Google Cloud

Partner Certification Academy

Professional Cloud Developer

The information in this presentation is classified:

Google confidential & proprietary

⚠ This presentation is shared with you under NDA.

- Do **not** record or take screenshots of this presentation.
- Do **not** share or otherwise distribute the information in this presentation with anyone **inside** or **outside** of your organization.

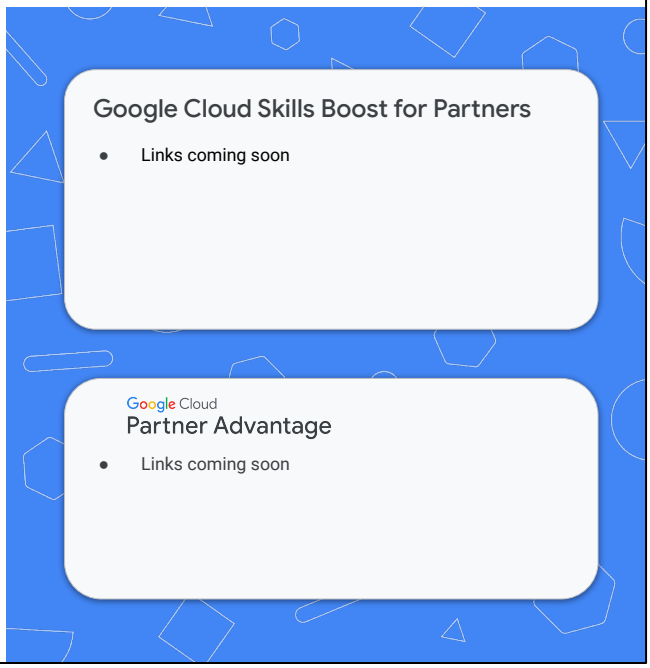
Thank you!

Google Cloud

Source Materials

Some of this program's content has been sourced from the following resources:

- Google Cloud certification site
- Google Cloud documentation
- Google Cloud console
- Google Cloud courses and workshops
- Google Cloud white papers
- Google Cloud Blog
- Google Cloud YouTube channel
- Google Cloud samples
- Google codelabs
- Google Cloud partner-exclusive resources

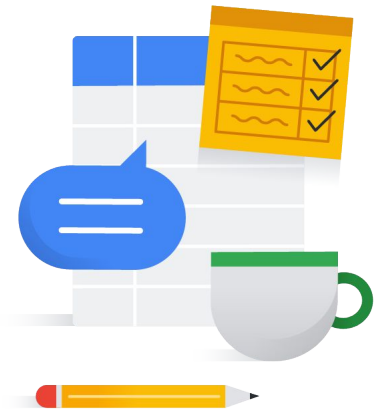


Session logistics

- When you have a question, please:
  - Click the Raise hand button in Google Meet.
  - Or add your question to the Q&A section of Google Meet.
  - Please note that answers may be deferred until the end of the session.
- These slides are available in the Student Lecture section of your Qwiklabs classroom.
- The session is **not recorded.**
- Google Meet does not have persistent chat.
  - If you get disconnected, you will lose the chat history.
  - Please copy any important URLs to a local text file as they appear in the chat.

Program issues or concerns?

- For questions regarding Cloud Skills Boost access, Qwiklabs issues, voucher queries, etc.
  - cloud-partner-training@google.com
- For questions regarding Partner Advantage access
  - https://support.google.com/googlecloud/topic/9198654



Partner Certification Academy

A differentiated learning experience for the busy professional

Our goal is to help you prepare for Google Cloud certification exams

These programs may include:

- On-demand learning
- Self-paced labs
- Mentor-led workshops
- A voucher for the exam

The workshop sessions:

- **Are NOT training sessions - that's the purpose of the on-demand content.**
- Help you review key concepts on the exam guide.
- Will NOT discuss actual exam questions.

Professional Cloud Developer (PCD)

A Professional Cloud Developer builds scalable and highly available applications using Google-recommended tools and best practices. This individual has experience with cloud-native applications, developer tools, managed services, and next-generation databases. A Professional Cloud Developer also has proficiency with at least one general-purpose programming language and instruments their code to produce metrics, logs, and traces.

