReady), it’s possible to get into a broken state that requires manual intervention to repair

Pod Management Policies

In Kubernetes 1.7 and later, StatefulSet allows you to relax its ordering guarantees while preserving its uniqueness and identity guarantees via its .spec.podManagementPolicy field.

- OrderedReady Pod Management
- OrderedReady pod management is the default for StatefulSets. It implements the behavior described above.

Parallel Pod Management

Parallel pod management tells the StatefulSet controller to launch or terminate all Pods in parallel, and to not wait for Pods to become Running and Ready or completely terminated prior to launching or terminating another Pod. This option only affects the behavior for scaling operations. Updates are not affected.

Update Strategies

In Kubernetes 1.7 and later, StatefulSet’s .spec.updateStrategy field allows you to configure and disable automated rolling updates for containers, labels, resource request/limits, and annotations for the Pods in a StatefulSet.

On Delete

The OnDelete update strategy implements the legacy (1.6 and prior) behavior. When a StatefulSet’s .spec.updateStrategy.type is set to OnDelete, the StatefulSet controller will not automatically update the Pods in a StatefulSet. Users must manually delete Pods to cause the controller to create new Pods that reflect modifications made to a StatefulSet’s .spec.template.

Rolling Updates

The RollingUpdate update strategy implements automated, rolling update for the Pods in a StatefulSet. It is the default strategy when .spec.updateStrategy is left unspecified. When a StatefulSet’s .spec.updateStrategy.type is set to RollingUpdate, the StatefulSet controller will delete and recreate each Pod in the StatefulSet. It will proceed in the same order as Pod termination (from the largest ordinal to the smallest), updating each Pod one at a time. It will wait until an updated Pod is Running and Ready prior to updating its predecessor.

</>

Kubernetes network model

- TL,DR:

Our cluster (nodes and pods) is one big flat IP network.

- In detail:
  - all nodes must be able to reach each other, without NAT
  - all pods must be able to reach each other, without NAT
  - pods and nodes must be able to reach each other, without NAT
  - each pod is aware of its IP address (no NAT)

- Kubernetes doesn’t mandate any particular implementation

</>

Kubernetes network model: the good

- Everything can reach everything
- No address translation
- No port translation
- No new protocol
- Pods cannot move from a node to another and keep their IP address
- IP addresses don’t have to be “portable” from a node to another (We can use e.g. a subnet per node and use a simple routed topology)
- The specification is simple enough to allow many various implementations

</>

Kubernetes network model: the less good

- Everything can reach everything
- if you want security, you need to add network policies
- the network implementation that you use needs to support them
- There are literally dozens of implementations out there (15 are listed in the Kubernetes documentation)
- It looks like you have a level 3 network, but it’s only level 4 (The spec requires UDP and TCP, but not port ranges or arbitrary IP packets)
- kube-proxy is on the data path when connecting to a pod or container, and it’s not particularly fast (relies on userland proxying or iptables)

</>

Kubernetes network model: in practice

- The nodes that we are using have been set up to use Weave
- We don’t endorse Weave in a particular way, it just Works For Us
- Don’t worry about the warning about kube-proxy performance
- Unless you:
  - routinely saturate 10G network interfaces
  - count packet rates in millions per second
  - run high-traffic VOIP or gaming platforms
  - do weird things that involve millions of simultaneous connections (in which case you’re already familiar with kernel tuning)

</>

Kubernetes Dashboard

Graphical user interface for interacting with a Kubernetes cluster.

- Create, update, delete objects
- Visual representation of state
- Take care to properly secure

</>

Behind the scenes of kubectl run

Let’s look at the resources that were created by kubectl run

List most resource types:

kubectl get all

We should see the following things:

deploy/pingpong (the deployment that we just created)

rs/pingpong-xxxx (a replica set created by the deployment)

po/pingpong-yyyy (a pod created by the replica set)

</>

What are these different things?

A deployment is a high-level construct

allows scaling, rolling updates, rollbacks

multiple deployments can be used together to implement a canary deployment

delegates pods management to replica sets

A replica set is a low-level construct

makes sure that a given number of identical pods are running

allows scaling

rarely used directly

A replication controller is the (deprecated) predecessor of a replica set

</>

Our pingpong deployment

kubectl run created a deployment, deploy/pingpong

That deployment created a replica set, rs/pingpong-xxxx

That replica set created a pod, po/pingpong-yyyy

We'll see later how these folks play together for:

scaling

high availability

rolling updates

</>

Viewing container output

Let’s use the kubectl logs command

We will pass either a pod name, or a type/name (E.g. if we specify a deployment or replica set, it will get the first pod in it)

Unless specified otherwise, it will only show logs of the first container in the pod (Good thing there’s only one in ours!)

View the result of our ping command:

kubectl logs deploy/pingpong

</>

Streaming logs in real time

Just like docker logs, kubectl logs supports convenient options:

-f/--follow to stream logs in real time (à la tail -f)

--tail to indicate how many lines you want to see (from the end)

--since to get logs only after a given timestamp

View the latest logs of our ping command:

kubectl logs deploy/pingpong --tail 1 --follow

</>

Scaling our application

We can create additional copies of our container (or rather our pod) with kubectl scale

Scale our pingpong deployment:

kubectl scale deploy/pingpong --replicas 8

Note: what if we tried to scale rs/pingpong-xxxx? We could! But the deployment would notice it right away, and scale back to the initial level.

</>

Resilience

The deployment pingpong watches its replica set

The replica set ensures that the right number of pods are running

What happens if pods disappear?

In a separate window, list pods, and keep watching them:

kubectl get pods -w

Ctrl-C to terminate watching.

If you wanted to destroy a pod, you would use this pattern where yyyy was the identifier of the particular pod:

kubectl delete pod pingpong-yyyy

<devAcademy>

</>

Viewing logs of multiple pods

When we specify a deployment name, only one single pod's logs are shown

We can view the logs of multiple pods by specifying a selector

A selector is a logic expression using labels

Conveniently, when you kubectl run somename, the associated objects have a run=somename label

View the last line of log from all pods with the run=pingpong label:

kubectl logs -l run=pingpong --tail 1

Unfortunately, --follow cannot (yet) be used to stream the logs from multiple containers.

</>

Clean-up

Clean up your deployment by deleting pingpong

kubectl delete deploy/pingpong

</>

Running containers with open ports

Since ping doesn’t have anything to connect to, we’ll have to run something else

Start a bunch of ElasticSearch containers:

kubectl run elastic --image=elasticsearch:2 --replicas=4

Watch them being started:

kubectl get pods -w

The -w option “watches” events happening on the specified resources.

Note: please DO NOT call the service search. It would collide with the TLD.

