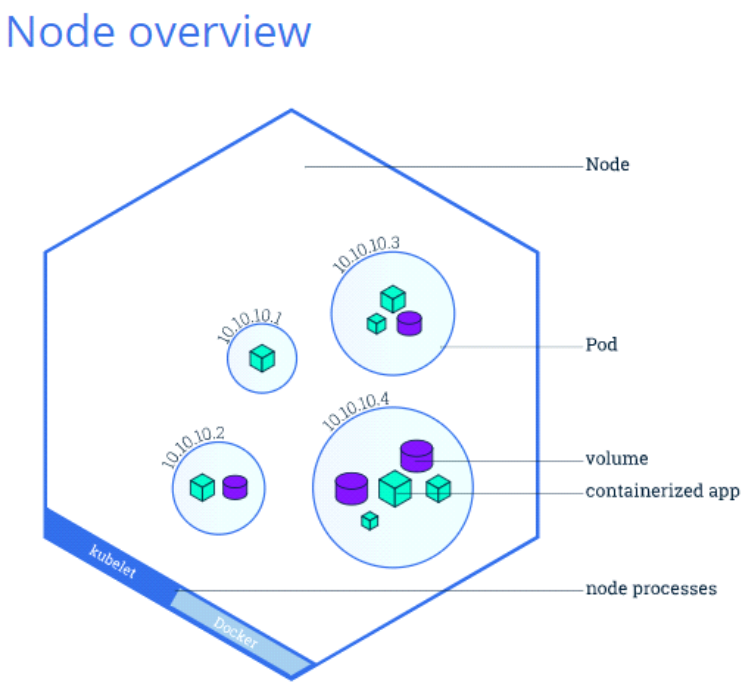
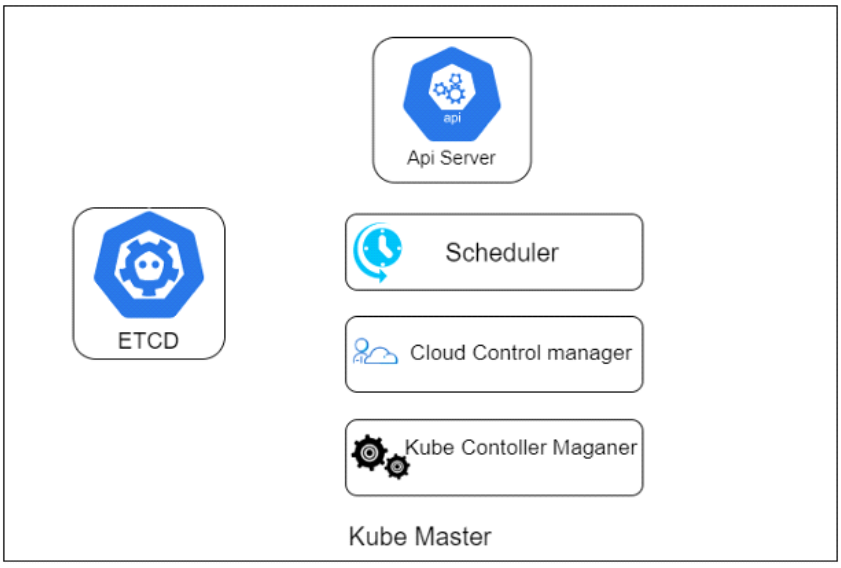
Anatomy of kubernetes cluster

Wednesday, January 13, 2021 4:28 PM



1. **Master node:**
   a. Masters run several components that provide us with something called a *control plane*. They make decisions about the cluster, such as where to schedule specific workloads, etc.
   b. The Master is responsible for the state of the cluster. It continually keeps an eye on everything to make sure that everything is working fine.
   c. Node runs the component that provides the runtime environment (they are basically the workers with the container runtime). These nodes provide the resources for the cluster such as the CPU, RAM, etc.
   d. When you deploy a container on Kubernetes, the Master will pick a node to run it on.
   e. Here's a diagram of a Kubernetes Master node.



f. API server

   i. The API server is the front end of the control plane; it exposes the API for all the master functions.
   Every time you communicate with the Master or something else interacts with the Master (Cloud shell in GCP), it will be through this API server.

g. Etcd

   i. Etcd is Kubernetes' own database. It stores all Kubernetes' configurations and states. I call it a database, but Etcd is just a key-value store.

h. Scheduler

   i. The scheduler is responsible for scheduling workloads. What that means is when you want to deploy a container, the Scheduler will pick a node to run that container on.
   The node it selects can be affected by a lot of factors such as current load on each available node, requirements of your container, and some other customizable constrains.

i. Cloud Controller Manager

   i. Cloud Controller Manager allows Kubernetes to work with cloud platforms. Remember, Kubernetes is itself an open-source project with contributions from large companies and not just Google, so it doesn't natively contain functionality for GCP only.
   This manager is responsible for managing things like networking and load balancing as it translates to the product and services of a particular cloud platform.

j. Kube Control Manager

   i. The Kube Control Manager's job is to manage a handful of controllers in the cluster. The controllers themselves look after things like nodes and a few other

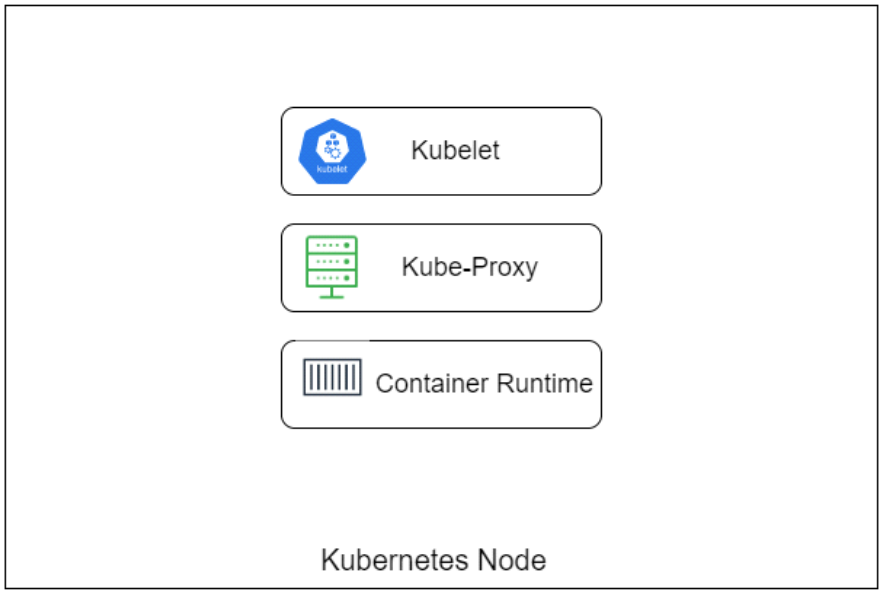
objects.
These controllers include:

1. Node controller: Responsible for noticing and responding when nodes go down.
2. Replication controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.
3. Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
4. Service Account & Token controllers: Create default accounts and API access tokens for new namespaces

2. Worker Node:

a. Now let's look at the node (Worker) of Kubernetes. They are a lot more straightforward than the Master.

b. Here's a diagram of the Worker node.



c. Kubelet

   i. Kubelet is an agent of Kubernetes. It communicates with the control plane and takes instructions like deploying containers when it's told to. Kubelet also performs probes as part of the diagnostic checks, on nodes.

d. Kube-proxy

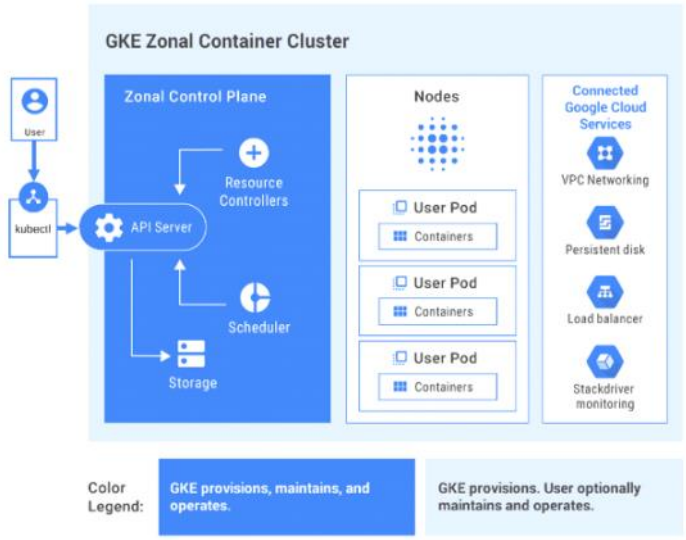
   i. The Kube-proxy is responsible for network connections in and out of the node.

e. Container runtime

   i. The node will run Docker as a container runtime to allow it to run containers.