The Professional Cloud Developer exam assesses your ability to:

- ✓ Design highly scalable, available, reliable cloud-native applications
- ✓ Deploy applications
- ✓ Manage deployed applications
- ✓ Build and test applications
- ✓ Integrate Google Cloud services

https://cloud.google.com/learn/certification/cloud-developer

Module Agenda

01 Overview of deployment & testing patterns

02 Managed Instance Group deployment patterns

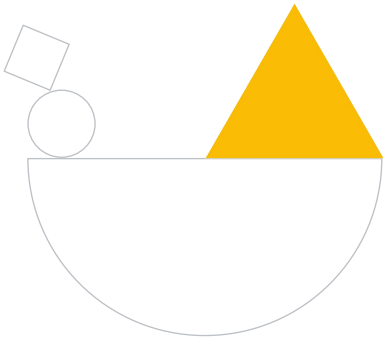
03 Google Kubernetes Engine deployment patterns

04 Cloud Run deployment patterns

05 Patterns for scalable and resilient apps

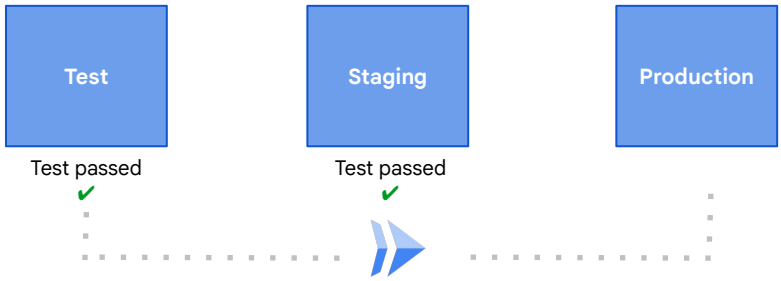
06 Container Best Practices & Security Command Center

Overview of deployment & testing patterns



Best practice: Have multiple, similar environments for deployment testing

- Most organizations have a minimum 3 separate environments
  - Test/development, staging and production
- Create these to be as similar as possible so that changes can be tested adequately before going into production
    - Goal: To deploy new versions with zero downtime

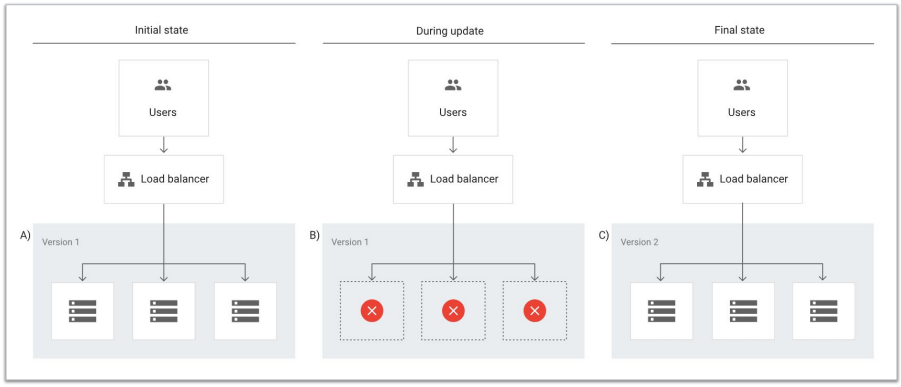


Types of deployments and testing strategies

- Common types of deployments are
  - Recreate
  - Rolling
  - Blue/Green
- Testing strategies
  - Canary
  - A/B
  - Shadow
- Each are discussed in the following slides

Recreate deployment pattern

- Fully scale down the existing application version before you scale up the new application version
- Key Benefit
  - Simplicity
- Considerations
  - Downtime



Application deployment and testing strategies

From:
https://cloud.google.com/architecture/application-deployment-and-testing-strategies

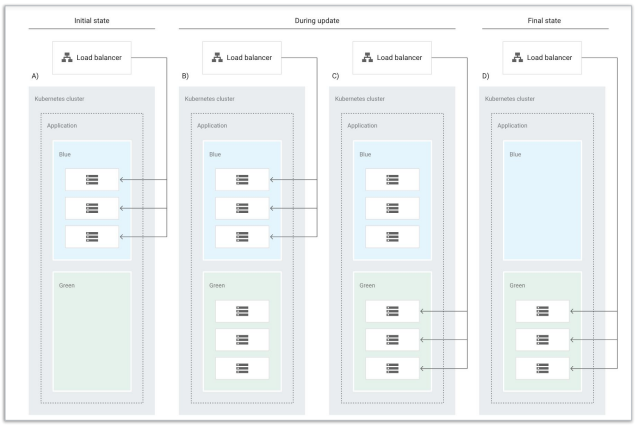
Rolling update deployment pattern

- Update a subset of running application instances instead of simultaneously updating every application instance
- Key Benefits
  - No downtime
  - Reduced deployment risk
    - Any instability in the new version affects only a portion of the users
- Considerations
  - Slow rollback
    - Must terminate the new replicas and redeploy the old version