</>

Exposing our deployment

We'll create a default ClusterIP service

Expose the ElasticSearch HTTP API port:

kubectl expose deploy/elastic --port 9200

Look up which IP address was allocated:

kubectl get svc

</>

Services are layer 4 constructs

You can assign IP addresses to services, but they are still layer 4 (i.e. a service is not an IP address; it's an IP address + protocol + port)

This is caused by the current implementation of kube-proxy (it relies on mechanisms that don't support layer 3)

As a result: you have to indicate the port number for your service

Running services with arbitrary port (or port ranges) requires hacks (e.g. host networking mode)

</>

Testing our service

We will now send a few HTTP requests to our ElasticSearch pods

Let’s obtain the IP address that was allocated for our service, programatically:

```
IP=$(kubectl get svc elastic -o go-template --template
'{{ .spec.clusterIP }}')
```

Send a few requests:

```
curl http://$IP:9200/
```

Our requests are load balanced across multiple pods.

</>

Clean up

We’re done with the elastic deployment, so let’s clean it up

kubectl delete deploy/elastic

Outline

- Kubernetes
- Prometheus
- Kubernetes metrics & sources
- Deployment

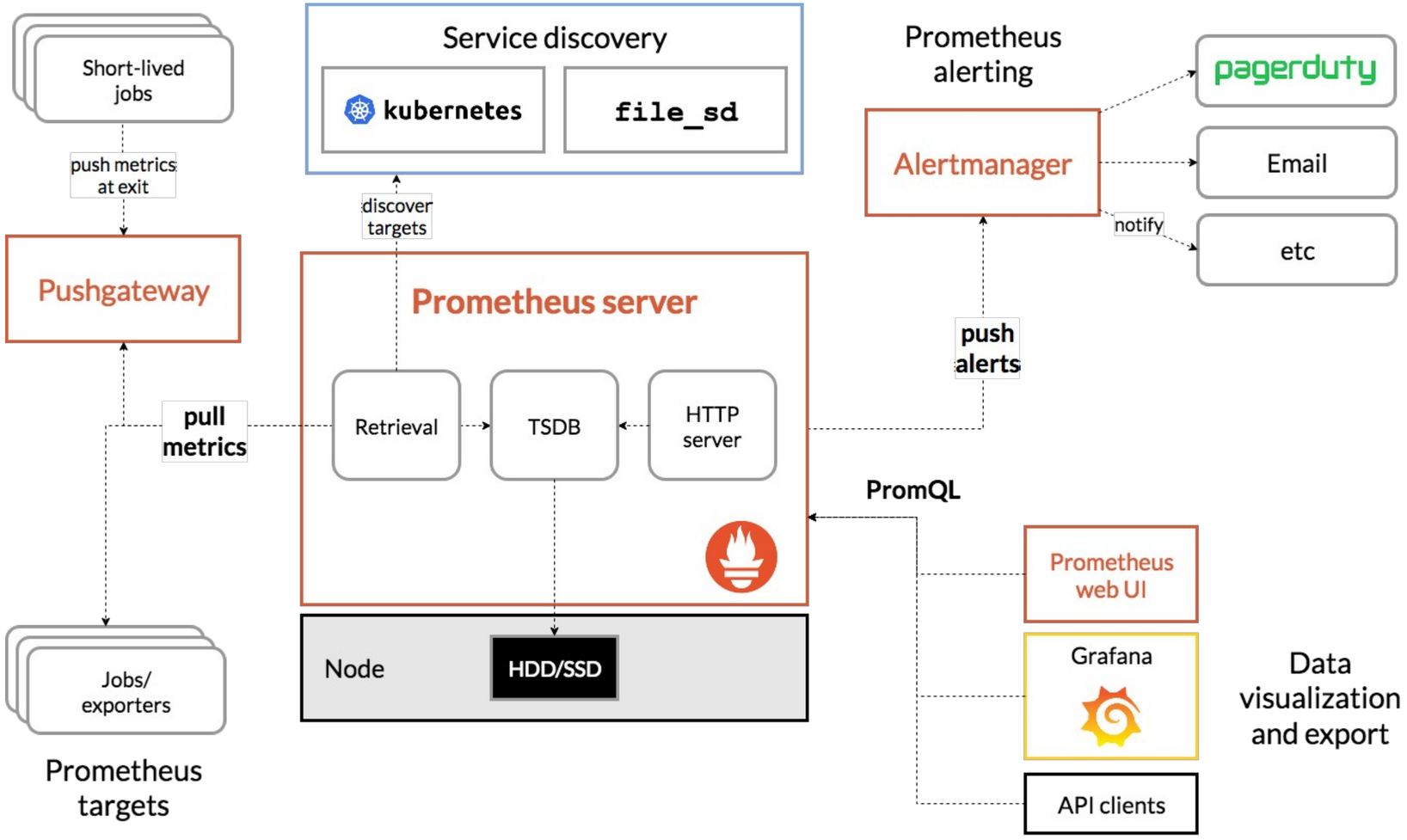
Monitor why?

- Know about outages before users tell me
- Understand my production environment (or try…)
- Plan/trend/forecast

Prometheus

- Started at SoundCloud in 2012
- Motivated by challenges with monitoring dynamic
- environments
- Made public 2015, now second CNCF “graduate

Monitoring



It’s all about the pull

- Prom scrapes targets to get metrics
- Nice side effect: know when target down
- Needs to know what to scrape

What should Prometheus scrape?

- Service discovery provides answer
- Azure, Consul, GCE, K8S, EC2, ...
- Can also watch a file containing target list

</>

Monitoring

Dimensional data model

Query:

```
http_requests_total{code="200", method="get"}
```

Metric name

Selector (aka filter)

</>

Monitoring

Dimensional data model

Query:

```
http_requests_total{code=”200”, method=”get”}
```

Response:

```
http_requests_total{code="200", method=”get”, route="/api/users"}
1528706829.115 1741

http_requests_total{code="200", method=”get”, route="/api/objects"}
1528706829.115 1920
```

Label/value pairs (aka dimensions)

<devAcademy>

Dimensional data model

Query:

```
http_requests_total{code=”200”, method=”get”}
```

Response:

```
http_requests_total{code="200", method=”get”, route="/api/users"}
1528706829.115 1741

http_requests_total{code="200", method=”get”, route="/api/objects"}
1528706829.115 1920
```

Timestamp value

Dimensional data model

- SD also provides metadata
- Metadata can be mixed in with metrics
- Powerful relabelling feature for label manipulation at ingest

Dimensional data model

Monitoring resources and methods

- For resources like memory, queues, CPUs, disks…
  - USE Method: Utilization, Saturation, Errors

    http://www.brendangregg.com/usemethod.html
- For services
  - “RED” Method: Request rate, Error rate, Duration
  - https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/

