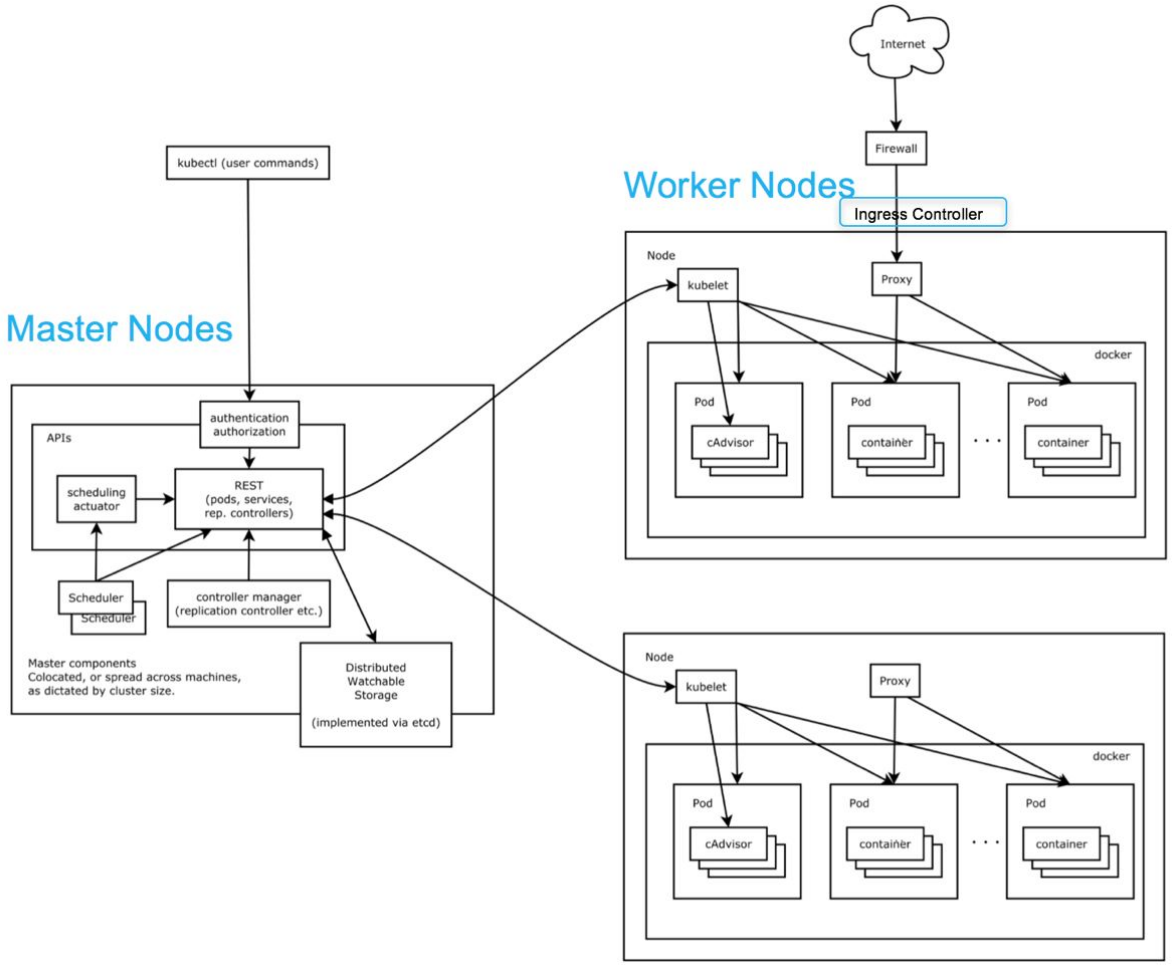
Kubernetes Cheatsheet

Kubernetes Architecture



https://lh3.googleusercontent.com/EU3DgtFKagWp5S0UpKj-wRgx8WK2nvQ2BG-4dGio57pGNj42A7Lip9IARBba34hIm84-_7zwWt6iImQE8beSqLxpzXm-2w_84M_X2IHQ7jvpWtIDMF81hmq6N4hGSxp6DQoFW5qX

Master components:

1. **kube-apiserver -** Its main Kubernetes component that is managing all Master and Worker components using REST API. The Kubernetes API server validates and configures data for the api objects which include pods, services, replicationcontrollers, and others.