[Application deployment and testing strategies]

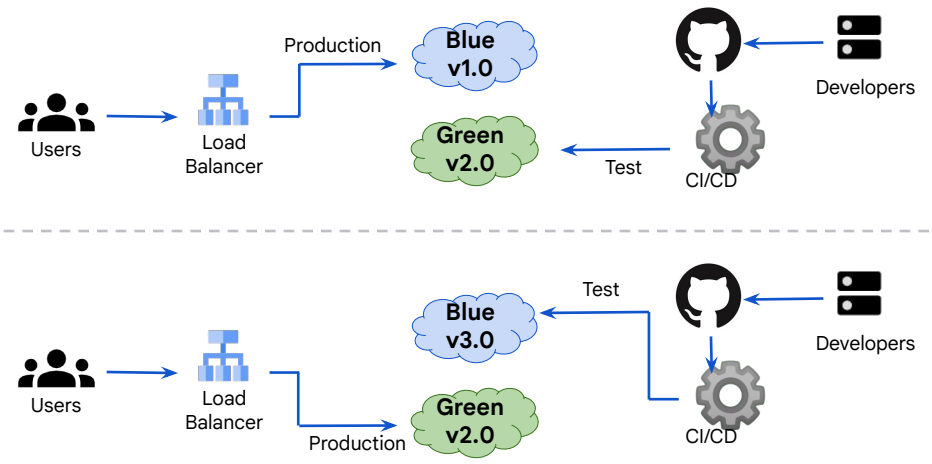
Blue/Green (Red/Black) deployment pattern

- Have two identical deployments of your application
- Blue version
  - Live version running in production
    - All production traffic goes here
- Green version
  - Newly updated version
  - Testing traffic goes here
- Upon successful completion of tests
  - Route production traffic to the Green version
  - If an issue arises, direct traffic back to Blue



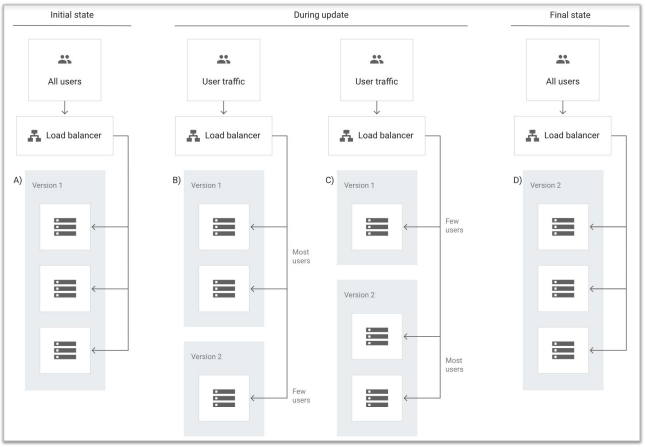
[Application deployment and testing strategies]

Blue/Green deployment pattern



Canary test pattern

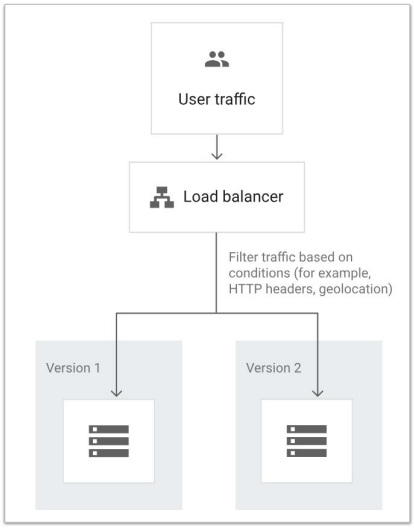
- Deploy a new version of your application alongside the ***production*** version
  - Split and route a percentage of traffic from production to the canary version and evaluate the canary's performance
- Benefits
  - Ability to test live production traffic
  - Fast rollback
  - Zero downtime



[Application deployment and testing strategies]

A/B test pattern

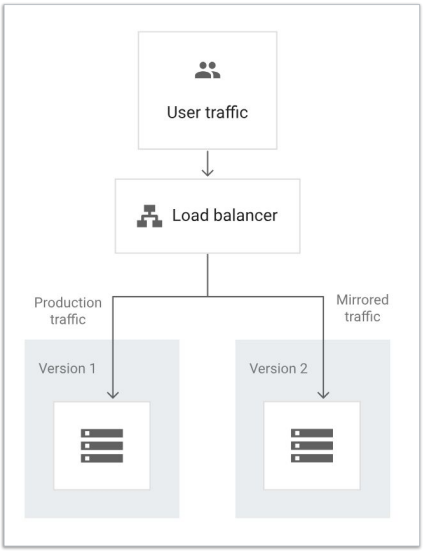
- Used to measure the effectiveness of a change in functionality in an application
  - Route a target audience to the new version
    - Based on routing rules such as location, browser version, etc.
  - Monitor any statistically significant differences in user behavior
- Benefits
  - Able to gain feedback on a new version prior to it going into production



