Cloud-Native Applications and Kubernetes* (k8s)

Alex Glikson

15-719, S19
February 25th, 2019

*Trivia: Κυβερνήτης – steersman, a person who steers a ship

https://en.wiktionary.org/wiki/%CE%BA%CF%85%CE%B2%CE%B5%CF%81%CE%BD%CE%AE%CF%84%CE%B7%CF%82

Icons made by Freepik from www.flaticon.com
# Outline

<table>
<thead>
<tr>
<th>Background</th>
<th>• Cloud-Native Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>K8s Overview</td>
<td>• K8s In a Nutshell</td>
</tr>
<tr>
<td></td>
<td>• Architecture</td>
</tr>
<tr>
<td>Applications</td>
<td>• Unit of Deployment: Pod</td>
</tr>
<tr>
<td></td>
<td>• Composite Applications</td>
</tr>
<tr>
<td>Control Plane</td>
<td>• Scheduler</td>
</tr>
<tr>
<td></td>
<td>• Custom Resources and Controllers</td>
</tr>
<tr>
<td>Summary</td>
<td>• Related Projects</td>
</tr>
<tr>
<td></td>
<td>• Main Takeaways</td>
</tr>
</tbody>
</table>
Cloud-Native Applications: Motivation

- **Elasticity** and **ubiquity** of cloud infrastructure enabled new generation of applications, with a potential to disrupt their markets, or to create new markets, e.g.:
  - Netflix
  - Airbnb
  - Spotify
  - Pinterest
  - Snapchat
  - Whatsapp
Cloud-Native Applications: Example, Requirements

• Example: Netflix

  o Value proposition (competitive advantage):
    • Low cost video streaming with superb user experience, at scale

  o Application properties and unique requirements:
    • >100 millions of users in 190 countries (most of them in the US)
      – Vast variance in load, within minutes (evenings, campaigns, etc)
    • 10s of thousands of servers
      – At least one server will fail every day
    • 1000s of daily application changes, across 100s of functions
      – Video streaming, catalog, recommendations, subscription, etc
      – ~1 update every minute
Cloud-Native Applications: Requirements → Design Principles

- Low Cost + Variance in Load (+User Experience)
  - Can’t afford over- or under-provisioning → **Auto-Scaling**
- Hardware Failures + High Scale (+User Experience)
  - Accommodate HW failures without downtime → **Design for Failure**
- Frequent Application Updates + High Scale (+User Experience)
  - Can’t afford redeploying everything every time → **Modularity**
  - Can’t afford testing everything every time → **Stable Internal APIs**
- Frequent Application Updates + Low Cost (+User Experience)
  - Can’t afford manual QA/admin effort for each update → **Automation**
Cloud–Native Applications: Design → Common Services

<table>
<thead>
<tr>
<th>Design Principle</th>
<th>Common Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto-Scaling</td>
<td>Horizontal Auto-Scaling</td>
</tr>
<tr>
<td></td>
<td>Elastic Load Balancing</td>
</tr>
<tr>
<td>Design for Failure</td>
<td>Replication</td>
</tr>
<tr>
<td></td>
<td>Health Monitoring</td>
</tr>
<tr>
<td>Modularity</td>
<td>Decoupling into homogenous (micro)services*</td>
</tr>
<tr>
<td></td>
<td>Unified packaging (across dev/test/prod) with Docker</td>
</tr>
<tr>
<td>API-driven composition</td>
<td>Discovery</td>
</tr>
<tr>
<td></td>
<td>Routing</td>
</tr>
<tr>
<td>Automation</td>
<td>Fully programmable life cycle of components*</td>
</tr>
<tr>
<td></td>
<td>Observability (monitoring, tracing, etc)</td>
</tr>
</tbody>
</table>

* not a service
Cloud–Native Applications: Platforms

• New platforms emerged, offering common services (features) that make it easier to develop cloud-native applications
  o Auto-scaling, replication, load balancing, health monitoring, service discovery, application-level routing, programmability
  o Commonly referred as ‘Platform as a Service’ (PaaS)
## Outline