2. **etcd** - Kubernetes datastore used for saving cluster state in distributed key-value store. Etcd is the most crucial cluster component, and the only one that needs to be backup.

3. **kube-scheduler** - Its a component that is responsible for scheduling pods on nodes. Scheduler is making decisions taking into account: node health, resource utilization, node selectors, affinity/anti-affinity rules, taints and tolerations etc.
4. **kube-controller-manager** - Its responsible for running Kubernetes controllers: replication controller, endpoint controller, node controller, service account controller, token controller etc. Controllers are running endless control loops that regulate the state of Kubernetes components.

5. **cloud-controller-manager** (optional) - Its a component that integrates Kubernetes with cloud providers services. Using cloud-controller-manager Kubernetes can:

- dynamically request volumes and attach them to the pods
- create (and configure) load balancer and (using services) attach it to the pods
- and many more

Worker (minion) components:

1. **kubelet** - Its a daemon that is responsible for managing container Runtime that is installed on each worker node.
2. **kube-proxy** - Provides service abstraction by passing traffic into pods (using iptables).
3. **container runtime** - Its a set of standards and technologies that together can run containerized applications.

kubectl command

kubectl is a binary used for Kubernetes objects management. Using this command you can view, create, delete and edit K8s objects.

Syntax:

```
kubectl [command] [TYPE] [NAME] [flags]
```

`command`: Specifies the operation that you want to perform on one or more resources, for example create, get, describe, delete.

`TYPE`: Specifies the resource type - pod, deployment, job, cronjob etc

`NAME`: Specifies the name of the resource. Names are case-sensitive. If the name is omitted, details for all resources are displayed, for example **kubectl get pods.**

`flags`: Specifies optional flags.

Examples:

Command	Description
`kubectl run nginx --image nginx`	Create nginx deployment
`kubectl expose deployment nginx --type=NodePort`	Expose nginx deployment on hosts
`kubectl get pods`	Get list of running pods
`kubectl describe pod $POD_NAME`	Describe pod settings
`kubectl port-forward $POD_NAME -i`	Create tunnel to app in a pod
`kubectl exec -it $POD_NAME -- command`	Execute command in a pod
`kubectl label pods $POD_NAME $LABEL=VALUE`	Add label to a pod
`kubectl -f $MANIFEST_FILE`	Create object based on manifest file

Official kubectl cheatsheet is available at:
https://kubernetes.io/docs/reference/kubectl/cheatsheet/

Kubernetes Objects and Features

Each Kubernetes object needs the following fields:

`apiVersion` - Which version of the Kubernetes API you're using to create this object
`kind` - What kind of object you want to create
`metadata` - Data that helps uniquely identify the object, including a name string, UID, and optional namespace
`spec` - The precise format of the object spec is different for every Kubernetes object, and contains nested fields specific to that object.

Namespaces
Used for objects separation, provides scope for names, and splits physical cluster into logical spaces. You can also limit the quota to each namespace for resources utilization.

There are 3 default namespaces:
`default` - default namespace for Kubernetes objects without a namespace
`kube-system` - namespace for objects created by Kubernetes
`kube-public` - namespace for public resources

Pods
Pod is the smallest shedulable unit in Kubernetes. A pod describes and application running on Kubernetes. Pod define group of containers that share: namespaces, volumes, ip address and port space. Containers within a POD can communicate using localhost. Kubernetes Pods are mortal. They are born they die, and they are not resurrected.
Example manifest:

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

PodSpec fields:
`affinity` - define affinity rules
`containers` - container configuration (array)
`dnsConfig` - manage name resolution inside a POD
`dnsPolicy` - configure one of dns policies
`hostname` - define a hostname (in POD runtime)
`imagePullSecrets` - provide credentials needed for secured docker registries
`initContainers` - configure container (and action) that is executed before main PODs
`nodeSelector` - declare on which node POD should be running (using labels and selectors)
`priority` - define POD priority
`restartPolicy` - configure conditions according to which POD will be restarted
`serviceAccountName` - define service account name that will be used for POD run
`terminationGracePeriodSeconds` - define grace POD termination time
`tolerations` - sets tolerations
`volumes` - define volumes that can be mounted in containers