Application deployment and testing strategies

Shadow test pattern

- Deploy and run a new version alongside the current version
- New version does not handle live traffic
  - Incoming requests are mirrored and replayed in a test environment
  - Can occur in real time or asynchronously after capturing production traffic and replaying it against the new service
- Benefits
  - Zero production impact
  - Testing new features using production load
  - Reduced deployment risk



[Application deployment and testing strategies]

Choosing the right strategy

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.	✘	✘	✘	Fast but disruptive due to downtime	No extra setup required
Rolling update Version 2 is gradually rolled out and replaces Version 1.	✓	✘	✘	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.	✓	✘	✘	Instant	Need to maintain blue and green environments simultaneously
Canary Version 2 is released to a subset of users, followed by a full rollout.	✓	✓	✘	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.	✓	✓	✘	Does not apply	Need to maintain parallel environments in order to capture and replay user requests

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. You should consider the following when choosing the right approach.

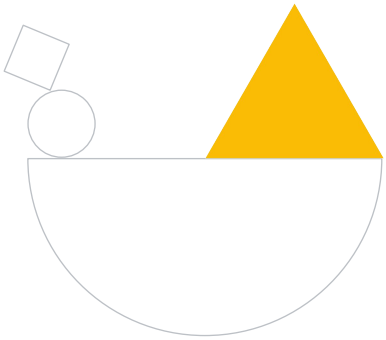
What are your most critical considerations? For example, is downtime acceptable?

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, refer to 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?

The table shown here summarizes the salient characteristics of the deployment and testing patterns. When you weigh the advantages and disadvantages of various deployment and testing approaches, consider your business needs and technological resources, and then select the option that benefits you the most.

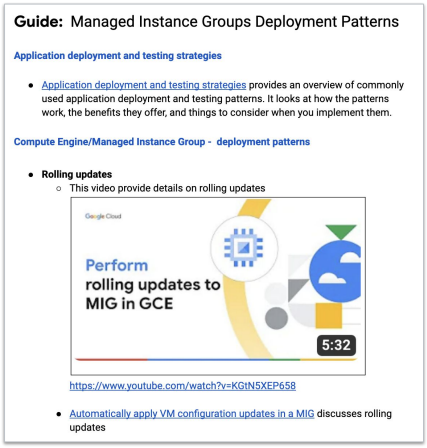
Managed Instance Group deployment patterns



Guide: Managed Instance Group Deployment Patterns

Topics covered include

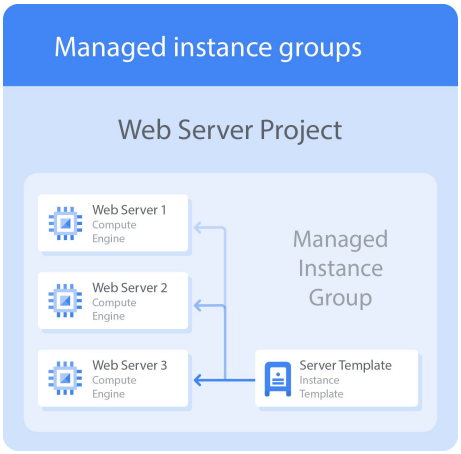
- Overview of deployment patterns
- Rolling Updates
- Traffic splitting
  - Blue/Green
  - Canary
  - A/B



Managed Instance Group Deployment Patterns

Review: Managed instance groups create VMs based on templates

- Instance templates define the VMs: image, machine type, etc.
  - Test to find the smallest machine type that will run your program
- Instance group manager creates the machines
- Optimize cost and meet varying user workloads via autoscaling
- Enable auto healing via health checks
- Can be single zone or regional
  - Use multiple zones for high availability