<table>
<thead>
<tr>
<th>Background</th>
<th>Cloud-Native Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>K8s Overview</strong></td>
<td>✓</td>
</tr>
<tr>
<td>• K8s In a Nutshell</td>
<td></td>
</tr>
<tr>
<td>• Architecture</td>
<td></td>
</tr>
<tr>
<td><strong>Applications</strong></td>
<td></td>
</tr>
<tr>
<td>• Unit of Deployment: Pod</td>
<td></td>
</tr>
<tr>
<td>• Composite Applications</td>
<td></td>
</tr>
<tr>
<td><strong>Control Plane</strong></td>
<td></td>
</tr>
<tr>
<td>• Scheduler</td>
<td></td>
</tr>
<tr>
<td>• Custom Resources and Controllers</td>
<td></td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td></td>
</tr>
<tr>
<td>• Related Projects</td>
<td></td>
</tr>
<tr>
<td>• Main Takeaways</td>
<td></td>
</tr>
</tbody>
</table>
Kubernetes (k8s):
Open-Source, Cloud-Native & Container-Based Platform

- Originally developed by Google, based on Borg and Omega ideas
- Available to all on github.com
- As of 02’2019: ~2000 contributors

- Common Services
- Programmable
- Extensible

- Monitoring
- Logging
- HA
- Scaling
- Updates

- Auto-Scaling
- Design for Failure
- Modularity, APIs
- Automation

Open source platform for deployment and management of arbitrary container-based, cloud-native applications on clusters of servers

- Specification
- Provisioning
- Packaging and runtime isolation using Docker
- Physical or virtual, public or private cloud (e.g., EC2, OpenStack)
Kubernetes in a Nutshell

https://www.youtube.com/embed/4ht22ReBjno?start=0&end=220&autoplay=1
Kubernetes in a Nutshell
Kubernetes Architecture

Master Node

- API Server
- DB Proxy
- Scheduler
- Controllers

Worker Node

- kubelet
- Docker
- Pod

Inspired by: https://x-team.com/blog/introduction-kubernetes-architecture/
Pod: One or More Containers + Resources + Labels

Why Multi-Container Pods?

- **Scaling/replication unit**: runtime components that must run together
- **Sidecar pattern**: “secondary” functions deployed in separate containers
  - e.g., web content synchronization
- **Ambassador/Proxy pattern**: simplify access to an external system
  - e.g., key-value store
- **Adapter pattern**: simplify access from an external system
  - e.g., monitoring

Source: [https://www.slideshare.net/ssuser6bb12d/kubernetes-introduction-71846110](https://www.slideshare.net/ssuser6bb12d/kubernetes-introduction-71846110)

Source: [https://www.usenix.org/node/196347](https://www.usenix.org/node/196347)
Event-Driven Architecture in k8s: Pod Creation

Source: [https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca](https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca)
# Outline

<table>
<thead>
<tr>
<th>Background</th>
<th>✓ Cloud-Native Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>K8s Overview</td>
<td>✓ K8s in a Nutshell</td>
</tr>
<tr>
<td></td>
<td>✓ Architecture</td>
</tr>
<tr>
<td>Applications</td>
<td>✓ Unit of Deployment: Pod</td>
</tr>
<tr>
<td></td>
<td>• Composite Applications</td>
</tr>
<tr>
<td>Control Plane</td>
<td>• Scheduler</td>
</tr>
<tr>
<td></td>
<td>• Custom Resources and Controllers</td>
</tr>
<tr>
<td>Summary</td>
<td>• Related Projects</td>
</tr>
<tr>
<td></td>
<td>• Main Takeaways</td>
</tr>
</tbody>
</table>
From Pods to Applications: Deployment, Replica Set

- **Goal:** Maintain a given number of replica's of a given Pod
  - HA, load balancing
- **Example:** NGiNX web server