Kubernetes GCP architecture

Monday, January 13, 2020 9:16 PM



Docker is a containerization platform, and Kubernetes is a container orchestrator for container platforms like Docker.

Cluster architecture

In Google Kubernetes Engine (GKE), a cluster consists of at least one cluster master and multiple worker machines called nodes. These master and node machines run the Kubernetes cluster orchestration system.

A cluster is the foundation of GKE: the Kubernetes objects that represent your containerized applications all run on top of a cluster.

Cluster master

The cluster master runs the Kubernetes control plane processes, including the Kubernetes API server, scheduler, and core resource controllers. The master's lifecycle is managed by GKE when you create or delete a cluster. This includes upgrades to the Kubernetes version running on the cluster master, which GKE performs automatically, or manually at your request if you prefer to upgrade earlier than the automatic schedule.

Cluster master and the Kubernetes API

The master is the unified endpoint for your cluster. All interactions with the cluster are done via Kubernetes API calls, and the master runs the Kubernetes API Server process to handle those requests. You can make Kubernetes API calls directly via HTTP/gRPC, or indirectly, by running commands from the Kubernetes command-line client (kubectl) or interacting with the UI in the Cloud Console.

The cluster master's API server process is the hub for all communication for the cluster. All internal cluster processes (such as the cluster nodes, system and components, application controllers) all act as clients of the API server; the API server is the single "source of truth" for the entire cluster.

Master and node interaction

The cluster master is responsible for deciding what runs on all of the cluster's nodes. This can include scheduling workloads, like containerized applications, and managing the workloads' lifecycle, scaling, and upgrades. The master also manages network and storage resources for those workloads.

The master and nodes also communicate using Kubernetes APIs.

Master interactions with the gcr.io container registry

When you create or update a cluster, container images for the Kubernetes software running on the masters (and nodes) are pulled from the gcr.io container registry. An outage affecting the gcr.io registry may cause the following types of failures:

- Creating new clusters will fail during the outage.
- Upgrading clusters will fail during the outage.
- Disruptions to workloads may occur even without user intervention, depending on the specific nature and duration of the outage.

In the event of a zonal or regional outage of the gcr.io container registry, Google may redirect requests to a zone or region not affected by the outage.

Nodes

A cluster typically has one or more nodes, which are the worker machines that run your containerized applications and other workloads. The individual machines are Compute Engine VM instances that GKE creates on your behalf when you create a cluster.

Each node is managed from the master, which receives updates on each node's self-reported status. You can exercise some manual control over node lifecycle, or you can have GKE perform automatic repairs and automatic upgrades on your cluster's nodes.

A node runs the services necessary to support the Docker containers that make up your cluster's workloads. These include the Docker runtime and the Kubernetes node agent (kubelet) which communicates with the master and is responsible for starting and running Docker containers scheduled on that node.

In GKE, there are also a number of special containers that run as per-node agents to provide functionality such as log collection and intra-cluster network connectivity.

Node Pools

A node pool is a group of nodes within a cluster that all have the same configuration. Node pools use a NodeConfig specification. Each node in the pool has a Kubernetes node label, cloud.google.com/gke-nodepool, which has the node pool's name as its value. A node pool can contain only a single node or many nodes.

When you create a cluster, the number and type of nodes that you specify becomes the default node pool. Then, you can add additional custom node pools of different sizes and types to your cluster. All nodes in any given node pool are identical to one another.

For example, you might create a node pool in your cluster with local SSDs, a minimum CPU platform, preemptible VMs, a specific node image, larger instance sizes, or different machine types. Custom node pools are useful when you need to schedule Pods that require more resources than others, such as more memory or more local disk space. If you need more control of where Pods are scheduled, you can use node taints.

You can create, upgrade, and delete node pools individually without affecting the whole cluster using the gcloud container node-pools command. You cannot configure a single node in a node pool; any configuration changes affect all nodes in the node pool.

By default, all new node pools run the latest stable version of Kubernetes. Existing node pools can be manually upgraded or automatically upgraded. You can also run multiple Kubernetes node versions on each node pool in your cluster, update each node pool independently, and target different node pools for specific deployments.

Deploying Services to specific node pools

When you define a Service, you can indirectly control which node pool it is deployed into. The node pool is not dependent on the configuration of the Service, but the configuration of the Pods.

- You can explicitly deploy a Pod to a specific node pool by setting a nodeSelector in the Pod manifest. This forces a Pod to run only on Nodes in that node pool.
- You can specify resource requests for the containers. The Pod will only run on nodes that satisfy the resource requests. For instance, if the Pod definition includes a container that requires four CPUs, the Service will not select Pods running on Nodes with two CPUs.

Nodes in multi-zonal clusters

If you created a multi-zonal cluster, all of the node pools are replicated to those zones automatically. Any new node pool is

automatically created in those zones. Similarly, any deletions delete those node pools from the additional zones as well.

Note that because of this multiplicative effect, this may consume more of your project's quota for a specific region when creating node pools.

Pods on GKE

Monday, January 13, 2020 9:22 PM

Pods are the smallest, most basic deployable objects in Kubernetes. A Pod represents a single instance of a running process in your cluster.

Pods contain one or more containers, such as Docker containers. When a Pod runs multiple containers, the containers are managed as a single entity and share the Pod's resources. Generally, running multiple containers in a single Pod is an advanced use case.

Pods also contain shared networking and storage resources for their containers:

- Network: Pods are automatically assigned unique IP addresses. Pod containers share the same network namespace, including IP address and network ports. Containers in a Pod communicate with each other inside the Pod on localhost.
- Storage: Pods can specify a set of shared storage volumes that can be shared among the containers.

You can consider a Pod to be a self-contained, isolated "logical host" that contains the systemic needs of the application it serves.

A Pod is meant to run a single instance of your application on your cluster. However, it is not recommended to create individual Pods directly. Instead, you generally create a set of identical Pods, called replicas, to run your application. Such a set of replicated Pods are created and managed by a controller, such as a Deployment. Controllers manage the lifecycle of their constituent Pods and can also perform horizontal scaling, changing the number of Pods as necessary.

Although you might occasionally interact with Pods directly to debug, troubleshoot, or inspect them, it is highly recommended that you use a controller to manage your Pods.

Pods run on nodes in your cluster. Once created, a Pod remains on its node until its process is complete, the Pod is deleted, the Pod is evicted from the node due to lack of resources, or the node fails. If a node fails, Pods on the node are automatically scheduled for deletion.

Pod lifecycle

Pods are ephemeral. They are not designed to run forever, and when a Pod is terminated it cannot be brought back. In general, Pods do not disappear until they are deleted by a user or by a controller.

Pods do not "heal" or repair themselves. For example, if a Pod is scheduled on a node which later fails, the Pod is deleted. Similarly, if a Pod is evicted from a node for any reason, the Pod does not replace itself.

Each Pod has a PodStatus API object, which is represented by a Pod's status field. Pods publish their phase to the status: phase field. The phase of a Pod is a high-level summary of the Pod in its current state.

When you run kubectl get pod to inspect a Pod running on your cluster, a Pod can be in one of the following possible phases:

- Pending: Pod has been created and accepted by the cluster, but one or more of its containers are not yet running. This phase includes time spent being scheduled on a node and downloading images.
- Running: Pod has been bound to a node, and all of the containers have been created. At least one container is running, is in the process of starting, or is restarting.
- Succeeded: All containers in the Pod have terminated successfully. Terminated Pods do not restart.
- Failed: All containers in the Pod have terminated, and at least one container has terminated in failure. A container "fails" if it exits with a non-zero status.
- Unknown: The state of the Pod cannot be determined.

Additionally, PodStatus contains an array called PodConditions, which is represented in the Pod manifest as conditions. The field has a type and status field. conditions indicates more specifically the conditions within the Pod that are causing its current status.

The type field can contain PodScheduled, Ready, Initialized, and Unschedulable. The status field corresponds with the type field, and can contain True, False, or Unknown.

Note: You can run kubectl get pod [POD_NAME] -o yaml to view the Pod's entire manifest, including the phase and conditions fields.
Creating Pods
Because Pods are ephemeral, it is not necessary to create Pods directly. Similarly, because Pods cannot repair or replace themselves, it is not recommended to create Pods directly.

Instead, you can use a controller, such as a Deployment, which creates and manages Pods for you. Controllers are also useful for rolling out updates, such as changing the version of an application running in a container, because the controller manages the whole update process for you.