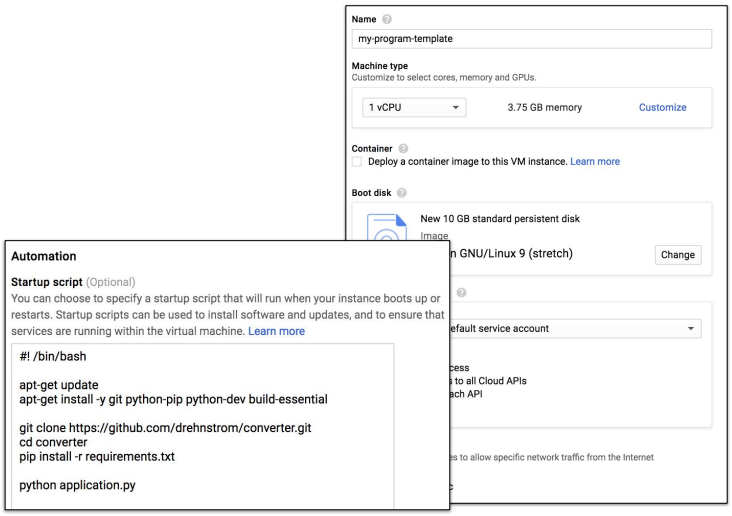
Managed instance groups create VMs based on instance templates. Instance templates are just a resource used to define VMs and managed instance groups. The templates define the boot disk image or container image to be used, the machine type, labels, and other instance properties like a startup script to install software from a Git repository.

The virtual machines in a managed instance group are created by an instance group manager. Using a managed instance group offers many advantages, such as autohealing to re-create instances that don't respond and creating instances in multiple zones for high availability.

Review: Creating an Instance Template

Contains the information required to build a VM for a deployment

- All VMs in a MIG use the same template
- Can include a startup script to automate code deployment
- Can also use a custom image



https://cloud.google.com/compute/docs/instance-templates

Google Cloud

Creating an instance template is similar to creating a virtual machine.

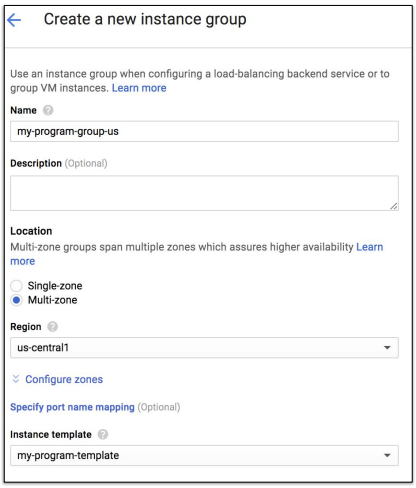
The template will include all information required to build the instance.

The information provided to build the template can include startup scripts to automate code deployment if needed.

Review: Creating an Instance Group

Instance groups create VMs based on instance templates

- Specify how many machines to create
- In which region
- Based on what template
- Add an autoscaler to add/remove instances based on demand
- Add a health check to ensure applications are responsive



An instance group enables you to manage a group of virtual machines as a single entity.

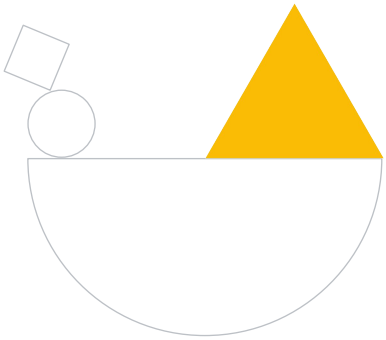
All virtual machines in a managed instance group are built from the same template.

When building the group, you specify:

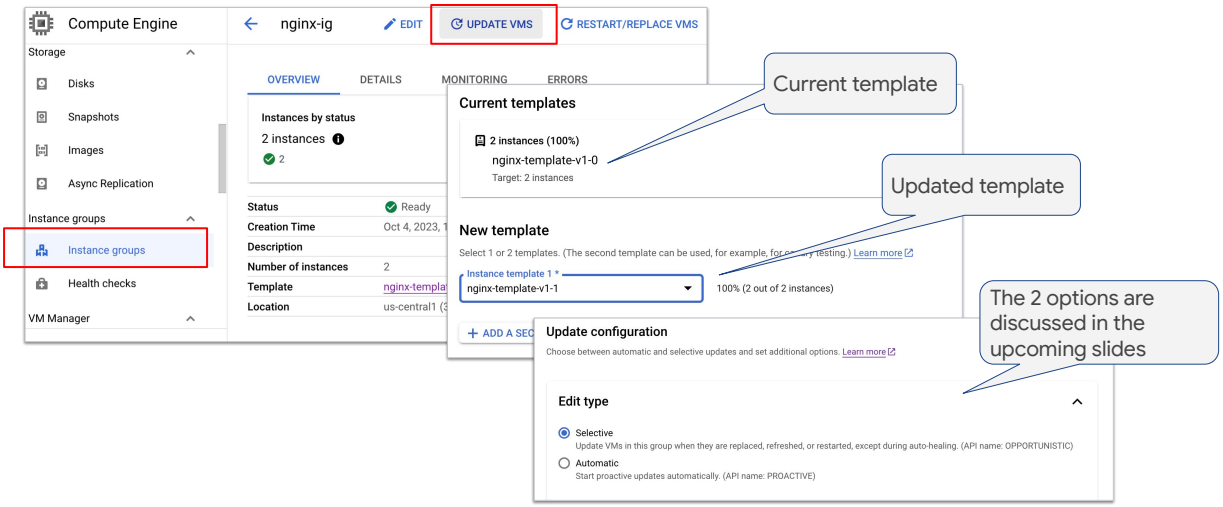
- Number and location of instances
- Template to build from
- Metric or metrics to use to scale instances
- Health check to verify machines are working properly