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      type: my-nginx-pod
  template:
    metadata:
      labels:
        type: my-nginx-pod
    spec:
      containers:
      - name: my-nginx-container
        image: nginx
        ports:
        - containerPort: 80
```
Composite Application Types in Kubernetes

- **Deployment (Replica Set)**
  - Maintains a given number of replica’s of a given Pod
    - High availability, load balancing
- **Daemon Set**
  - Runs exactly one instance of a given Pod on every node
    - Access to unique host resources, load balancing
- **Stateful Set**
  - Maintains a scalable collection of Pods with unique roles
    - Persistent resource and identities
- **Job**
  - Enforces policies for execution and completion of groups of tasks (e.g., MPI)
    - Controlled parallelism, work queues, schedule-based execution, etc
- **Custom**
  - Your own
## Outline

<table>
<thead>
<tr>
<th>Background</th>
<th>✓ Cloud-Native Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>K8s Overview</td>
<td>✓ K8s in a Nutshell</td>
</tr>
<tr>
<td></td>
<td>✓ Architecture</td>
</tr>
<tr>
<td>Applications</td>
<td>✓ Unit of Deployment: Pod</td>
</tr>
<tr>
<td></td>
<td>✓ Composite Applications</td>
</tr>
<tr>
<td>Control Plane</td>
<td>• Scheduler</td>
</tr>
<tr>
<td></td>
<td>• Custom Resources and Controllers</td>
</tr>
<tr>
<td>Summary</td>
<td>• Related Projects</td>
</tr>
<tr>
<td></td>
<td>• Main Takeaways</td>
</tr>
</tbody>
</table>
Kubernetes Default Scheduler

• Main goal: placement of Pods on Nodes
  o Triggered primarily by watching for new Pods in ‘pending’ state
• FIFO scheduling
  o Supports priority-based arbitration and eviction
• Emphasis on scale and speed
  o No complex logic (e.g., superlinear in number of nodes)
Kubernetes Default Scheduler: Placement Policies

• Filter() + Prioritize()

• Filtering:
  o Configurable set of node **predicates**
  o A node must pass all predicates to qualify for prioritization

• Prioritization:
  o Optimization goals expressed as **utility functions**
  o Prefer nodes with maximal utility
    • Can target multiple optimization goals with weights
Placement Policies: Built-In Predicates

- **PodFitsResources**
  - CPU/memory allocation
- **PodMatchNodeSelector**
  - Labels match - e.g., specific HW capabilities (e.g., SSD storage)
- **MatchInterPodAffinity**
  - Same/Different node as given (running) Pod(s)

Node 1
- 8 CPUs
- 16 GB mem

Node 2
- 8 CPUs
- 32 GB mem

Node 3
- 8 CPUs
- 32 GB mem
  - disk=SSD

- **NodeSelector:** disk=SSD
- **Anti-Affinity:** app=DB
Placement Policies: Built-In Priority Functions

- **InterPodAffinity**
  - Maximal affinity with given Pods
- **LeastResourceAllocation**
  - Maximal spreading
- **MostResourceAllocation**
  - Maximal packing
- **BalancedResourceAllocation**
  - Minimal resource fragmentation
- **ImageLocality**
  - Minimal startup time

Node 1
- 8 CPUs
- 16 GB mem

Node 2
- 8 CPUs
- 32 GB mem

Node 3
- 8 CPUs
- 32 GB mem
disk=SSD
Placement Policies: Built-In Predicates and Priorities

**Predicates***
- PodFitsResources
  - CPU/memory allocation
- PodMatchNodeSelector
  - Labels match - e.g., specific HW capabilities (e.g., SSD storage)
- MatchInterPodAffinity
  - Same node as given (running) Pod(s)

**Priorities***
- InterPodAffinity
  - Maximal affinity with given Pods
- LeastResourceAllocation
  - Maximal spreading
- MostResourceAllocation
  - Maximal packing
- BalancedResourceAllocation
  - Minimal resource fragmentation
- ImageLocality
  - Minimal startup time

(*) Partial List. Can also implement custom predicates/prioritizers
Placement Policies: Affinity / Anti-Affinity Examples

- Affinity = same host/rack/zone/etc, Anti-affinity = different

- Example 1:
  - If a host fails, at most one pod out of N is down (high availability)
    - Replica Set with N replicas, host-level anti-affinity

- Example 2:
  - Maintain one pod in each region, to reduce latency to clients
    - Daemon Set with region-level anti-affinity

- Example 3:
  - Keep all tasks of a job in the same rack, for better performance
    - Job with rack-level affinity
Affinity/Anti-Affinity Example: Web Server + Redis

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      containers:
        - name: redis-server
          image: redis:3.2-alpine

apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  selector:
    matchLabels:
      app: web-store
  replicas: 3
  template:
    metadata:
      labels:
        app: web-store
    spec:
      containers:
        - name: web-app
          image: nginx:1.12-alpine
```

![Node Placement Diagram](image)
Placement Policies – Example: Web Server + Redis

Different hosts

Same host

Bug!!