Pod Container v1 core array:
`args` - arguments passed to command
`command` - PID1 process in container
`env` - define environment variables for a container
`envFrom` - import array of variables from configMap or Secret
`image` - set container image
`imagePullPolicy` - define image pull policy
`lifecycle` - configure PostStart or PreStop lifecycle hooks
`livenessProbe` - configure livenessProbe healthcheck
`readinessProbe` - configure readinessProbe healthcheck

`name` - define Pod name
`ports` - define container ports
`resources` - set hard and soft quotas for memory and cpu
`securityContext` - manage advanced security features
`stdin` - allocate runtime buffer for stdin
`tty` - allocates tty for the container itself
`volumeMounts` - volume mountpoint in a container
`workingDir` - container working directory

Labels and Selectors
Labels are key-values pairs that can be attached to different kubernetes objects. They help with organisation of K8s objects according to different purpose, environment or client. They can be added to an object at creation time and can be added or modified at the runtime.

Assigning labels:
```
metadata:
  labels:
    app: nginx
```

Assigning selectors:
```
spec:
  selector:
    app: nginx
```

Equality-based selectors	Set-based selectors
kubectl get pods -l env=production	kubectl get pods -l env=production

`selector:`
`  env: production`
	`selector:`
	`  matchLabels:`
	`  env: production`
	`  matchExpressions:`
	`  - {key: env: In, values: [production]}`
	`  - {key: env, operator: NotIn, values: [development]}`

Matching objects must satisfy all of the specified label constraints, though they may have additional labels as well.	Not every resource supports set-based selectors.

Replication Controllers
ReplicationController (rc or rcs) ensures that specified number of PODs are running in a cluster. Replication Controller doesn't support set-based selectors - use RepicaSet when its possible.

Example RC manifest:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

ReplicaSets

RepicaSet (rs) is a new version of Replication Controller that supports set-based selectors. It ensures that specified number of PODs are running in a cluster.

Example RS manifest:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchExpressions:
      - {key: app, operator: In, values: [nginx]}
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
```

```
        image: nginx
        ports:
        - containerPort: 80
```

Deployments
Deployments control ReplicaSets and PODs configuration using one manifest file. After creation state of Pods and ReplicaSets are checked by Deployment Controller. Deployments are used for long running processes - daemons.

Example Deployment manifest:

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

DeploymentSpec:

`replicas` - number of related Pod replicas (managed by ReplicaSets)
`revisionHistoryLimit` - specify how many old ReplicaSets for this Deployment you want to retain. The rest will be garbage-collected in the background. By default, it is 10.
`strategy:`
  `type: Recreate` - kills existing Pods before creating new ones

`strategy:`
`type: RollingUpdate` - define rolling update strategy (no outage)
`rollingUpdate:`
`maxUnavailable: 2` - specifies the maximum number of Pods that can be unavailable during the update process.
`maxSurge: 2` - specifies the maximum number of Pods that can be created over the desired number of Pods.

Jobs
Jobs are tasks that run to completion (they do not behave like daemons - long running processes.

Example manifest:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: example-job
spec:
  activeDeadlineSeconds: 120
  backoffLimit: 3
  completions: 3
  parallelism: 3
  template:
    metadata:
      name: example-job
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl"]
        args: ["-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

JobsSpec:
`activeDeadlineSeconds` - specify a time (related to startTime) after which K8s will try to terminate a POD
`backoffLimit` - number of retries before marking job as failed
`completions` - number of successfully finished pods
parallelism - the maximum desired number of pods the job should run at any given time

Cronjobs
Runs jobs at specified point in time:
- Once at a specified point in time

- Repeatedly at a specified point in time

Kubernetes CronJobs are using cron format to define in which jobs will be executed.