Dimensional data model

- Host metrics
  - CPU
  - Memory
  - Disk
  - Network
  - ...
- Not K8S specific, but useful as referential and for totals

Kube-static metrics

```
$ kubectl get deploy my-app -o yaml
 apiVersion: extensions/v1beta1
 kind: Deployment
 metadata:
 name: my-app
 ...
 spec:
 replicas: 4
 ...
 status:
 replicas: 4
```

</>

Monitoring

Kube-static metrics

```
$ kubectl  get deploy my-app -o yaml
  apiVersion: extensions/v1beta1
  kind: Deployment
  metadata:
    name: my-app
    ...
  spec:
    replicas: 4 <-- kube_deployment_spec_replicas{deployment="my-app", ...}
    ...
  status:
    replicas: 4 <-- kube_deployment_status_replicas{deployment="my-app", ...
```

Metrics created by kube-state-metrics
With label set from this deployment

Sample kube-state-metrics queries

Deployments with issues

```
kube_deployment_spec_replicas
!=
kube_deployment_status_replicas_available
```

Top 10 longest-running pods (“reverse uptime”)

```
topk(10, sort_desc(time() - kube_pod_created))
```

Kube core service metrics

- API Server
- etcd3
- kube-dns
- scheduler, controller-manager

</>

Monitoring

Metrics recap

	Deployment mode	How many	Metrics about
node_exporter	daemonset	1 per node	node resources
cAdvisor	inside kubelet	1 per node	container resources
kube-state-metrics	deployment	singleton	k8s object state
etcd, Api Server, controller manager, ...	core service	singleton or HA group	Itself

</>

Security

Containers themselves allows you to adopt a fundamentally different security model

- Short lived and frequently re-deployed, you can be constantly patching
- Immutable, you can control what is deployed in your environment
- With technology like gVisor, you can isolate at a sub-VM level
- Good security defaults are usually one line changes

Securing Container vs Securing VM

- Surface of attack: Minimalist host OS limits the surface of an attack
- Resource Isolation: Host resources are separated using namespaces and cgroups
- Permissions: Access controls are for app privileges and shared resources
- Lifetime: Containers have a shorter average lifetime