If respective Redis server is killed, the web server is not re-deployed
## Outline

<table>
<thead>
<tr>
<th>Background</th>
<th>✓ Cloud-Native Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>K8s Overview</td>
<td>✓ K8s in a Nutshell</td>
</tr>
<tr>
<td></td>
<td>✓ Architecture</td>
</tr>
<tr>
<td>Applications</td>
<td>✓ Unit of Deployment: Pod</td>
</tr>
<tr>
<td></td>
<td>✓ Composite Applications</td>
</tr>
<tr>
<td>Control Plane</td>
<td>✓ Scheduler</td>
</tr>
<tr>
<td></td>
<td>• Custom Resources and Controllers</td>
</tr>
<tr>
<td>Summary</td>
<td>• Related Projects</td>
</tr>
<tr>
<td></td>
<td>• Main Takeaways</td>
</tr>
</tbody>
</table>
Extending Kubernetes with Custom Resources

- Example: Nuclio serverless function as a custom resource type in k8s

```yaml
apiVersion: "nuclio.io/v1beta1"

kind: "NuclioFunction"

spec:
  runtime: "golang"
  handler: main:Handler
  minReplicas: 1
  maxReplicas: 1
  triggers:
    test_rmq:
      kind: "rabbit-mq"
      url: "amqp://user:password@rabbitmq"
      attributes:
        exchangeName: "exchange-name"
        queueName: "queue-name"
```

- Under the covers, Nuclio Controller maintains K8s Pods for each function

```yaml
apiVersion: apiextensions.k8s.io/v1beta1

kind: CustomResourceDefinition

metadata:
  name: nucliofunctions.nuclio.io

spec:
  group: nuclio.io
  names:
    kind: NuclioFunction
    plural: nucliofunctions
    singular: nucliofunction
  scope: Namespaced
  version: v1beta1
```

Source: [https://github.com/nuclio/nuclio](https://github.com/nuclio/nuclio)
Extending Kubernetes with Custom Resources
Example: Life Cycle of NuclioFunction Resources*

- Cluster initialization [Administrator]
  - Register NuclioFunction CRD
  - Deploy Nuclio Controller
  - Subscribe Nuclio Controller to notifications for NuclioFunction (and related) events

- Function deployment [Application Owner]
  - NuclioFunction resource is created
  - Nuclio Controller is triggered by the API server
  - Nuclio Controller creates necessary K8s resources (Pods, routes, etc)

- Function invocation [Application User]
  - Trigger is invoked (e.g., http request)
  - Scaling logic is invoked (if necessary)
  - Payload is forwarded to the Pod selected to run the function invocation

* Conceptual workflow
# Outline

<table>
<thead>
<tr>
<th>Background</th>
<th>✓ Cloud-Native Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>K8s Overview</td>
<td>✓ K8s in a Nutshell</td>
</tr>
<tr>
<td></td>
<td>✓ Architecture</td>
</tr>
<tr>
<td>Applications</td>
<td>✓ Unit of Deployment: Pod</td>
</tr>
<tr>
<td></td>
<td>✓ Composite Applications</td>
</tr>
<tr>
<td>Control Plane</td>
<td>✓ Scheduler</td>
</tr>
<tr>
<td></td>
<td>✓ Controllers</td>
</tr>
</tbody>
</table>

## Summary

- Related Projects
- Main Takeaways
Related Project (Partial List)

• Virtual Kubelet
  o Proxy to another container management platform
• kube-batch aka kube-arbitrator
  o Gang scheduling (e.g., MPI), non-FIFO scheduling
• Knative
  o Serverless web applications on Kubernetes
• De-scheduler
  o Eviction of pods – e.g., to reduce resource fragmentation
• Rapidly growing ecosystem of projects governed by CNCF
  o [https://www.cncf.io/](https://www.cncf.io/)
Summary: Main Takeaways

1. Docker + Cloud-Native ➔ K8s
   - Container-native platform targeting arbitrary applications

2. Pod = collection of containers
   - Composable into Replica Sets, Daemon Sets, Jobs, Your Own Pattern

3. K8s API server provides CRUD and watchers (“events”)
   - Triggering a set of controllers that orchestrate the life cycle of resources

4. K8s has a powerful, flexible and extensible Scheduler

5. K8s is fun!
QUESTIONS?
Trends: Linux Containers

- Linux features providing isolation within a single OS:
  - Hardware resources: cgroups
    - CPU, RAM, I/O devices, etc
  - OS resources: namespaces, e.g.:
    - USER, PID, NET, MNT

- De-facto standard to efficiently run services/jobs at scale
  - Enable workload consolidation, infrastructure elasticity
  - Pioneered by Google (Linux features, massive internal use*)
    - 2 billion containers per week! (>3000 per second on average)
Trends: Linux Containers – Implementations

• Example (open source) implementations:
  o LXC, Cloud Foundry, Mesos, YARN, ...