Managed Instance Group deployment patterns

- **Rolling Update**
- Recreate strategy
- Load balancing review
- Blue/Green
- Canary testing
- A/B testing
- Shadow testing



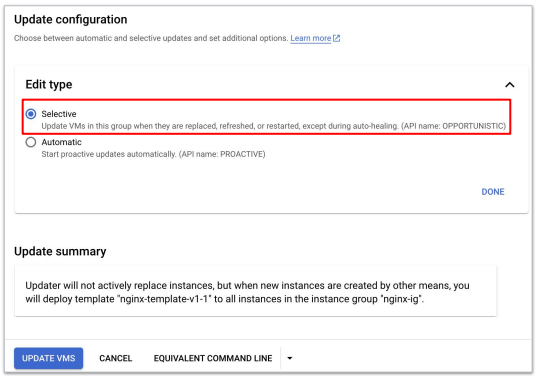
Performing a rolling update using the Console



[Apply new VM configurations in a MIG]

Performing a selective/opportunistic update with the Console

- Select the new template
- Choose "`Selective`" for the "`Edit type`"
- New configuration is applied selectively, either manually or automatically
- Example of an automatic update
  - Applying a non-critical update to an autoscaled MIG
    - When scaling in, the autoscaler preferentially terminates VMs with the old configuration
    - When scaling out, VMs with the latest configuration are created



Methods to apply a new configuration to existing VMs

Performing a manual selective/opportunistic update using the CLI

- Set a new instance template for the group

```
gcloud compute instance-groups managed set-instance-template MIG_NAME \
    --template=NEW_TEMPLATE
```

- Manually applying the new template to specific VMs

```
gcloud compute instance-groups managed update-instances MIG_NAME \
    --instances INSTANCE_NAMES \
    --most-disruptive-allowed-action replace \
    --minimal-action replace
```

DISRUPTION_LEVEL is discussed on the next slide

- Manually applying the new template to all VMs

```
gcloud compute instance-groups managed update-instances MIG_NAME \
    --all-instances \
    --most-disruptive-allowed-action replace \
    --minimal-action replace
```

Selectively apply VM configuration updates in a MIG

Controlling the disruption level during selective updates

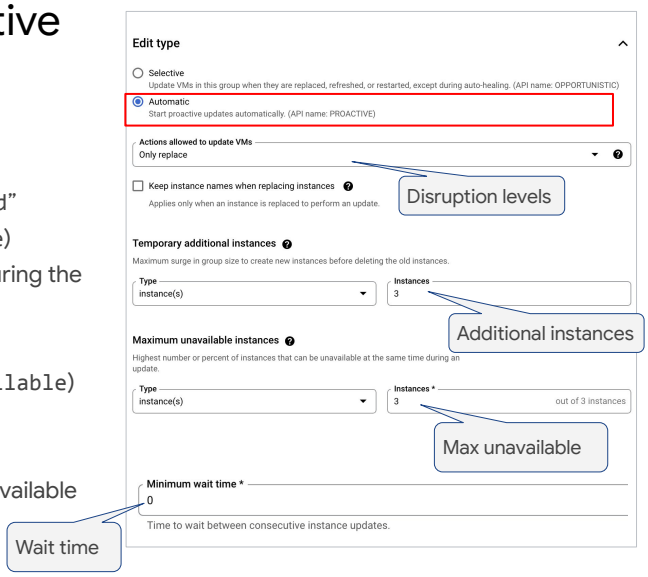
- The default minimal action is NONE, which limits disruption as much as possible
- The default ***`most-disruptive-allowed-action`*** is REPLACE

Value	Description	Which instance properties can be updated?
NONE	Do not disrupt the instance at all.	None
REFRESH	Do not stop the instance.	Additional disks, instance metadata, labels, tags
RESTART	Stop the instance and start it again.	Additional disks, instance metadata, labels, tags, machine type
REPLACE	Delete the instance and create it again.	All instance properties stored in the instance template