Pod requests

When a Pod starts running, it requests an amount of CPU and memory. This helps Kubernetes schedule the Pod onto an appropriate node to run the workload. A Pod will not be scheduled onto a node that doesn't have the resources to honor the Pod's request. A request is the minimum amount of CPU or memory that Kubernetes guarantees to a Pod.

Note: Pod requests differ from and work in conjunction with Pod limits.

You can configure the CPU and memory requests for a Pod, based on the resources your applications need. You can also specify requests for individual containers running in the Pod. Keep the following in mind:

- The default request for CPU is 100m. This is too small for many applications, and is probably much smaller than the amount of CPU available on the node.
- There is no default request for memory. A Pod with no default memory request could be scheduled onto a node without enough memory to run the Pod's workloads.
- Setting too small a value for CPU or memory requests could cause too many Pods or a suboptimal combination of Pods to be scheduled onto a given node and reduce performance.
- Setting too large a value for CPU or memory requests could cause the Pod to be unschedulable and increase the cost of the cluster's resources.
- In addition to, or instead of, setting a Pod's resources, you can specify resources for individual containers running in the Pod. If you only specify resources for the containers, the Pod's requests are the sum of the requests specified for the containers. If you specify both, the sum of requests for all containers must not exceed the Pod requests.

It is strongly recommended that you configure requests for your Pods, based on the requirements of the actual workloads.

Pod limits

By default, a Pod has no upper bound on the maximum amount of CPU or memory it can use on a node. You can set limits to control the amount of CPU or memory your Pod can use on a node. A limit is the maximum amount of CPU or memory that Kubernetes guarantees to a Pod.

In addition to, or instead of, setting a Pod's limits, you can specify limits for individual containers running in the Pod. If you only specify limits for the containers, the Pod's limits are the sum of the limits specified for the containers. However, each container can only access resources up to its limit, so if you choose to specify the limits on containers only, you must specify limits for each container. If you specify both, the sum of limits for all containers must not exceed the Pod limit.

Note: A limit must always be greater than or equal to a request for the same type of resource. If you attempt to set the limit below the request, the Pod's container cannot run and an error is logged.

Limits are not taken into consideration when scheduling Pods, but can prevent resource contention among Pods on the same node, and can prevent a Pod from causing system instability on the node by starving the underlying operating system of resources.

Note: Pod limits differ from and work in conjunction with Pod requests.

It is strongly recommended that you configure limits for your Pods, based on the requirements of the actual workloads.

Pod templates

Controller objects, such as Deployments and StatefulSets, contain a Pod template field. Pod templates contain a Pod specification which determines how each Pod should run, including which containers should be run within the Pods and which volumes the Pods should mount.

Controller objects use Pod templates to create Pods and to manage their "desired state" within your cluster. When a Pod template is changed, all future Pods reflect the new template, but all existing Pods do not.

Controlling which nodes a Pod runs on

By default, Pods run on nodes in the default node pool for the cluster. You can configure the node pool a Pod selects explicitly or implicitly:

You can explicitly force a Pod to deploy to a specific node pool by setting a nodeSelector in the Pod manifest. This forces a Pod to run only on Nodes in that node pool.

You can specify resource requests for the containers you run. The Pod will only run on nodes that satisfy the resource requests. For instance, if the Pod definition includes a container that requires four CPUs, the Service will not select Pods running on Nodes with two CPUs.

Pod usage patterns

Pods can be used in two main ways:

- Pods that run a single container. The simplest and most common Pod pattern is a single container per pod, where the single container represents an entire application. In this case, you can think of a Pod as a wrapper.
- Pods that run multiple containers that need to work together. Pods with multiple containers are primarily used to support colocated, co-managed programs that need to share resources. These colocated containers might form a single cohesive unit of service—one container serving files from a shared volume while another container refreshes or updates those files. The Pod wraps these containers and storage resources together as a single manageable entity.

Each Pod is meant to run a single instance of a given application. If you want to run multiple instances, you should use one Pod for each instance of the application. This is generally referred to as replication. Replicated Pods are created and managed as a group by a controller, such as a Deployment.

Pod termination

Pods terminate gracefully when their processes are complete. Kubernetes imposes a default graceful termination period of 30 seconds. When deleting a Pod you can override this grace period by setting the --grace-period flag to the number of seconds to wait for the Pod to terminate before forcibly terminating it.

Pod reliability using health check probes

Friday, January 15, 2021 12:04 AM

Configure Liveness, Readiness and Startup Probes:

There are 3 types of health check probes:

1. Liveness: The kubelet uses liveness probes to know when to restart a container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.
2. Readiness: The kubelet uses readiness probes to know when a container is ready to start accepting traffic. A Pod is considered ready when all of its containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.
3. Startup: The kubelet uses startup probes to know when a container application has started. If such a probe is configured, it disables liveness and readiness checks until it succeeds, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

Startup probes happen before any other probes, so you can afford to have a long timeout.

Configuring Probes:

A Probe is a diagnostic performed periodically by the kubelet on a Container. To perform a diagnostic, the kubelet calls a Handler implemented by the container. There are three types of handlers:

- **ExecAction:** Executes a specified command inside the container. The diagnostic is considered successful if the command exits with a status code of 0.
- **TCPSocketAction:** Performs a TCP check against the Pod's IP address on a specified port. The diagnostic is considered successful if the port is open.
- **HTTPGetAction:** Performs an HTTP GET request against the Pod's IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

Each probe has one of three results:

- Success: The container passed the diagnostic.
- Failure: The container failed the diagnostic.
- Unknown: The diagnostic failed, so no action should be taken.

Probes have a number of fields that you can use to more precisely control the behavior of liveness and readiness checks:

- initialDelaySeconds: Number of seconds after the container has started before liveness or readiness probes are initiated. Defaults to 0 seconds. Minimum value is 0.
- 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 and startup Probes. Minimum value is 1.
- failureThreshold: When a probe fails, Kubernetes will try failureThreshold times before giving up. Giving up in case of liveness probe means restarting the container. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

HTTP probes

HTTP probes have additional fields that can be set on httpGet:

- 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. Defaults to /.
- 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.

For an HTTP probe, the kubelet sends an HTTP request to the specified path and port to perform the check. The kubelet sends the probe to the pod's IP address, unless the address is overridden by the optional host field in httpGet. If scheme field is set to HTTPS, the kubelet sends an HTTPS request skipping the certificate verification.

TCP probes

For a TCP probe, the kubelet makes the probe connection at the node, not in the pod, which means that you can not use a service name in the host parameter since the kubelet is unable to resolve it.