Thinking of Container Security

Privilege Escalation

Kubernetes API compromise

Unpatched vulnerability

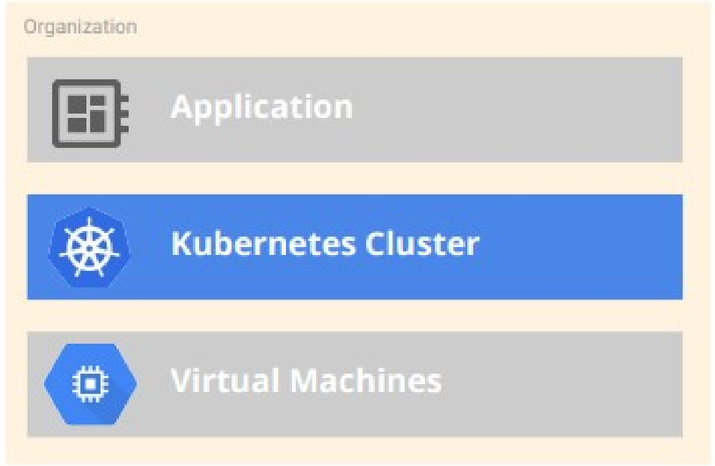
Zero day exploit in common library

DDoS

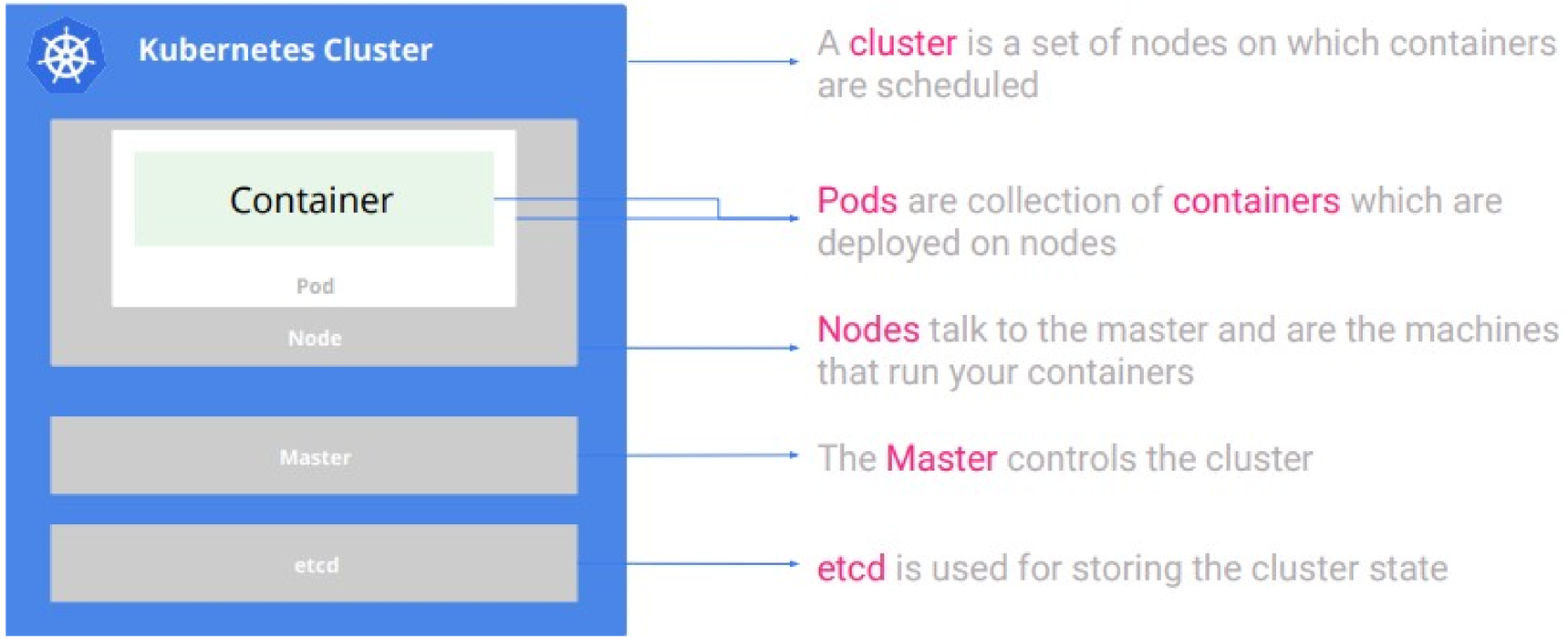
Node compromise and exploit

Thinking of Container Security

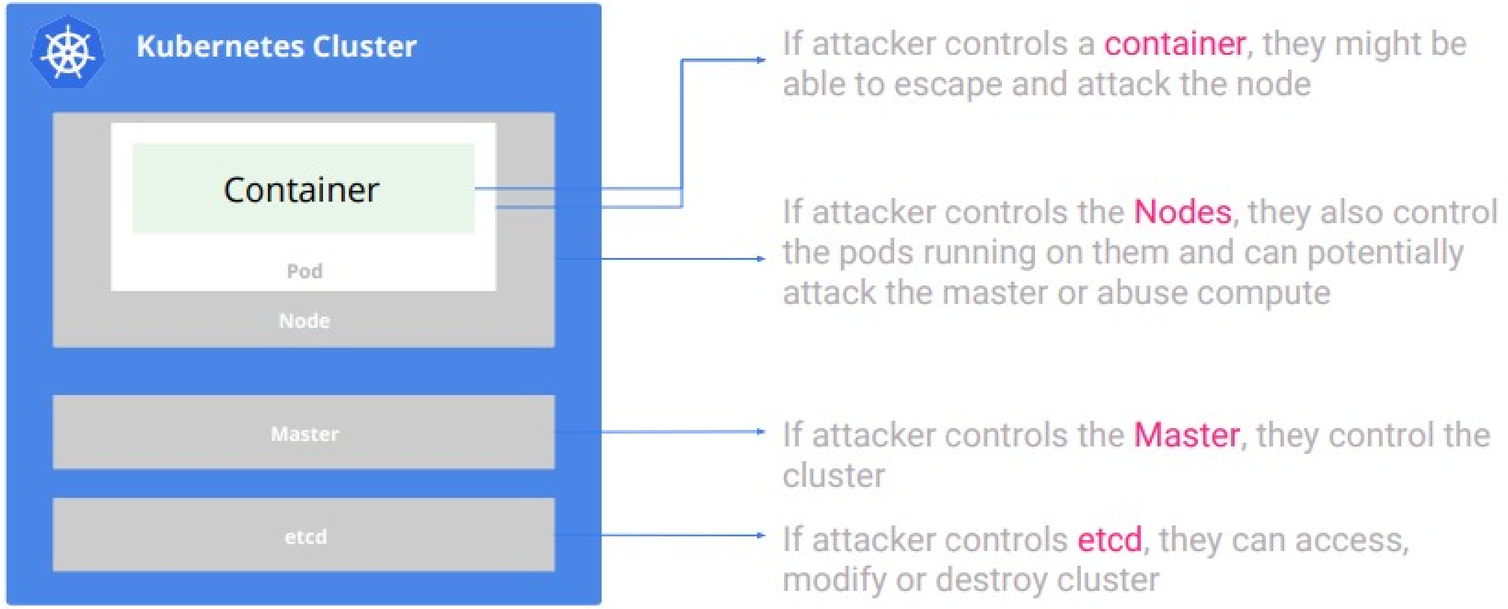
Thinking oKubernetes is only a part of your security journeyf Container Security



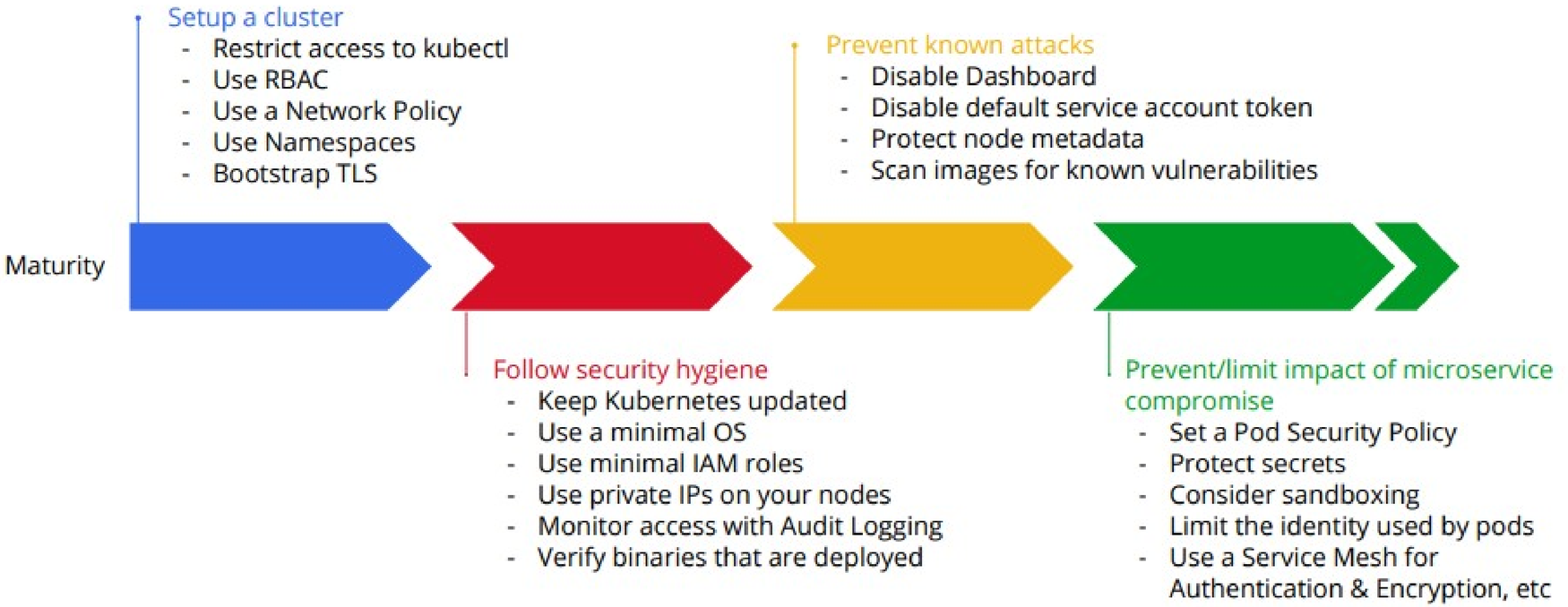
Simplified Kubernetes architecture



Simplified Kubernetes architecture



Security journey



Security journey

1. Restrict access to kubectl

Prevent unauthorized users from accessing your cluster

2. Use RBAC

Use role-based access control to define roles with rules containing a set of permissions

3. Use a Network Policy

Control pod to pod traffic

4. Protect Kube Dashboard

Either disable it or restrict access, as it uses highly privileged Kubernetes service account

5. Disable account token

Disable automatic mounting of service account token, as it can be abused by attacker

6. Use Pod Security Policy

Enable Docker seccomp and other security restrictions

Tesla Kube Dashboard was hacked

- Hackers accessed the K8s Console, which was not password protected and discovered on public internet
- Console contained privileged cloud account credentials
- Used credentials to access resources and mine cryptocurrency

http://fortune.com/2018/02/20/tesla-hack-amazon-cloud-cryptocurrency-mining/

- Obviously, the solution here is to have literally any kind of security

Fixing it:

- Don’t deploy it at all
- Reduce its permissions
- Reduce its permissions and add authentication

Dashboards do not have admin access in GKE 1.7+ clusters

- Disabled by default in GKE 1.10+ clusters
- Use Google Cloud Platform Console for the same (and more) things you would have used the K8s Dashboard for

```
# Updating existing cluster
$ gcloud container clusters update demo-cluster \
--update-addons=KubernetesDashboard=DISABLED
```

- Every namespace has a default (Kubernetes) service account
- Pods use service accounts to assert their identity to other workloads, including to the API server
- When you create a pod, if you do not specify a service account, the pod

will assign to default service account for that NS.