```
# ┌──────────────── minute (0 - 59)
# │ ┌──────────────── hour (0 - 23)
# │ │ ┌──────────────── day of month (1 - 31)
# │ │ │ ┌──────────────── month (1 - 12)
# │ │ │ │ ┌──────────────── day of week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                 7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * command to execute
```

Example manifest:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: example-cronjob
spec:
  schedule: "0 0 * * *"
  jobTemplate:
    spec:
      backoffLimit: 0
      template:
        spec:
          containers:
          - name: example-job
            image: ubuntu
            command: ["/bin/sh", "-c", "--"]
            args: ["for i in `seq 1 1 100`; do echo $i; done"]
        restartPolicy: Never
```

CronJobSpec v1beta1 batch:

`concurrencyPolicy` - Specifies how to treat concurrent executions of a Job.
`Allow` - allows for concurrent jobs (default)
`Forbid` - skips next run if previous one doesn't finished
`Replace` - kills currently running job and create a new one
`failedJobsHistoryLimit` - The number of failed finished jobs to retain.
`schedule` - cron format schedule of a job
`startingDeadlineSeconds` - time buffer for missed job execution, after this deadline job will be counted as failed
`successfulJobsHistoryLimit` - The number of failed finished jobs to retain.
`suspend` - suspend job execution, it will not suspend already running job

DaemonSets
Ensures that one instance of a Pod is running on each (or some) nodes. When new nodes are added to cluster K8s automatically runs Pod from a DaemonSets on a new node. Mainly used for:
- Cluster storage daemons (ceph, glusterd)
- Logs collector daemons (fluentd, logstash)
- Monitoring daemons (Prometheus Node Exporter, Datadog agent...)

Example manifest file:

```
apiVersion: v1
kind: DaemonSet
metadata:
  name: daemonset-example
spec:
  updateStrategy: OnDelete
  selector:
    matchLabels:
      node: daemonset-example
  template:
    metadata:
      labels:
        app: daemonset-example
    spec:
      containers:
      - name: daemonset-example
        image: ubuntu
        command:
        - /bin/sh
        args:
        - -c
        - >-
          while [ true ]; do
          echo "DaemonSet running on $(hostname)" ;
          sleep 10 ;
          done
```

StatefullSets
StatefulSets are the way of managing Pods that persists their identity. Each Pod managed by StatefulSets has a persistent identifier that it maintains across any rescheduling.
Pods are using the following naming convention:
<statefulset name>-<ordinal index>
nginx-01, nginx-02...
StatefulSets Pods behaviour:
- each Pod has a unique network identifier (dns name)
- each Pod requires persistent storage (manually created before deployment or using persistent volume claim)

- deployment and scaling of Pods is done in ordered manner <
- deletion and termination of Pods is done in ordered manner >
- rolling updates of Pods are also done in ordered manner <

Annotations
Annotations are a special kind of metadata that are attached to objects. Different client tools and libraries can retrieve this metadata and use them for different purposes. Annotations are not used internally by k8s.

Example Annotation snippet:

```
metadata:
  annotations:
    key1: "value"
    key2: "value"
```

Services
Services are REST objects that act as persistent endpoint and loadbalancer for pods communication. Each service name and IP is propagated by Kubernetes DNS to the entire K8s cluster. Service redirects traffic based on labels configured on Pods.

ClusterIP(default) - Creates a service that is accessible only inside a cluster.

Example manifest file:

```
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  type: ClusterIP
  ports:
  - port: 80
```

NodePort - Publish service on a node using one of dynamically allocated high ports (default: 30000-32767), port can be also set to particular value (using nodePort field).

Example manifest file:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    service: nginx
```

```
  name: nginx
spec:
  type: NodePort
  ports:
  - name: "443"
    port: 443
    targetPort: 443
    nodePort: 30443
  selector:
    app: nginx
```

LoadBalancer - This service will provision external loadbalancer using cloud providers infrastructure.

```
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: LoadBalancer
```

ExternalName - Maps service name to external DNS name.

```
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  type: ExternalName
  externalName: nginx.com
```

ExternalIP - Maps external ip to Kubernetes service. Externals ip needs to route to one or more cluster nodes.