Example:

```
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 1
  periodSeconds: 10
```

```
startupProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10

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

- When you have a deployment with some pods in running and other pods in the pending state, more often than not it is a problem with resources on the nodes.

Services, Networking, Storage

Monday, January 6, 2020 12:57 PM

Services:

1. Services expose a set of pods to the network
2. It assigns a fixed IP to your pods
3. 3 types of services:
   a. ClusterIP
   b. Node port
   c. Load balancer
4. Services can be accessed using labels assigned to the deployment.

There are four types of Kubernetes services:

1. **ClusterIP.** This default type exposes the service on a cluster-internal IP. You can reach the service only from *within* the cluster. IP provided is unique to that cluster, not any other VMs.
2. **NodePort.** This type of service exposes the service on each node's IP at a static port. A ClusterIP service is created automatically, and the NodePort service will route to it. From *outside* the cluster, you can contact the NodePort service by using "<NodeIP>:<NodePort>". The Node ip is given from the VPC subnet you have configured to that project. Hence any VM on the subnet can reach the pods.
3. **LoadBalancer.** This service type exposes the service externally using the load balancer of your cloud provider. The external load balancer routes to your NodePort and ClusterIP services, which are created automatically.
   a. Internal TCP/UDP Load Balancing makes your cluster's services accessible to applications outside of your cluster that use the same VPC network and are located in the same Google Cloud region. For example, suppose you have a cluster in the us-west1 region and you need to make one of its services accessible to Compute Engine VM instances running in that region on the same VPC network.
   b. Internal TCP/UDP Load Balancing creates an internal IP address for the Service that receives traffic from clients in the same VPC network and compute region. If you enable global access, clients in any region of the same VPC network can access the Service. In addition, clients in a VPC network connected to the LoadBalancer network using VPC Network Peering can also access the Service.
   c. Global access is an optional parameter for internal LoadBalancer Services that allows clients from any region in your VPC network to access the internal TCP/UDP load balancer. Without global access, traffic originating from clients in your VPC network must be in the same region as the load balancer. Global access allows clients in any region to access the load balancer. Backend instances must still be located in the same region as the load balancer.
   d. An internal TCP/UDP load balancer is not assigned a fully-qualified domain name (FQDN), and it is not possible to configure one using GKE. You can assign a static IP address to your internal TCP/UDP load balancer and assign that IP to a record in Cloud DNS, but Cloud DNS is not tightly integrated with GKE.
4. **ExternalName.** This type maps the service to the contents of the externalName field (e.g., foo.bar.example.com). It does this by returning a value for the CNAME record.

SVC or the service network, maintains the entire cluster. It can be reached on an ip which doesn't really exist since the SVC itself isn't an actual machine. Each pod has a kube-proxy (using linux IPVS) that enable it to reach the correct service whenever we need to reach the SVC. The SVC however knows each pod in the cluster and has labels on pod ip, healthy pods etc. Mostly pods are unique (I.e ip) and not repeated. Hence whenever we try to reach an app on a particular port, the svc itself knows which pod to reach.

Storage:

Memory stored in pods are not saved, hence storage volumes (LUNDS, devices, mounts, shared, spaces etc) are decoupled from pods. Kubernetes leaves actual storage requirements to the providers and only provides an interface I.e container storage interface (CSI) to work with storages. The CSI is an 'out-of-tree' module of kubernetes I.e its not part of the kubernetes main module (it used to be in-tree). CSI is fairly new and open-source. To use a persistent volume a pod needs to have a persistent volume claim (PVC).

A Persistent Volume can be mounted on a host in any way supported by the resource provider. As shown in the table below, providers will have different capabilities and each PV's access modes are set to the specific modes supported by that particular volume. For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only. Each PV gets its own set of access modes describing that specific PV's capabilities.

The access modes are:

- ReadWriteOnce – the volume can be mounted as read-write by a single node
- ReadOnlyMany – the volume can be mounted read-only by many nodes
- ReadWriteMany (RWM)– the volume can be mounted as read-write by many nodes

GCE doesn't doesn't support RWM.
Important! A volume can only be mounted using one access mode at a time, even if it supports many. For example, a GCEPersistentDisk can be mounted as ReadWriteOnce by a single node or ReadOnlyMany by many nodes, but not at the same time.

Storage classes allow you to create cluster connected volumes from the yml itself.
This is done by a process on gcp called the 'PV Subsystem control loop'. It keeps listening to the cluster to see if there is a pv claim from the kubernetes api I.e k8s-api-server. Once a claim is done, it creates the pv.

Ways to get persistent volume in pods:

- PersistentVolumeClaims: A PersistentVolumeClaim is a request for and claim to a PersistentVolume resource. PersistentVolumeClaim objects request a specific size, access mode, and StorageClass for the PersistentVolume. If a PersistentVolume that satisfies the request exists or can be provisioned, the PersistentVolumeClaim is bound to that PersistentVolume. Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for the Pod.
- Dynamically provisioning PersistentVolumes: Most of the time, you don't need to directly configure PersistentVolume objects or create Compute Engine persistent disks. Instead, you can create a PersistentVolumeClaim and Kubernetes automatically provisions a persistent disk for you.

Example:
1) Persistant volume claim request:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mysql-volumeclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 200Gi
```

2) Persistant volume claim binding:

```
    volumeMounts:
      - name: mysql-persistent-storage
        mountPath: /var/lib/mysql
  volumes:
    - name: mysql-persistent-storage
      persistentVolumeClaim:
        claimName: mysql-volumeclaim
```

Networking:

All nodes in a kubernete cluster need to talk to one another. Kubernetes doesn't enable the networking though GCP would do It by default.
To reach 1 cluster from another, create a service of type ClusterIP and point the second to the first. This ensure that the main cluster is not exposed to the internet.
You can expose your service with minimal exposure- ClusterIp (within k8s cluster) or larger exposure with NodePort (within cluster external to k8s cluster) or LoadBalancer (external world or whatever you defined in your LB).
`ClusterIp exposure < NodePort exposure < LoadBalancer exposure`

Resources type and tags

Tuesday, January 19, 2021 1:06 PM

Requests and limits

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its request for that resource specifies. However, a container is not allowed to use more than its resource limit.

For example, if you set a memory request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM.

If you set a memory limit of 4GiB for that Container, the kubelet (and container runtime) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel terminates the process that attempted the allocation, with an out of memory (OOM) error.

Limits can be implemented either reactively (the system intervenes once it sees a violation) or by enforcement (the system prevents the container from ever exceeding the limit). Different runtimes can have different ways to implement the same restrictions.

Resource types

CPU and *memory* are each a *resource type*. A resource type has a base unit. CPU represents compute processing and is specified in units of Kubernetes CPUs. Memory is specified in units of bytes. If you're using Kubernetes v1.14 or newer, you can specify *huge page* resources. Huge pages are a Linux-specific feature where the node kernel allocates blocks of memory that are much larger than the default page size.

For example, on a system where the default page size is 4KiB, you could specify a limit, hugepages-2Mi: 80Mi. If the container tries allocating over 40 2MiB huge pages (a total of 80 MiB), that allocation fails.

Meaning of CPU

Limits and requests for CPU resources are measured in *cpu* units. One cpu, in Kubernetes, is equivalent to **1 vCPU/Core** for cloud providers and **1 hyperthread** on bare-metal Intel processors.

Fractional requests are allowed.

A Container with spec.containers[].resources.requests.cpu of 0.5 is guaranteed half as much CPU as one that asks for 1 CPU. 1 CPU = 1000 millicores (1000m). The expression 0.1 is equivalent to the expression 100m, which can be read as "one hundred millicpu".

Some people say "one hundred millicores", and this is understood to mean the same thing. A request with a decimal point, like 0.1, is converted to 100m by the API, and precision finer than 1m is not allowed. For this reason, the form 100m might be preferred.

CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the

same amount of CPU on a single-core, dual-core, or 48-core machine.

Meaning of memory

Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent roughly the same value:

128974848, 129e6, 129M, 123Mi

Here's an example. The following Pod has two Containers. Each Container has a request of 0.25 cpu and 64MiB (226 bytes) of memory. Each Container has a limit of 0.5 cpu and 128MiB of memory. You can say the Pod has a request of 0.5 cpu and 128 MiB of memory, and a limit of 1 cpu and 256MiB of memory.

```
apiVersion: v1

kind: Pod

metadata:

 name: frontend

spec:

 containers:


- name: app

   image: images.my-company.example/app:v4

   resources:

     requests:

       memory: "64Mi"

       cpu: "250m"

      limits:
        memory: "128Mi"

       cpu: "500m"

- name: log-aggregator

 image: images.my-company.example/log-aggregator:v6

   resources:

    requests:

         memory: "64Mi"

      cpu: "250m"
```

```
limits:
    memory: "128Mi"
     cpu: "500m"
```

Types of deployments

Friday, January 10, 2020 5:09 PM

Below are 3 different resources that Kubernetes provides for deploying pods.

1. Deployments
2. StatefulSets
3. DaemonSets

Deployments

Deployment is the easiest and most used resource for deploying your application. It is a Kubernetes controller that matches the current state of your cluster to the desired state mentioned in the Deployment manifest. e.g. If you create a deployment with 1 replica, it will check that the desired state of ReplicaSet is 1 and current state is 0, so it will create a ReplicaSet, which will further create the pod. If you create a deployment with name **counter**, it will create a ReplicaSet with name **counter-<replica-set-id>**, which will further create a Pod with name **counter-<replica-set->-<pod-id>.**

Deployments are usually used for stateless applications. However, you can save the state of deployment by attaching a Persistent Volume to it and make it stateful, but all the pods of a deployment will be sharing the same Volume and data across all of them will be same.

StatefulSets

StatefulSet (stable-GA in k8s v1.9) is a Kubernetes resource used to manage stateful applications. It manages the deployment and scaling of a set of Pods, and provides guarantee about the ordering and uniqueness of these Pods.

StatefulSet is also a Controller but unlike Deployments, it doesn't create ReplicaSet rather itself creates the Pod with a unique naming convention. e.g. If you create a StatefulSet with name **counter,** it will create a pod with name **counter-0,** and for multiple replicas of a statefulset, their names will increment like **counter-0**, **counter-1**, **counter-2, etc**

Every replica of a stateful set will have its own state, and each of the pods will be creating its own PVC(Persistent Volume Claim). So a statefulset with 3 replicas will create 3 pods, each having its own Volume, so total 3 PVCs.

DaemonSet

A DaemonSet is a controller that ensures that the pod runs on all the nodes of the cluster. If a node is added/removed from a cluster, DaemonSet automatically adds/deletes the pod.

Some typical use cases of a DaemonSet is to run cluster level applications like:

- **Monitoring Exporters**: You would want to monitor all the nodes of your cluster so you will need to run a monitor on all the nodes of the cluster like NodeExporter.
- **Logs Collection Daemon:** You would want to export logs from all nodes so you would need a DaemonSet of log collector like Fluentd to export logs from all your nodes.