• Docker (Moby*)
  o Targeting arbitrary applications, e.g. by offering:
    • Flexible packaging
    • Seamless inter-container network communication
    • Configurable cluster-wide storage/network (SDN, NAS, etc)
  o Developer-friendly
    • Lightweight standalone engine
    • Intuitive git-style image management
    • Easy sharing and reuse of images - Docker Hub
Trends: Linux Containers – Popular Docker Images

Operating System  Web Server  Data Processing  Programming Language

---

February 25, 2019
CMU CS 15719 Advanced Cloud Computing S19
Cloud-Native PaaS + Docker = Container-native Platforms (e.g., Kubernetes)

- 2002: ns
- 2004: UnionFS
- 2006: cgroups
- 2008: LXC

Pre-history:
- 1979: chroot
- 1999: VMware
- 2003: Xen

2004:
- EC2
- Heroku
- GAE
- Azure

2006:
- Cloud Foundry
- AWS Beanstalk

2010:
- Kubernetes

2011:
- YARN
- Mesos

2012:
- Omega

2013:
- Docker

2017:
- moby containerd

2018:
- Kubernetes
Surfacing Applications to Users: Service

- Goal: Maintain a given number of replica’s of a given Pod
  - HA, load balancing
- Example: NGiNX web server

```
---
10   template:
11     metadata:
12       labels:
13         type: my-nginx-pod
14     spec:
15       containers:
16         - name: my-nginx-container
17         image: nginx
18       ports:
19         - containerPort: 80
20 ---
21   apiVersion: v1
22   kind: Service
23     metadata:
24       name: my-nginx-service
25     spec:
26       ports:
27         - port: 8080
28         targetPort: 80
29         protocol: TCP
30     selector:
31       - type: my-nginx-pod
---
```
Another Example of Composite Application Type: Job

- Goal: enforce policies for execution and completion of groups of tasks (batch jobs, scheduled jobs)
Resource Orchestration in Kubernetes: Controllers

```python
for {
    desired := getDesiredState()
    current := getCurrentState()
    makeChanges(desired, current)
}
```

Source: [https://github.com/kubernetes/community/blob/master/contributors/devel/controllers.md](https://github.com/kubernetes/community/blob/master/contributors/devel/controllers.md)
Controller Example - Replica Set for Deployment myD:

```python
vagrant@ubuntu-xenial: ~
1  def controller():
2      for event in apiServer.deployments.watch(id=myD.id).stream:
3          curReplicas = myD.pods.filter(state="running").length  # Event: User changed #replicas
4          desiredReplicas = event.resource.numReplicas
5          matchReplicas(myD, curReplicas, desiredReplicas)
6
7      for event in apiServer.pods.watch(label=myD.label).stream:
8          if event.resource.id in myD.pods:
9              curReplicas = myD.pods.filter(state="running").length
10             desiredReplicas = myD.numReplicas
11             matchReplicas(myD, curReplicas, desiredReplicas)
12
13  def matchReplicas(D, curReplicas, desiredReplicas):
14      if curReplicas > desiredReplicas:
15          killPods(D.pods, curReplicas - desiredReplicas)
16      elif curReplicas < desiredReplicas:
17          addPods(D.podTemplate, desiredReplicas - curReplicas)
```
Kubernetes: Open Source Project and Community

- One of the most vibrant projects on Github
  - 48K stars, 16K forks, 44K PRs, 75K commits, 2000 contributors

9 lines of code per minute
Kubernetes: Commercial Ecosystem

- Governed by Cloud Native Computing Foundation (CNCF)
  - Industry-wide consensus among leading (and competing) companies

- Multiple commercial Kubernetes products and hosted offerings
  - Red Hat, IBM, CoreOS, etc; Google GKE, Amazon EKS, etc