- The pod mounts these service account credentials, which are authorized to talk to the API server, if pod is compromised, these credentials can be used to perform arbitrary operations.

</>

GCP – Hands On

To launch Cloud Shell, perform the following steps:

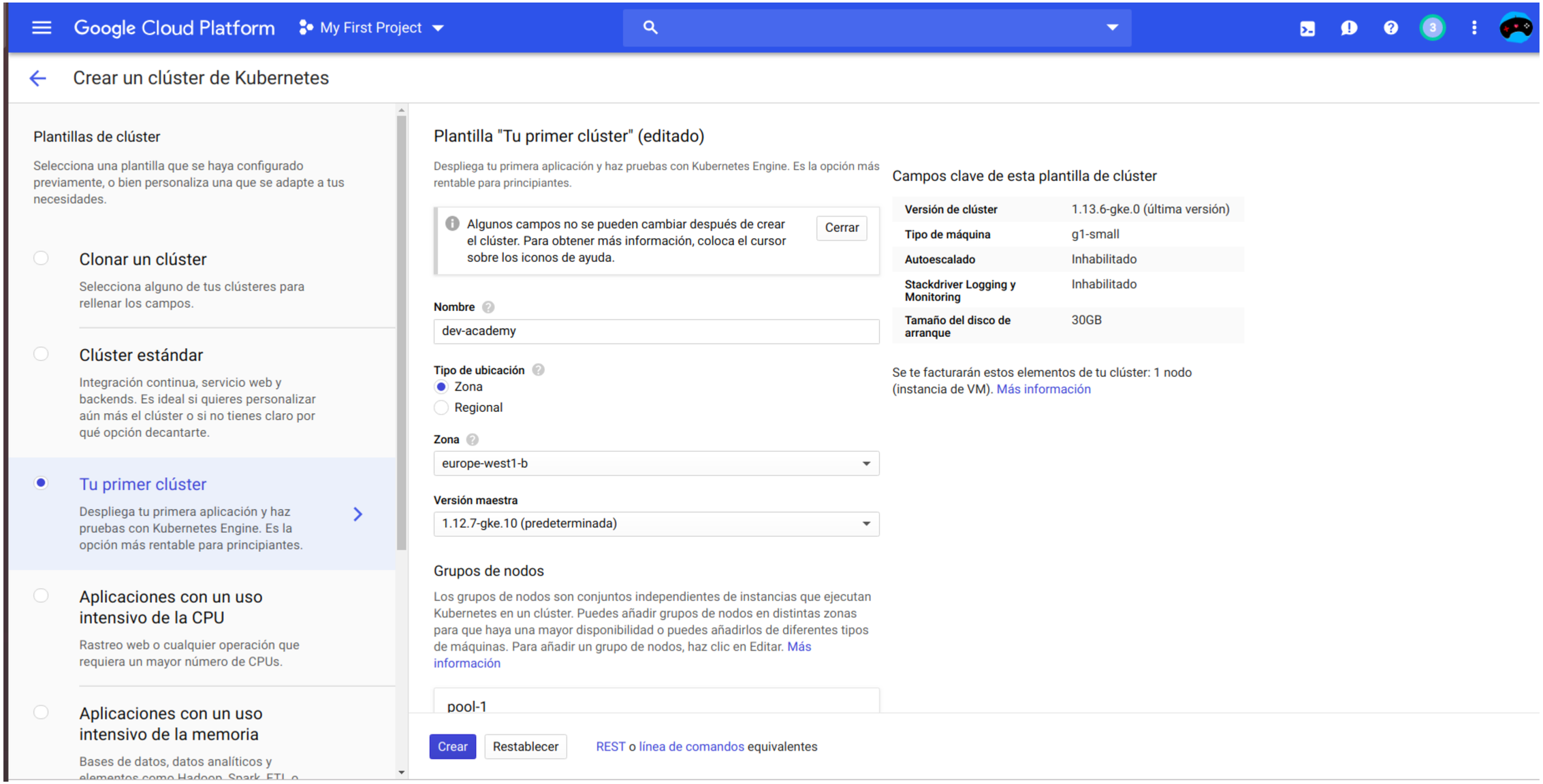
Go to Google Cloud Platform Console.

Google Cloud Platform Console

From the top-right corner of the console, click the Activate Google Cloud Shell button:

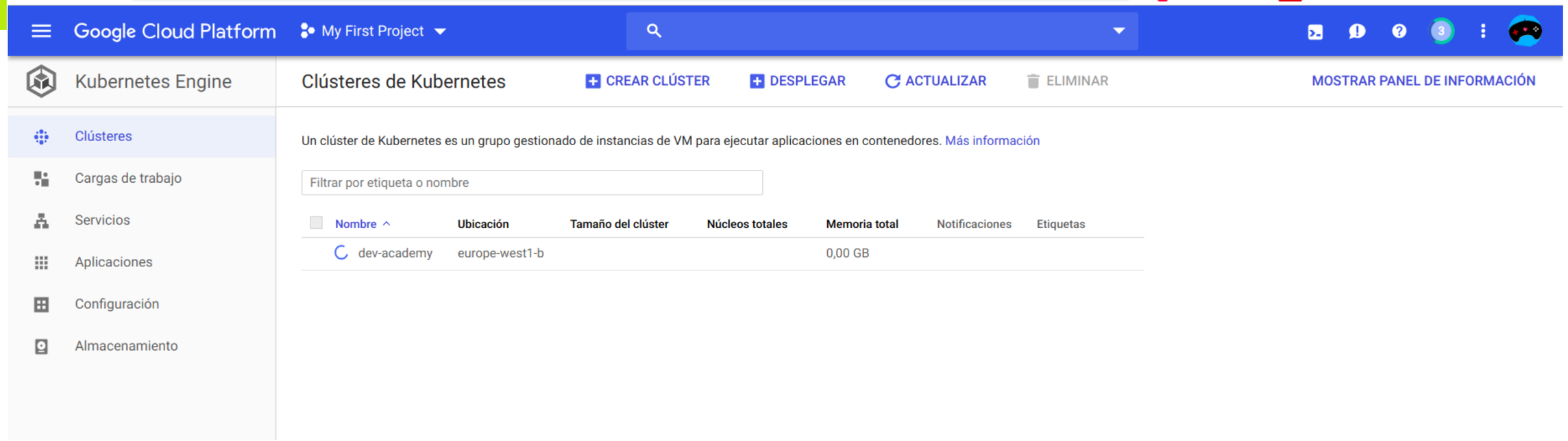
A Cloud Shell session opens inside a frame at the bottom of the console. You use this shell to run gcloud and kubectl commands.