However, Daemonset automatically doesn't run on nodes which have a taint e.g. Master. You will have to specify the tolerations for it on the pod.

Taints are a way of telling the nodes to repel the pods i.e. no pods will be schedule on this node unless the pod tolerates the node with the same toleration. The master node is already tainted by:

```
taints:
- effect: NoSchedule
key: node-role.kubernetes.io/master
```

Which means it will repel all pods that do not tolerate this taint, so for daemonset to run on all nodes, you would have to add following tolerations on DaemonSet

```
spec:
    tolerations:
    - effect: NoSchedule
    operator: Exists
```

which means that it should tolerate all nodes.

Deployment, update and testing strategies

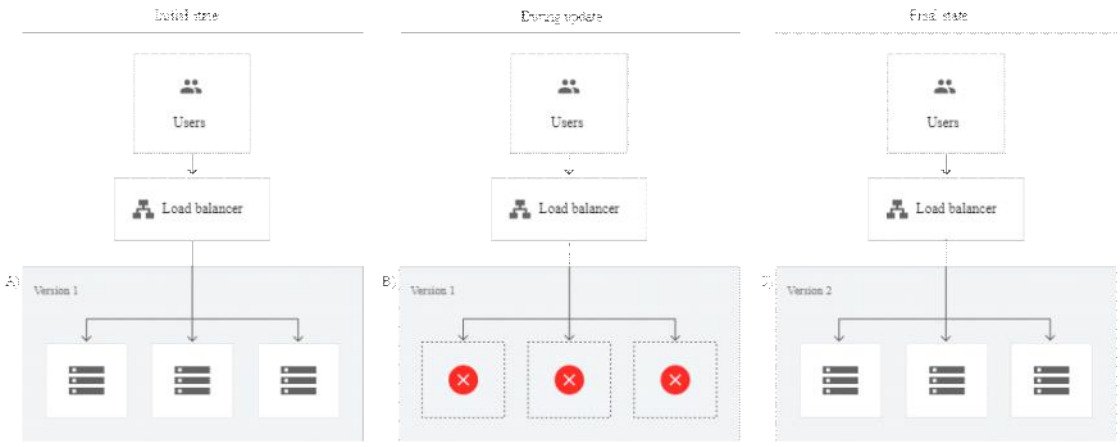
Friday, January 15, 2021 1:21 AM

Deployment or testing pattern	Zero downtime	Real production traffic testing	Releasing to users based on conditions	Rollback duration	Impact on hardware and cloud costs
Recreate Version 1 is terminated, and Version 2 is rolled out.	x	x	x	Fast but disruptive because of downtime	No extra setup required
Rolling update Version 2 is gradually rolled out and replaces Version 1.	✓	x	x	Slow	Can require extra setup for surge upgrades
Blue/green Version 2 is released alongside Version 1; the traffic is switched to Version 2 after it is tested.	✓	x	x	Instant	Need to maintain blue and green environments simultaneously
Canary Version 2 is released to a subset of users, followed by a full rollout.	✓	✓	x	Fast	No extra setup required
A/B Version 2 is released, under specific conditions, to a subset of users.	✓	✓	✓	Fast	No extra setup required
Shadow Version 2 receives real-world traffic without impacting user requests.	✓	✓	x	Does not apply	Need to maintain parallel environments in order to capture and replay user requests

When you *deploy* a service, it's not always exposed immediately to users. Sometimes, it's only after the service is *released* that users see changes in the application. However, when a service is *released in-place*, deployment and release occur simultaneously. In this case, when you deploy the new version, it starts accepting production traffic. Alternatively, there are deployment strategies for provisioning multiple service versions in parallel. These deployment patterns let you control and manage which version receives an incoming request.

- **Recreate deployment pattern**

With a recreate deployment, you fully scale down the existing application version before you scale up the new application version.
The following diagram shows how a recreate deployment works for an application.



Version 1 represents the current application version, and Version 2 represents the new application version. When you update the current application version, you first scale down the existing replicas of Version 1 to zero, and then you concurrently deploy replicas with the new version.

Key benefits

The advantage of the recreate approach is its simplicity. You don't have to manage more than one application version in parallel, and therefore you avoid backward compatibility challenges for your data and applications.

Considerations

The recreate method involves downtime during the update process. Downtime is not an issue for applications that can handle maintenance windows or outages. However, if you have mission-critical applications with high service level agreements (SLAs) and availability requirements, you might choose a different deployment strategy.

- **Rolling update deployment pattern**

In a rolling update deployment, you update a subset of running application instances instead of simultaneously updating every application instance, as the following diagram shows.

In this deployment approach, the number of instances that you update simultaneously is called the *window size*. In the preceding diagram, the rolling update has a window size of 1. One application instance is updated at a time. If you have a large cluster, you might increase the window size.

With rolling updates, you have flexibility in how you update your application:

- You can scale up the application instances with the new version before you scale down the old version (a process known as a *surge upgrade*).
- You can specify the maximum number of application instances that remain unavailable while you scale up new instances in parallel.

Key benefits

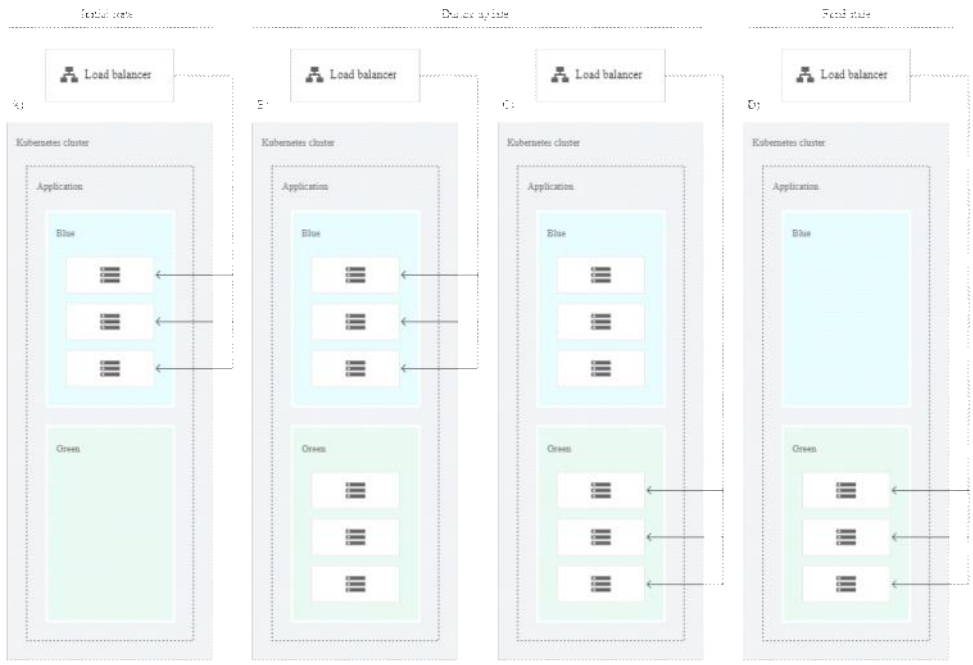
- **No downtime.** Based on the window size, you incrementally update deployment targets, for example, one by one or two by two. You direct traffic to the updated deployment targets only after the new version of the application is ready to accept traffic.
- **Reduced deployment risk.** When you roll out an update incrementally, any instability in the new version affects only a portion of the users.

Considerations

- **Slow rollback.** If the new rollout is unstable, you can terminate the new replicas and redeploy the old version. However, like a rollout, a rollback is a gradual, incremental process.
- **Backward compatibility.** Because new code and old code live side by side, users might be routed to either one of the versions arbitrarily. Therefore, ensure that the new deployment is backward compatible; that is, the new application version can read and handle data that the old version stores. This data can include data stored on disk, in a database, or as part of a user's browser session.
- **Sticky sessions.** If the application requires session persistence, we recommend that the load balancer supports stickiness and connection draining. Also, we recommend that you invoke session-sharing when possible (through session replication or session management using a datastore) so that the sessions can be decoupled from underlying resources.

- ## Blue/green deployment pattern

In a blue/green deployment (also known as a red/black deployment), you perform two identical deployments of your application, as the following diagram shows.



In the diagram, blue represents the current application version and green represents the new application version. Only one version is live at a time. Traffic is routed to the blue deployment while the green deployment is created and tested. After you're finished testing, you route traffic to the new version.

After the deployment succeeds, you can either keep the blue deployment for a possible rollback or decommission it. Alternatively, you can deploy a newer version of the application on these instances. In that case, the current (blue) environment serves as the staging area for the next release.