```
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  selector:
```

```
    app: nginx
  ports:
  - name: http
    protocol: TCP
    port: 80
  externalIPs:
  - 192.168.1.1
```

Ingress
An API object that manages external access to the services in a cluster, typically HTTP. Ingress is a collection of load balancing, SSL termination and name-based virtual hosting rules. Ingress requires 2 components: ingress controller and ingress object.

Nginx ingress controller can be installed using deployment manifest, instruction can be found at:
https://github.com/nginxinc/kubernetes-ingress/blob/master/build/README.md

Example manifest file (name-based):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress
spec:
  rules:
  - host: warsaw.example.com
    http:
      paths:
      - backend:
          serviceName: warsaw-web
          servicePort: 80
  - host: gdansk.example.com
    http:
      paths:
        backend:
          serviceName: gdansk-web
          servicePort: 80
```

Example manifest file (reverse-proxy):

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
```

```
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        backend:
          serviceName: web
          servicePort: 80
      - path: /metrics
        backend:
          serviceName: grafana
          servicePort: 80
```

Network Policies
Network Policies are Kubernetes internal firewalls. A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.

Network Policies require 2 components to work: CNI provider (Calico, Canal, Weave Net) and Network Policy manifest. CNI providers can be deployed using Kubernetes manifest, installation instruction can be found at websites of CNI providers.

Example Network Policy manifest:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: bookstore
      role: api
  ingress:
  - from:
      - podSelector:
          matchLabels:
            app: bookstore
```

Volumes
Volumes are external data sources that can be mounted inside a pod. Kubernetes supports many different types of volumes, each of them can be used by services inside pods:

```
awsElasticBlockStore
azureDisk
azureFile
cephfs
```

```
configMap
csi
downwardAPI
emptyDir
fc (fibre channel)
flocker
gcePersistentDisk
gitRepo (deprecated)
glusterfs
hostPath
iscsi
local
nfs
persistentVolumeClaim
projected
portworxVolume
quobyte
rbd
scaleIO
secret
storageos
vsphereVolume
```

Each volume type is mounted in a different way, instructions about mounting particular volumes can be found at: https://kubernetes.io/docs/concepts/storage/volumes/

Example of emptyDir manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  restartPolicy: Always
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: Always
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
    emptyDir: {}
```

ConfigMaps
ConfigMaps in Kubernetes are used for storing different types of application configurations (treat them as config files from /etc). ConfigMaps can be created using **`kubectl create configmap <map-name> <data-source>`** command or based on ConfigMap manifest file (recommended).

ConfigMaps can be mounted as:
- files
- environment variables
Example manifest snippet that mounts ConfigMap as file:

```
  volumeMounts:
  - name: example-configmap-volume
    mountPath: /example
volumes:
- name: example-configmap-volume
  configMap:
    name: example-configmap

apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  example_file: "Content of example file"
```

Example manifest snippet that mounts ConfigMap as env variable:

```
  envFrom:
  - configMapRef:
      name: example-configmap
volumes:
- name: example-configmap-volume
  configMap:
    name: example-configmap
```

Example manifest snippet that mounts only one env variable from ConfigMap:

```
  env:
  - name: ENV_VAR_NAME
    valueFrom:
      configMapKeyRef:
        name: example-configmap
        key: ENV1
    volumes:
    - name: example-configmap-volume
```

```
      configMap:
        name: example-configmap

apiVersion: v1
kind: ConfigMap
metadata:
  name: example-configmap
data:
  ENV1: "Content of ENV1 example variable"
  ENV2: "Content of ENV2 example variable"
```

Secrets

Secrets are designed to store fragile security data as passwords, certificates or tokens. Main purpose of secrets is storing security data as: db name, db password, tokens etc. There are three types of secrets:

- docker-registry Create a secret for use with a Docker registry
- generic Create a secret from a local file, directory or literal value
- tls Create a TLS secret

Secrets can be created using **`kubectl create secret <secret-type> <secret-name> <data-source>`** command or based on Secret manifest file (recommended).

```
  envFrom:
  - secretRef:
      name: api-credentials
  volumeMounts:
  - name: secrets-volume
    mountPath: /secrets
volumes:
- name: secrets-volume
  secret:
    secretName: secrets