</>


Google Cloud Platform
My First Project
Crear un clúster de Kubernetes
Plantillas de clúster
Selecciona una plantilla que se haya configurado previamente, o bien personaliza una que se adapte a tus necesidades.
Clonar un clúster
Selecciona alguno de tus clústeres para rellenar los campos.
Clúster estándar
Integración continua, servicio web y backends. Es ideal si quieres personalizar aún más el clúster o si no tienes claro por qué opción decantarte.
Tu primer clúster
Despliega tu primera aplicación y haz pruebas con Kubernetes Engine. Es la opción más rentable para principiantes.
Aplicaciones con un uso intensivo de la CPU
Rastreo web o cualquier operación que requiera un mayor número de CPUs.
Aplicaciones con un uso intensivo de la memoria
Bases de datos, datos analíticos y
Plantilla "Tu primer clúster" (editado)
Despliega tu primera aplicación y haz pruebas con Kubernetes Engine. Es la opción más rentable para principiantes.
Algunos campos no se pueden cambiar después de crear el clúster. Para obtener más información, coloca el cursor sobre los iconos de ayuda.
Cerrar
Nombre
dev-academy
Tipo de ubicación
Zona
Regional
Zona
europe-west1-b
Versión maestra
1.12.7-gke.10 (predeterminada)
Grupos de nodos
Los grupos de nodos son conjuntos independientes de instancias que ejecutan Kubernetes en un clúster. Puedes añadir grupos de nodos en distintas zonas para que haya una mayor disponibilidad o puedes añadirlos de diferentes tipos de máquinas. Para añadir un grupo de nodos, haz clic en Editar. Más información
pool-1
Campos clave de esta plantilla de clúster
Versión de clúster
1.13.6-gke.0 (última versión)
Tipo de máquina
g1-small
Autoescalado
Inhabilitado
Stackdriver Logging y Monitoring
Inhabilitado
Tamaño del disco de arranque
30GB
Se te facturarán estos elementos de tu clúster: 1 nodo (instancia de VM). Más información
Crear
Restablecer
REST o línea de comandos equivalentes

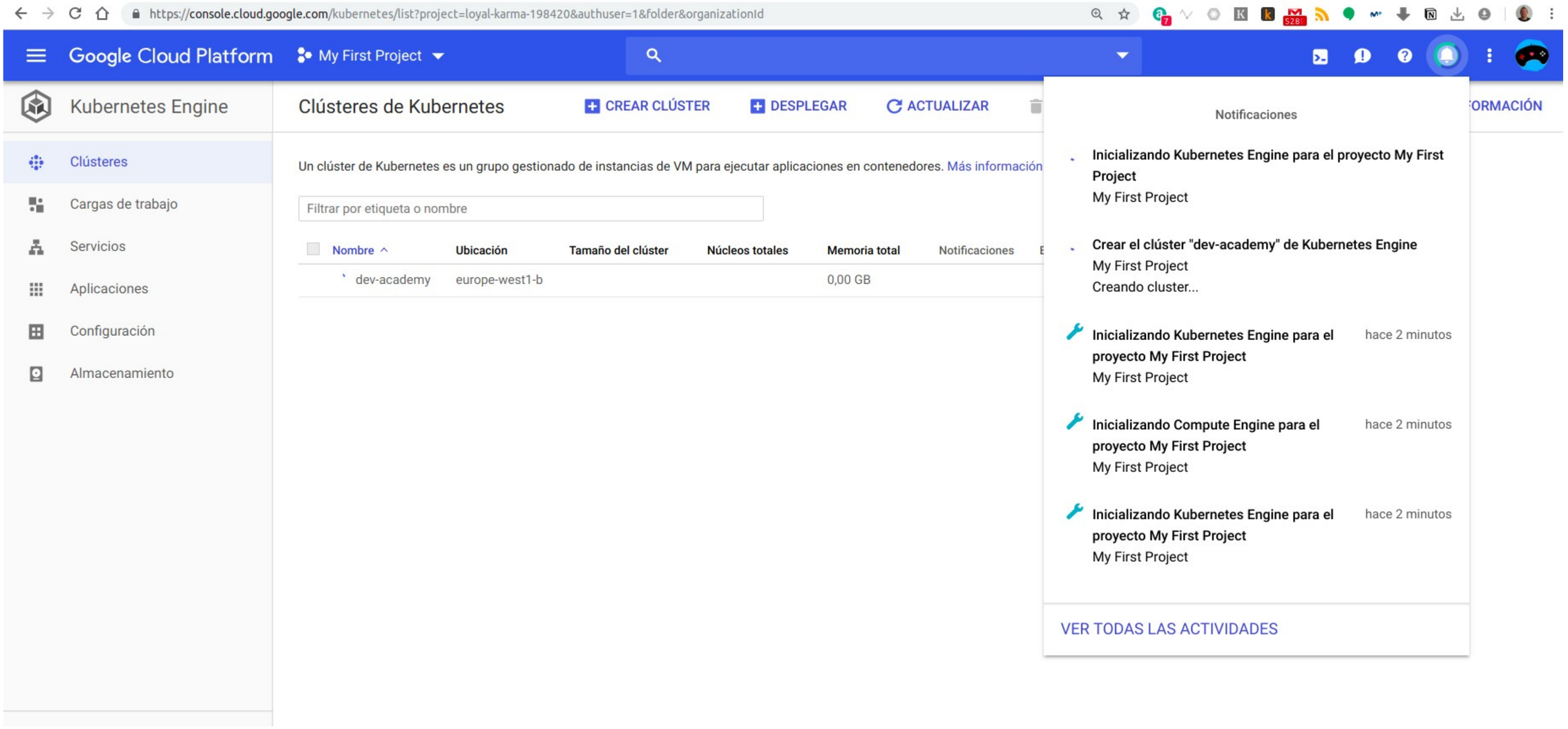
<devAcademy>

</>


Google Cloud Platform
My First Project
Kubernetes Engine
Clústeres de Kubernetes
CREAR CLÚSTER
DESPLEGAR
ACTUALIZAR
ELIMINAR
MOSTRAR PANEL DE INFORMACIÓN
Clústeres
Cargas de trabajo
Servicios
Aplicaciones
Configuración
Almacenamiento
Un clúster de Kubernetes es un grupo gestionado de instancias de VM para ejecutar aplicaciones en contenedores. Más información
Filtrar por etiqueta o nombre
Nombre
Ubicación
Tamaño del clúster
Núcleos totales
Memoria total
Notificaciones
Etiquetas
dev-academy
europe-west1-b
0,00 GB