Key benefits

- **Zero downtime.** Blue/green deployment allows cutover to happen quickly with no downtime.
- **Instant rollback.** You can roll back at any time during the deployment process by adjusting the load balancer to direct traffic back to the blue environment. The impact of downtime is limited to the time it takes to switch traffic to the blue environment after you detect an issue.
- **Environment separation.** Blue/green deployment ensures that spinning up a parallel green environment doesn't affect resources that support the blue environment. This separation reduces your deployment risk.

Considerations

- **Cost and operational overhead.** Adopting the blue/green deployment pattern can increase operational overhead and cost because you must maintain duplicate environments with identical infrastructure.
- **Backward compatibility.** Blue and green deployments can share data points and datastores. We recommend that you verify that both versions of the application can use the schema of the datastore and the format of the records. This backward compatibility is necessary if you want to switch seamlessly between the two versions if you need to roll back.
- **Cutover.** If you plan to decommission the current version, we recommend that you allow for appropriate connection draining on existing transactions and sessions. This step allows

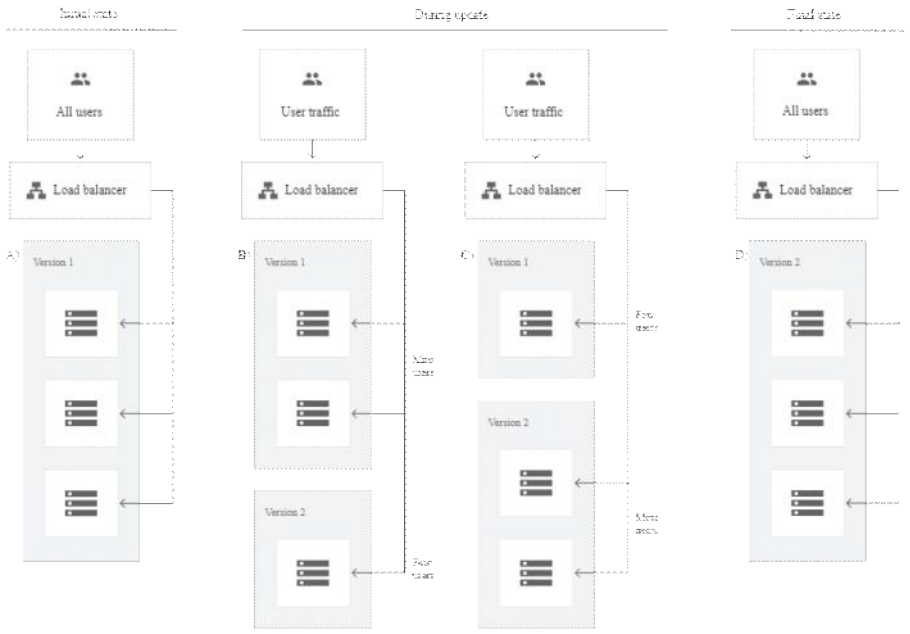
requests processed by the current deployment to be completed or terminated gracefully.

Testing strategies

The testing patterns discussed in this section are typically used to validate a service's reliability and stability over a reasonable period under a realistic level of concurrency and load.

Canary test pattern

In canary testing, you partially roll out a change and then evaluate its performance against a baseline deployment, as the following diagram shows.



In this test pattern, you deploy a new version of your application alongside the production version. You then split and route a percentage of traffic from the production version to the canary version and evaluate the canary's performance.

You select the key metrics for the evaluation when you configure the canary. We recommend that you compare the canary against an equivalent baseline and not the live production environment.

To reduce factors that might affect your analysis (such as caching, long-lived connections, and hash objects), we recommend that you take the following steps for the baseline version of your application:

- Ensure that the baseline and production versions of your application are identical.
- Deploy the baseline version at the same time that you deploy the canary.
- Ensure that the baseline deployment (such as the number of application instances and autoscaling policies) matches the canary deployment.
- Use the baseline version to serve the same traffic as the canary.

In canary tests, partial rollout can follow various partitioning strategies. For example, if the application has geographically distributed users, you can roll out the new version to a region or a specific location first. For more information, see Automating canary analysis on GKE with Spinnaker and best practices for configuring a canary.

Key benefits

- **Ability to test live production traffic.** Instead of testing an application by using simulated traffic in a staging environment, you can run canary tests on live production traffic. With canary rollouts, you need to decide in what increments you release the new application and when you trigger the next step in a release. The canary needs enough traffic so that monitoring can clearly detect any problems.
- **Fast rollback.** You can roll back quickly by redirecting the user traffic to the older version of the application.
- **Zero downtime.** Canary releases let you route the live production traffic to different versions of the application without any downtime.

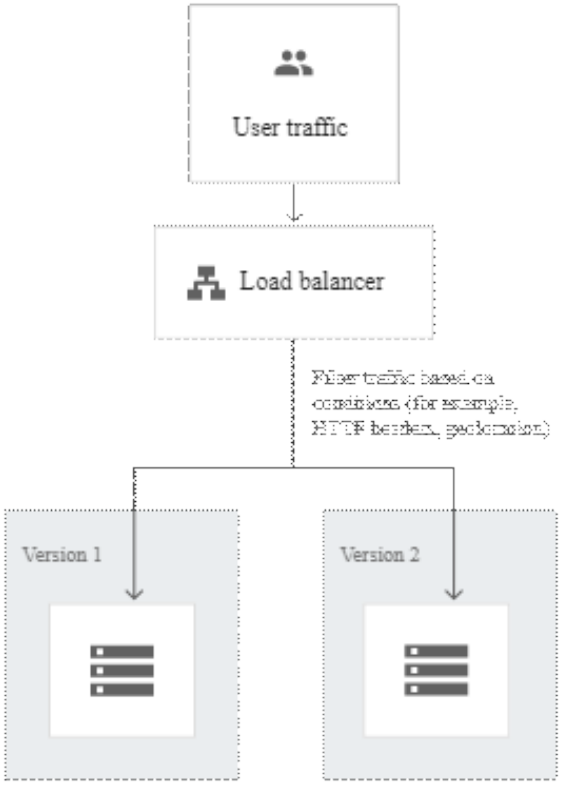
Considerations

- **Slow rollout.** Each incremental release requires monitoring for a reasonable period and, as a result, might delay the overall release. Canary tests can often take several hours.
- **Observability.** A prerequisite to implementing canary tests is the ability to effectively observe and monitor your infrastructure and application stack. Implementing robust monitoring can require a substantial effort.
- **Backward compatibility and sticky sessions.** As with rolling updates, canary testing can pose risks with backward incompatibility and session persistence because multiple application versions run in the environment while the canary is deployed.

A/B test pattern

With A/B testing, you test a hypothesis by using variant implementations. A/B testing is used to make business decisions (not only predictions) based on the results derived from data.

When you perform an A/B test, you route a subset of users to new functionality based on routing rules, as the following diagram shows.



Routing rules often include factors such as browser version, user agent, geolocation, and operating system. After you measure and compare the versions, you update the production environment with the version that yielded better results.

Key benefits

A/B testing is best used to measure the effectiveness of functionality in an application. Use cases for the deployment patterns discussed earlier focus on releasing new software safely and rolling back predictably. In A/B testing, you control your target audience for the new features and monitor any statistically significant differences in user behavior.

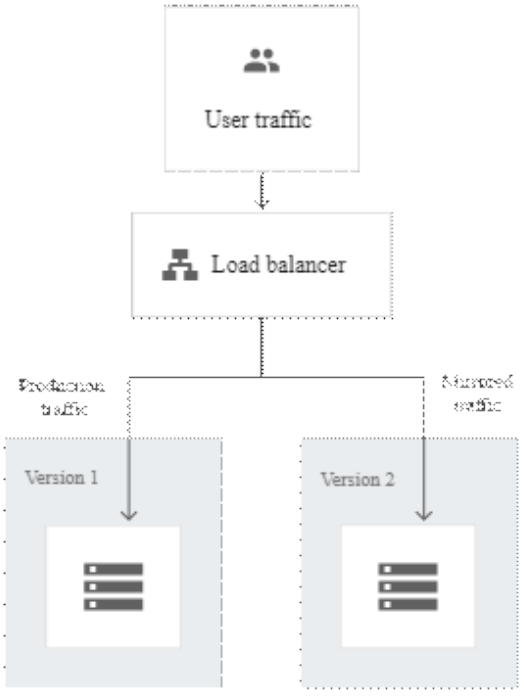
Considerations

- **Complex setup.** A/B tests need a representative sample that can be used to provide evidence that one version is better than the other. You need to pre-calculate the sample size (for example, by using an A/B test sample size calculator) and run the tests for a reasonable period to reach statistical significance of at least 95%.
- **Validity of results.** Several factors can skew the test results, including false positives, biased sampling, or external factors (such as seasonality or marketing promotions).
- **Observability.** When you run multiple A/B tests on overlapping traffic, the processes of monitoring and troubleshooting can be difficult. For example, if you test product page A versus product page B, or checkout page C versus checkout page D, distributed tracing becomes important to determine metrics such as the traffic split between versions.