Control the disruption level during selective updates

Performing automatic/proactive update with the Console

- Select the new template
- Choose "`Automatic`" for the "`Edit type`"
- Choose "`Only replace`" for the "`Actions allowed`"
- `Temporary additional instances` (`--max-surge`)
  - # or percent of additional instances to create during the update
    - Helps maintain availability
- `Maximum unavailable instances` (`--max-unavailable`)
  - # or percent that can be unavailable at one time
- `Minimum wait time` (beta) (`--min-ready`)
  - Time to wait before marking a new instance as available



[Methods to apply a new configuration to existing VMs]

Performing automatic/proactive update using the CLI

- Perform a rolling update of all VM instances and create up to 5 new instances above the target size at the time of the update using the new template
  - Only 2 VMs can be taken down at a time during the update

```
gcloud compute instance-groups managed rolling-action start-update MIG_NAME \
    --version=template=NEW_TEMPLATE \
    --max-surge=5 \
    --max-unavailable=2 \
    [--zone=ZONE | --region=REGION]
```

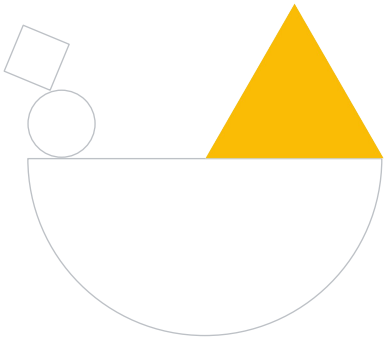
- Perform a rolling update with at most 3 unavailable machines and a minimum wait time of 2 minutes before marking a new instance as available

```
gcloud beta compute instance-groups managed rolling-action start-update MIG_NAME \
    --version=template=NEW_TEMPLATE \
    --min-ready=120 \
    --max-unavailable=3 \
    [--zone=ZONE | --region=REGION]
```

gcloud compute instance-groups managed rolling-action start-update

Managed Instance Group deployment patterns

- Rolling Update
- **Recreate strategy**
- Load balancing review
- Blue/Green
- Canary testing
- A/B testing
- Shadow testing



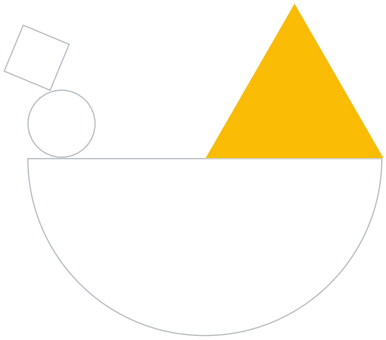
Recreate strategy with managed instance groups

- Implemented as a special case of the “rolling update” strategy
  - Create the new instance template
  - Perform a rolling update with specific parameters
    - **`--max-unavailable`**: Set to the total number of instances in the group
    - **`--action`**:
      - Set to **`RESTART`** to restarts instances with the same template
      - Set to **`REPLACE`** to use a new template

```
gcloud compute instance-groups managed rolling-action start-update MIG_NAME \
    --version template=NEW_INSTANCE_TEMPLATE \
    --max-unavailable=100% \
    --action=REPLACE
```

Managed Instance Group deployment patterns

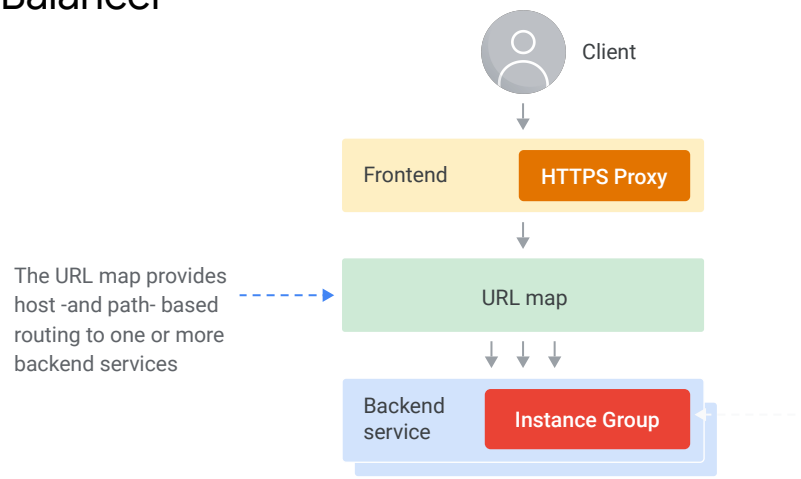
- Rolling updates
- Recreate strategy
- **Load balancing review**
- Blue/Green
- Canary testing
- A/B testing
- Shadow testing