<devAcademy>

</>


https://console.cloud.google.com/kubernetes/list?project=loyal-karma-198420&authuser=1&folder&organizationId
Google Cloud Platform
My First Project
Kubernetes Engine
Clústeres de Kubernetes
CREAR CLÚSTER
DESPLEGAR
ACTUALIZAR
Clústeres
Cargas de trabajo
Servicios
Aplicaciones
Configuración
Almacenamiento
Un clúster de Kubernetes es un grupo gestionado de instancias de VM para ejecutar aplicaciones en contenedores. Más información
Filtrar por etiqueta o nombre
Nombre
Ubicación
Tamaño del clúster
Núcleos totales
Memoria total
Notificaciones
dev-academy
europe-west1-b
0,00 GB
Notificaciones
Inicializando Kubernetes Engine para el proyecto My First Project
My First Project
Crear el clúster "dev-academy" de Kubernetes Engine
My First Project
Creando cluster...
Inicializando Kubernetes Engine para el proyecto My First Project
hace 2 minutos
My First Project
Inicializando Compute Engine para el proyecto My First Project
hace 2 minutos
My First Project
Inicializando Kubernetes Engine para el proyecto My First Project
hace 2 minutos
My First Project
VER TODAS LAS ACTIVIDADES

<devAcademy>

The company: Minecraft Gamification

</>

Introduction

Minecraft Gamification is a video game business dedicated to Minecraft Servers.

Has presence in Europe.

About 100 employees. 30 are IT people.

Their open servers received a lot of requests for Minecraft Clients (The game is in the top 10 sellers of the history)

</>

The services

<devAcademy>

The services

</>

Minecraft Server

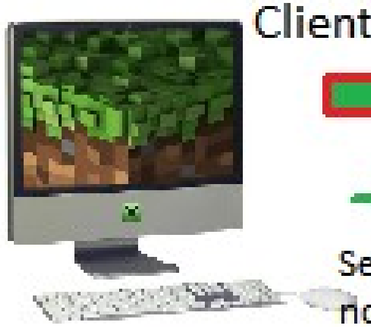
Server and Client are a java application and are already compiled and stored in a Github repository.

<devAcademy>

</>

The architecture

<devAcademy>



Client connects to server, requests a plugin list

Server sends plugin list, client checks list and requests plugins not already installed to client to be sent for download

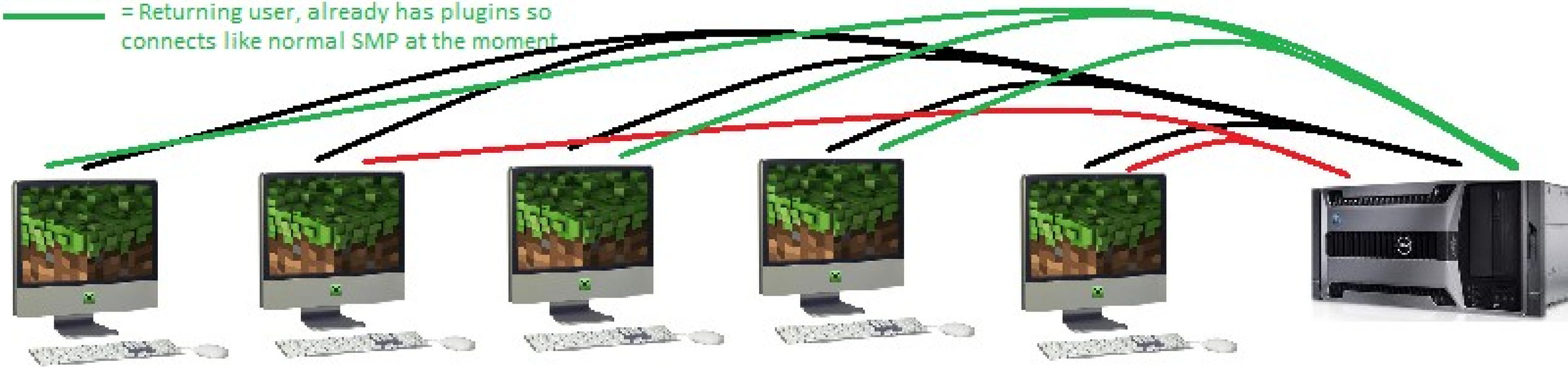
Client downloads all needed plugins from server, then

Minecraft Server

= First time connection, requests plugin list etc.