Shadow test pattern

Sequential experiment techniques like canary testing can potentially expose customers to an inferior application version during the early stages of the test. You can manage this risk by using offline techniques like simulation. However, offline techniques do not validate the application's improvements because there is no user interaction with the new versions.

With shadow testing, you deploy and run a new version alongside the current version, but in such a way that the new version is hidden from the users, as the following diagram shows.



An incoming request is mirrored and replayed in a test environment. This process can happen either in real time or asynchronously after a copy of the previously captured production traffic is replayed against the newly deployed service.

You need to ensure that the shadow tests do not trigger side effects that can alter the existing production environment or the user state.

Key benefits

- **Zero production impact.** Because traffic is duplicated, any bugs in services that are processing shadow data have no impact on production.
- **Testing new backend features by using the production load.** When used with tools such as Diffy, traffic shadowing lets you measure the behavior of your service against live production traffic. This ability lets you test for errors, exceptions, performance, and result parity between application versions.
- **Reduced deployment risk.** Traffic shadowing is typically combined with other approaches like canary testing. After testing a new feature by using traffic shadowing, you then test the user experience by gradually releasing the feature to an increasing number of users over time. No full rollout occurs until the application meets stability and performance requirements.

Considerations

- **Side effects.** With traffic shadowing, you need to be cautious in how you handle services that mutate state or interact with third-party services. For example, if you want to shadow test the payment service for a shopping cart platform, the customers could pay twice for their order. To avoid shadow tests that might result in unwanted mutations or other risk-prone interactions, we recommend that you use either stubs or virtualization tools such as Hoverfly instead of third-party systems or datastores.
- **Cost and operational overhead.** Shadow testing is fairly complex to set up. Also, like blue/green deployments, shadow deployments carry cost and operational implications because the setup requires running and managing two environments in parallel.

Choosing the right strategy

You can deploy and release your application in several ways. Each approach has advantages and disadvantages. The best choice comes down to the needs and constraints of your business. Consider the following:

- What are your most critical considerations? For example, is downtime acceptable? Do costs constrain you? Does your team have the right skills to undertake complex rollout and rollback setups?
- Do you have tight testing controls in place, or do you want to test the new releases against production traffic to ensure the stability of the release and limit any negative impact?
- Do you want to test features among a pool of users to cross-verify certain business hypotheses? Can you control whether targeted users accept the update? For example, updates on mobile devices require explicit user action and might require extra permissions.
- Are microservices in your environment fully autonomous? Or, do you have a hybrid of microservice-style applications working alongside traditional, difficult-to-change applications? For more information, see deployment patterns on hybrid and multi-cloud

environments.

- Does the new release involve any schema changes? If yes, are the schema changes too complex to decouple from the code changes?

Best practices

In order to keep deployment and testing risks to a minimum, application teams can follow several best practices:

- **Backward compatibility.** When you run multiple application versions at the same time, ensure that the database is compatible with all active versions. For example, a new release requires a schema change to the database (such as a new column). In such a scenario, you need to change the database schema so that it's backward compatible with the older version. After you complete a full rollout, you can remove support for the old schema, leaving support only for the newest version. One way to achieve backward compatibility is to decouple schema changes from the code changes. For more information, see parallel change and database refactoring patterns.
- **Continuous integration/continuous deployment (CI/CD).** CI ensures that code checked into the feature branch merges with its main branch only after it successfully passes dependency checks, unit and integration tests, and the build process. Therefore, every change to an application is tested before it can be deployed. With CD, the CI-built code artifact is packaged and ready to be deployed in one or more environments. For more information, see building a CI/CD pipeline with Google Cloud.
- **Automation.** If you continuously deliver application updates to the users, we recommend that you build an automated process that reliably builds, tests, and deploys the software. We also recommend that your code changes automatically flow through a CI/CD pipeline that includes artifact creation, unit testing, functional testing, and production rollout. By using automation tools such as Spinnaker, Jenkins, TravisCI, and Cloud Build, you can automate the deployment processes to be more efficient, reliable, and predictable.
- **Operating environments and configuration management.** Tools like Vagrant and Packer can help you maintain consistent local development, staging, and production environments. You can also use configuration management tools like Puppet, Chef, or Ansible to automatically apply OS settings or apply patches in target servers. For more information, see building custom images with Jenkins and Packer for virtual machines and containers.
- **Rollback strategy.** Create a rollback strategy to follow in the case that something goes wrong. Release automation tools like Spinnaker and Harness support rollbacks, or you can keep a backup of the last application version until you know the new one works properly.
- **Post-deployment monitoring.** An application performance management tool can help your team monitor critical performance metrics. Create a process for alerting the responsible team when a build or deployment fails. Enable automated rollbacks for deployments that fail health checks, whether because of availability or error rate issues.

Autoscaling

Monday, January 13, 2020 5:13 PM

For autoscaling kubernetes has 2 modes: Cluster autoscaling and horizontal pod autoscaler (HPA). Vertical pod autoscaler is still in 'alpha' stage of development.

HPA work on ACTUAL metric of the pods.
Cluster autoscaling works on the REQUESTED metric of the pods.

HPA: The hpa object is specified in the yml doc to keep an eye on a particular metric of the pod. The hpa DOESN'T do the scaling itself, it just knows when to do it and how to do it by editing the specs yml file. It does this by working on the minReplica and maxReplica objects of the yml file. There is always a cooldown period before hpa decides the extra pods are no longer required, and will delete extra pods.

Horizontal Pod Autoscaling can not be enabled for Daemon Sets, this is because there is only one instance of a pod per node in the cluster. In a replica deployment, when Horizontal Pod Autoscaling scales up, it can add pods to the same node or another node within the cluster. Since there can only be one pod per node in the Daemon Set workload, Horizontal Pod Autoscaling is not supported with Daemon Sets.

Example:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: myapp-hpa
spec:
  maxReplicas: 20
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: myapp-deployment
  targetCPUUtilizationPercentage: 60
```

Cluster autoscaling: Only works when you configure the pods in a node with 'resource requests' in the yml file. If deployed metric are less than request, cluster autoscaling comes in place. Wakes every 10seconds. Different cloud networks have different support for this. Cluster scaling works by created new nodes if the existing node's resources are used by the pods in it, and the utilization still isn't reducing.

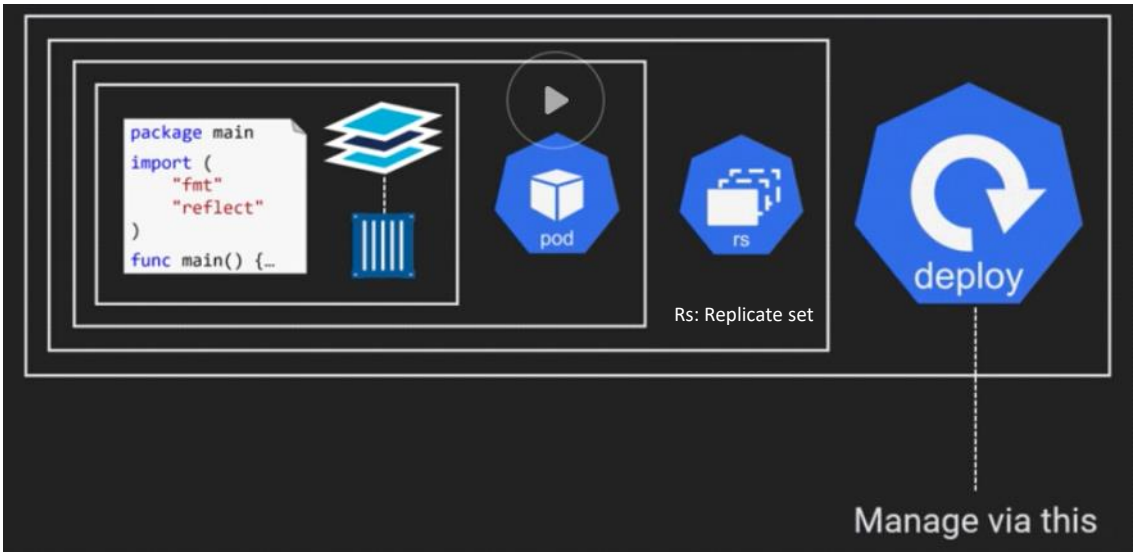
Config location:


Enable autoscaling
Minimum number of nodes *
1
Maximum number of nodes *
3

Deployment codes

Monday, January 13, 2020 4:49 PM

Labels connect deployment sets to pods. Prefer making changed to the pods using deployment sets.
To make any changes, edit the yml file and repost it, this makes things declarative and easier to manage.