apiVersion: v1
data:
  password: c2VjcmV0LXBhc3N3b3JkCg==
  user: YWRtaW4K
kind: Secret
metadata:
  name: secrets
```

Liveness and readiness probes
The kubelet uses probes to know when to restart a Container or when detach pods from Service. There are 3 types of probes:
- exec
- http
- socket

There are 2 types of probes:

livenessProbe	readinessProbe
The readiness check checks pod health and restarts it when probe fails If your container returns exit codes different than 0, you don't need livenessProbe	The readiness test will make sure that at startup, the pod receive traffic only when test succeed

`General probe options:`
`initialDelaySeconds`: Number of seconds after the container has started before liveness or readiness probes are initiated.
`periodSeconds`: How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
`timeoutSeconds`: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.
`successThreshold`: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness. Minimum value is 1.

`Http specific options:`
`host`: Host name to connect to, defaults to the pod IP. You probably want to set "Host" in httpHeaders instead.
`scheme`: Scheme to use for connecting to the host (HTTP or HTTPS). Defaults to HTTP.
`path`: Path to access on the HTTP server.
`httpHeaders`: Custom headers to set in the request. HTTP allows repeated headers.
port: Name or number of the port to access on the container. Number must be in the range 1 to 65535.

Example snippet of livenessProbe:

```
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

Example snippet of readinessProbe:

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 81
  initialDelaySeconds: 15
  timeoutSeconds: 1
  periodSeconds: 15
```

Horizontal Pod Autoscalers with Heapster
HPA can dynamically scale up and down Pods based on CPU or memory pressure (memory pressure requires external scripts https://github.com/powerupcloud/kubernetes-1).

HPA requires 2 components:
- heapster or metrics-server (can be installed using Deployment and DaemonSet)
- resource limits assigned to pods

Example resource requests and limits snippet:

```
resources:
  requests:
    memory: "64Mi"
    cpu: "250m"
  limits:
    memory: "128Mi"
    cpu: "500m"
```

Example HPA manifest:

```
kind: HorizontalPodAutoscaler
metadata:
  name: php-hpa
spec:
  scaleTargetRef:
    apiVersion: extensions/v1beta1
    kind: Deployment
    name: php
  minReplicas: 2
  maxReplicas: 4
  targetCPUUtilizationPercentage: 80
```

Affinity and anti-affinity rules
Affinity and anti-affinity rules allows you to do much more complex scheduling rules than with nodeSelector:
- The selecting language is more expressive.

- You can create rules that are not hard requirements but rather prefered rules, which means that scheduler can assign Pod to node even if rules cannot be met.
- You can create rules that takes other pod label into account

There are 2 kinds of affinity and anti-affinity rules:
node rules
pod rules

There are 2 types of node affinity rules:
1. Hard requirement (that works like the nodeSelector) - The rules must be met before pod will be scheduled.
- **`requiredDuringSchedulingIgnoredDuringExecution`**
2. Soft requirement - Even if the rules are not set, Pod can still be scheduled, this is just a preference.
- **`preferredDuringSchedulingIgnoredDuringExecution`**

Interpod affinity rules can influence scheduling based on the labels of other pods that are already running.
Remember that each pod is running inside a namespace so affinity rules apply to the pods in a particular namespace (if namespace is not defined affinity rule will apply to the namespace of a pod).
There are 2 types of interpod affinity rules:
1. Hard requirement
`requiredDuringSchedulingIgnoredDuringExecution`
2. Soft requirement
`preferredDuringSchedulingIgnoredDuringExecution`

Managing Namespace limits and quotas for CPU and Memory
If your cluster resources are divided using namespaces, and each namespace is used for different purpose for example: staging, testing and production. Its worth to set resource quotas per namespace - you will be sure that resources from staging namespace will not throttle production services. ResourceQuota is working in a namespace in which it was created.

Example ResourceQuota manifest:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Pod priorities and preemptions
Priority indicates the importance of a Pod relative to other Pods. If a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to make scheduling of the pending Pod possible.

Setting Pods priorities requires 2 steps:
- Creation of PriorityClass
- Running Pods with attached priorityClassName
Example assignment of PriorityClassName snippet:

```
containers:
- name: nginx
  image: nginx
  imagePullPolicy: Always
priorityClassName: high-priority
```

Example PriorityClass manifest:

```
apiVersion: scheduling.k8s.io/v1beta1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "Priority Class description"
```

Helm (highly recommended!!)
Helm is a Kubernetes package manager that allows to deploy Kubernetes applications (Deployments, Services, ConfigMaps, Secrets etc), and treat them as a single entity. One of the most powerful features of Helm is built-in Go Templating language, that allows for dynamic variable injection, into Kubernetes manifests, setting conditionals and many more.

Helm documentation is available at: https://docs.helm.sh/