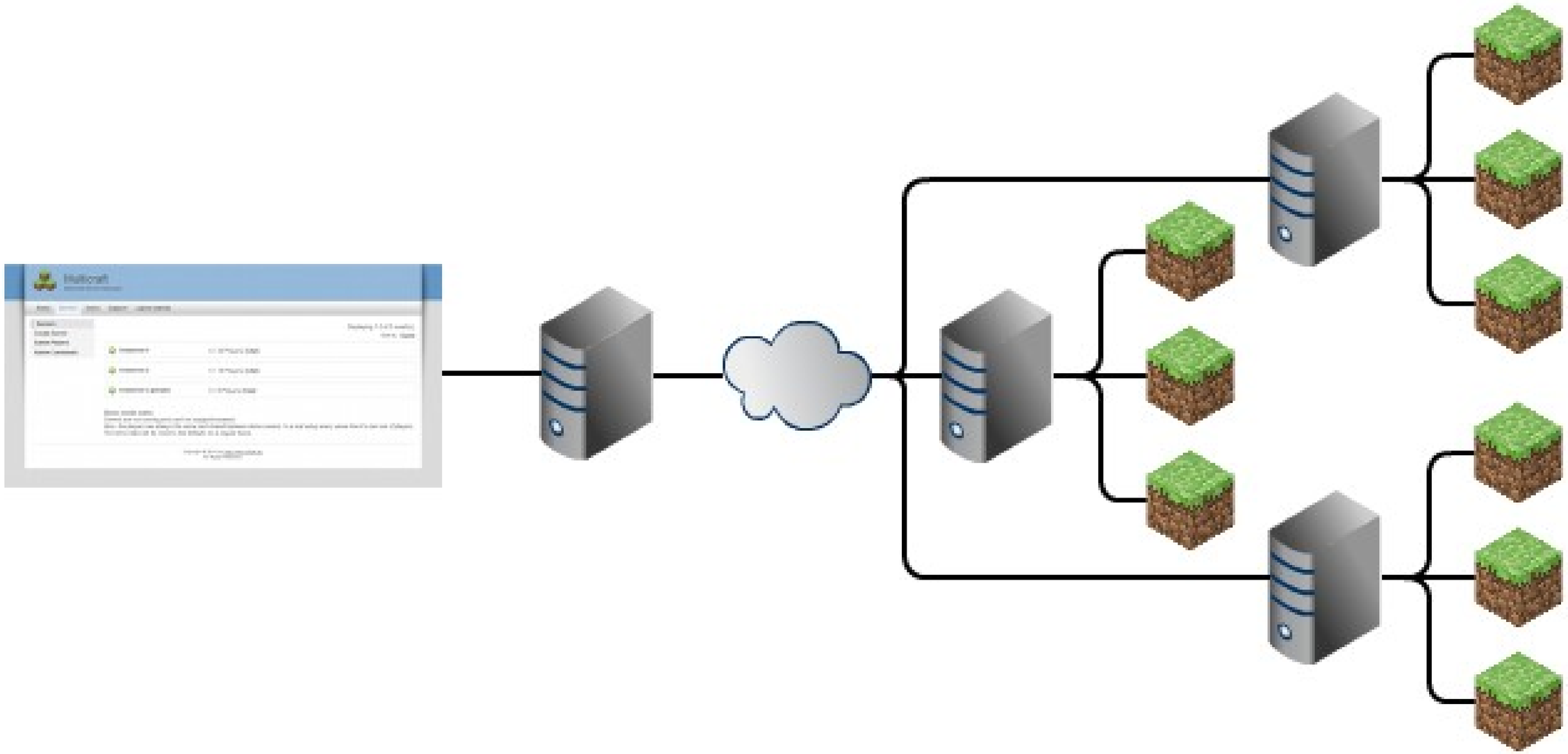
= Normal client/server connection

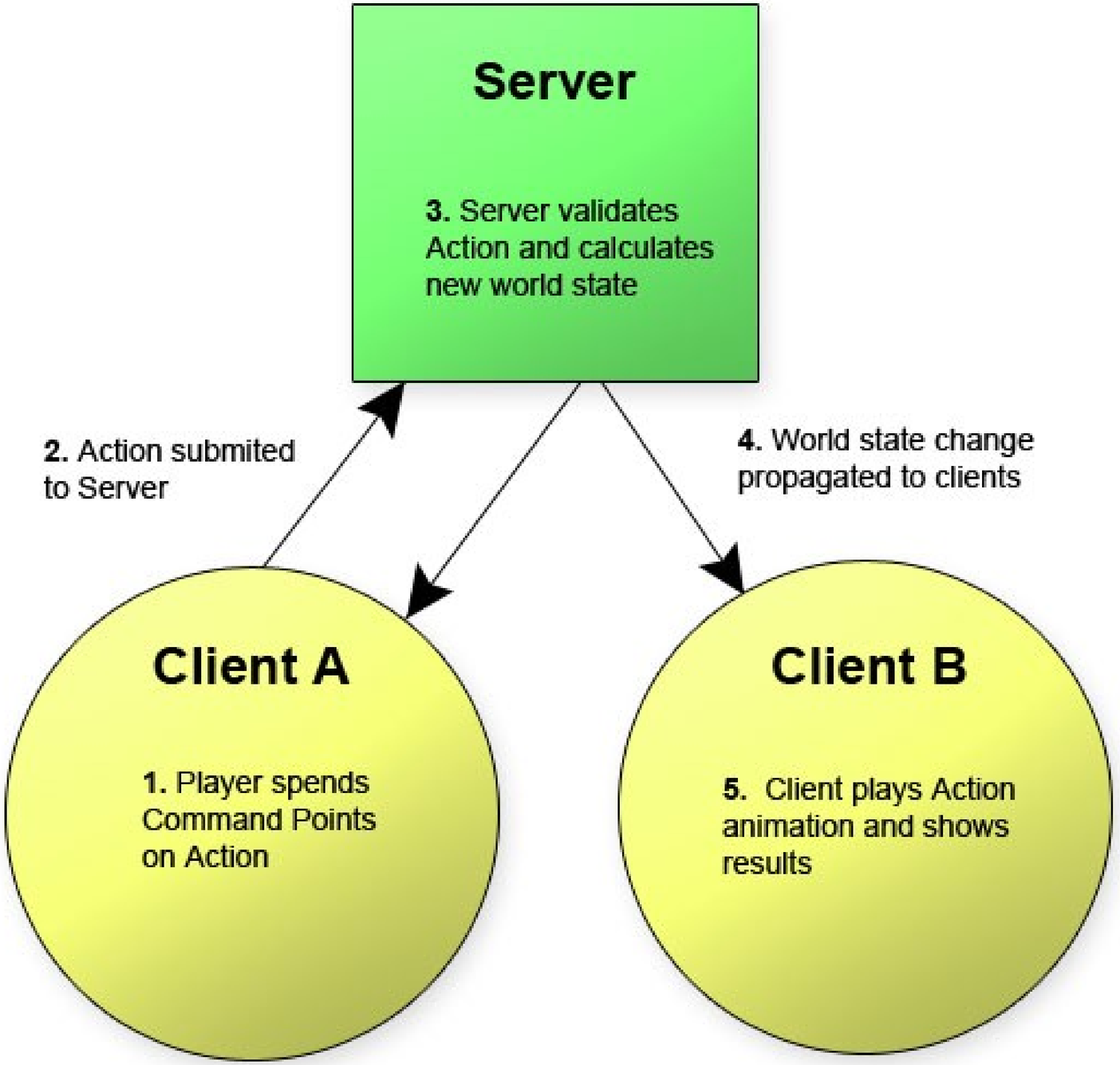
= Returning user, already has plugins so connects like normal SMP at the moment



<devAcademy>

</>




Server
3. Server validates Action and calculates new world state
2. Action submited to Server
4. World state change propagated to clients
Client A
1. Player spends Command Points on Action
Client B
5. Client plays Action animation and shows results

</>

References

https://github.com/juanviz/devacademyDocker

https://github.com/juanviz/CursoDevops1

https://kubernetes.io/

https://kubernetes.io/docs/home/

https://tryk8s.com/

<devAcademy>

Juan Vicente Herrera Ruiz de Alejo
Email de contacto: *juan.vicente.herrera@gmail.com*

info@devacademy.es

687374918

@DevAcademyES