```
apiVersion: apps/v1
                kind: Deployment #Means its an deployment api code or rules for deployment
                metadata:
                labels:
                run: prod-redis
                name: prod-redis
                spec:
                selector:
                matchLabels:
                run: prod-redis
                replicas: 3 #3 replicas of the pods with replica sets to take care of scaling
                minReadySeconds: 300 #Wait min 300secs after each update
                strategy:
                rollingUpdate:
                maxSurge: 1 #Max number of pods that can be updated at any point of the update process I.e 1 pod at a time will be updated
                maxUnavailable: 0 #Max allowed downtime pods
                type: RollingUpdate #Default option
                template:
                metadata:
                labels:
```

```
run: prod-redis
spec: #State specs I.e desired state
containers:
- image: redis:4.0
name: redis
```

Best practices

Monday, January 20, 2020 11:18 AM

1. What are containers good for?
    a. Stateless apps: Apps that don't need to save data to local disk
    b. Frontend systems where traffic can peak and fall
    c. Certain backend logic systems
2. What are containers bad for?
    a. Apps that need to save data to disk since providing mountable storage services to each pod can be tricky
    b. Non-scaling resource intensive monoliths
    c. Apps that require more human manual steps in-between
    d. Databases, since its more complex to implement scalably
3. To resize a cluster's node pools, run the following command:
    - gcloud container clusters resize **[CLUSTER_NAME]** --node-pool **[POOL_NAME]** \ --num-nodes **[NUM_NODES]**
    - Sample codes:
    - https://kubernetes.io/docs/tutorials/services/source-ip/#source-ip-for-services-with-type-clusterip
    - https://riptutorial.com/kubernetes/example/28083/hello-world

- **Configurations and Secrets:**
  Configurations allow you to push configs to pods in a cluster.
  Config maps allow you to create multiple key:value pairs to send to an application in a cluster. However avoid putting sensitive information as it is not encrypted. To push sensitive info like database pwds, use the kubectl command to create a **SECRET** with the pwd and then push it to the pods.

- **DaemonSet** controller ensures that a copy of a pod runs on nodes in the cluster, allowing for node management
    - **Deployment** controller is used to achieve a desired state of pods.

- **GKE Sandbox** provides an extra layer of security to prevent untrusted code from affecting the host kernel on your cluster nodes when containers in the Pod execute unknown or untrusted code. Multi-tenant clusters and clusters whose containers run untrusted workloads are more exposed to security vulnerabilities than other clusters. Examples include SaaS providers, web-hosting providers, or other organizations that allow their users to upload and run code. When you enable GKE Sandbox on a node pool, a sandbox is created for each Pod running on a node

in that node pool. In addition, nodes running sandboxed Pods are prevented from accessing other Google Cloud services or cluster metadata. Each sandbox uses its own userspace kernel. With this in mind, you can make decisions about how to group your containers into Pods, based on the level of isolation you require and the characteristics of your applications.
Ref: https://cloud.google.com/kubernetes-engine/docs/concepts/sandbox-pods

- Node auto-upgrades help you keep the nodes in your cluster up to date with the cluster master version when your master is updated on your behalf. When you create a new cluster or node pool with Google Cloud Console or the gcloud command, node auto-upgrade is enabled by default.
  - `gcloud container node-pools update node-pool-name --cluster cluster-name \ --zone compute-zone --enable-autoupgrade`
- Container-Optimized OS comes with the Docker container runtime and all Kubernetes components pre-installed for out of the box deployment, management, and orchestration of your containers. But these do not help with automatically upgrading GKE cluster versions.
  - This is useful if you want to directly deploy your containers to a managed instance template.

Commands [ref]

Tuesday, January 19, 2021 1:13 AM

Deployment related:

- gcloud container clusters get-credentials standard-cluster-1 --zone=us-central1-a : To initiate kubectl functionality from gcloud shell on that cluster named 'standard-cluster-1'
- docker build -r myapp . : Command to build new image from code. The '.' specifies current dir.
- docker tag myapp gcr.io/gkelabs/myapp:blue : Adding tags to your docker image. Docker tags convey useful information about a specific image version/variant. They are aliases to the ID of your image which often look like this: f1477ec11d12. It's just a way of referring to your image. A good analogy is how Git tags refer to a particular commit in your history.
  - Docker tag rmi <name> will delete any tagged images
- docker tag myapp gcr.io/gkelabs/myapp:blue : Command to push your image to GCP's container registery. Gcr.io/ is mandatory to do this, hence name the tag accordingly.
- kubectl apply -f myapp-deployment.yaml : Deploy using kubectl and deployment.yml file.
  - Kubectl –f apply myapp1.yaml, myapp2.yaml : To deploy multiple files
- Kubectl get pods/services : Gives info on pods and services created.
- kubectl rollout undo deployment.v1.apps/myapp-deployment : Undo a new rolled out version of your image. This will roll back the containers being used to the older ones used previously.
- kubectl expose deployment nginx --port 80 --type=LoadBalancer : Expose deployed app to public internet on port 80 using load balancer as type of network.
  - Additionally use –target-port 8080 to hit particular internal port (if not specified in the deployment yaml)
- kubectl get pods -l "app=prod" : This command correctly lists pods that have the label app=prod. When creating the deployment, we used the label app=prod so listing pods that have this label retrieve the pods belonging to nginx deployments. You can list pods by using Kubernetes CLI – kubectl get pods.
- kubectl get pod [POD_NAME] -o yaml : Get pod manifest to see its phases and condition
- kubectl exec --stdin --tty <container-name> -- /bin/bash : To ssh into a running container
  - Note: The short options -i and -t are the same as the long options --stdin and –tty
- gcloud container clusters resize NAME (--num-nodes=NUM_NODES | --size=NUM_NODES) [--async] [--node-pool=NODE_POOL] [--region=REGION | --zone=ZONE, -z ZONE] [GCLOUD_WIDE_FLAG …] : resizes an existing cluster for running containers.
  - When increasing the size of a container cluster, the new instances are created with the same configuration as the existing instances. Existing pods are not moved onto the new instances, but new pods (such as those created by resizing a replication controller) will be scheduled onto the new instances.
  - When decreasing a cluster, the nodes are drained. As a result, the pods running on these nodes are gracefully terminated. If your pods are being managed by a workload controller, the controller will attempt to reschedule them onto the remaining instances. If your pods are not managed by a workload controller, they will not be restarted. Note that when resizing down, instances running pods and instances without pods are not differentiated. Resize will pick instances to remove at random.

- gcloud container node-pools update node-pool-name --cluster cluster-name \ --zone compute-zone --enable-autoupgrade : Keep Node upto date

- kubectl get svc/services -o jsonpath='{.items[*].status.loadBalancer.ingress[0].ip}' **:** Get svc returns data and jsonpath formats it in the correct way

- kubectl get service --all-namespaces : Get service (not pod) ip

- gcloud container clusters create **cluster-name** --zone **compute-zone** \
  --enable-autorepair : Enable auto repair on that cluster or replace with node-pool too.

- kubectl set <image > deployment: You can perform a rolling update to update the images, configuration, labels, annotations, and resource limits/requests of the workloads in your clusters. Rolling updates incrementally replace your resource's Pods with new ones, which are then scheduled on nodes with available resources. Rolling updates are designed to update your workloads without downtime.
  You can use kubectl set to make changes to an object's image, resources (compute resource such as CPU and memory), or selector fields.
  For example, to update a Deployment from nginx version 1.7.9 to 1.9.1, run the following command:
  kubectl set image deployment nginx nginx=nginx:1.9.1
  The kubectl set image command updates the nginx image of the Deployment's Pods one at a time.

`kubectl autoscale deployment my-app --max 6 --min 4 --cpu-percent 50` : To autoscale kubernetes app

- The max limit is 300,000 containers per GKE cluster

- To submit a docker image to cloud run you can use either docker build or:
  - gcloud builds submit --config=[CONFIG_FILE_PATH] --gcs-log-dir=[GCS_LOG_DIR] [SOURCE] : This command correctly builds the container image, pushes the image to GCR (Google Container Registry) and uploads the build logs to Google Cloud Storage. --config flag specifies the YAML or JSON file to use as the build configuration file.
    - --gcs-log-dir specifies the directory in Google Cloud Storage to hold build logs. [SOURCE] is the location of the source to build.