Advanced capabilities of Cloud Load Balancing supports canary, blue/green, A/B and shadow testing

- Cloud Load Balancing provides multiple options for routing traffic
  - Traffic splitting
    - Uses “weights” (percentages) to send a percentage of traffic to one backend service
      - The remainder goes to another
    - The percentages can be changed over time to direct traffic to a certain service until it receives 100% of the traffic
  - Header-based routing, query based routing
    - Route traffic based on the data present in the HTTP request
      - Cookies, queries, etc
  - Traffic mirroring (shadowing)
    - Sends an identical request to the configured mirror backend service on a fire and forget basis

Review: The three parts of the Global HTTPS Load Balancer

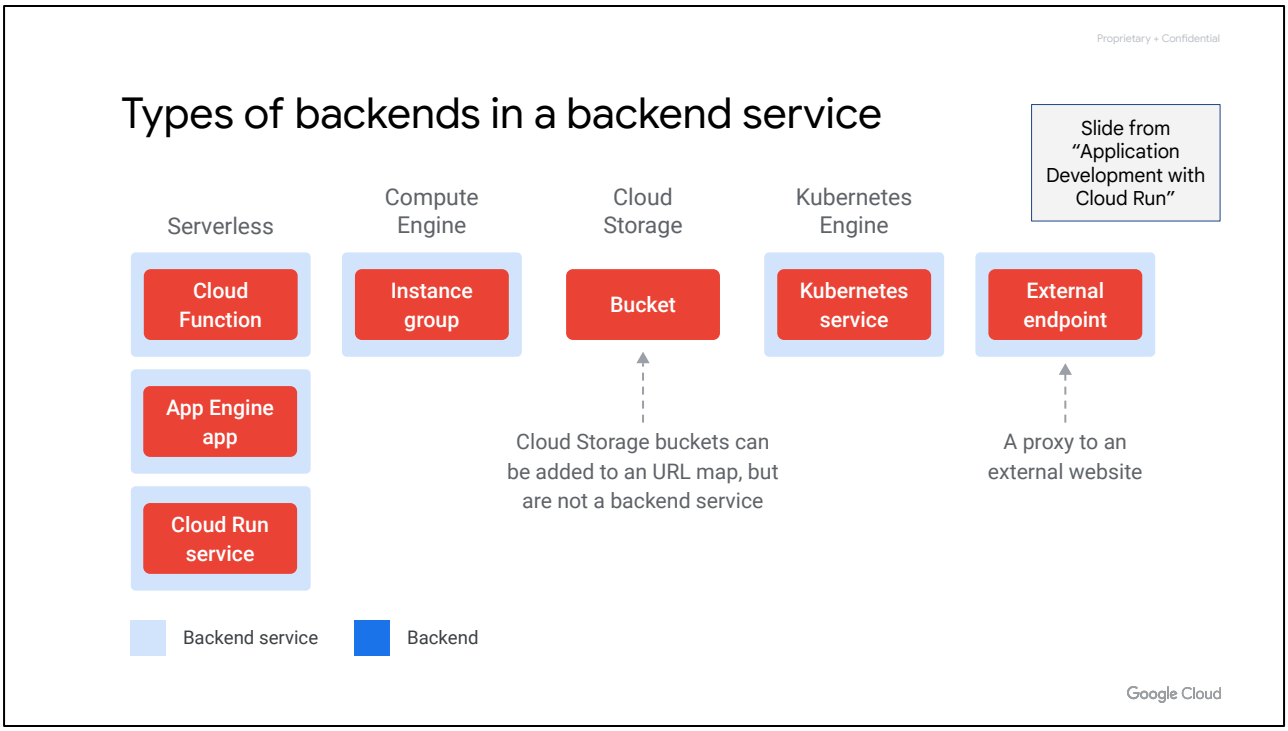


Frontend: You'll get a single, static IP address and you can send HTTPS traffic to it. The frontend forwards the traffic to the URL Map

URL map: The next part is a URL map, which has path-based routing configuration to send traffic to one (or more) backend services.

Backend services: Finally, the backend services contain the backends that handle the requests. Managed Instance Groups, Cloud Run and other services can be a backend.

With the global load balancer, you can deploy your application in multiple regions and send client requests to the region closest to them. This is called cross-region load balancing.



Let's explore the three parts of the GLB from bottom to top, starting with the backend services.

A *backend service* holds backends of the same type

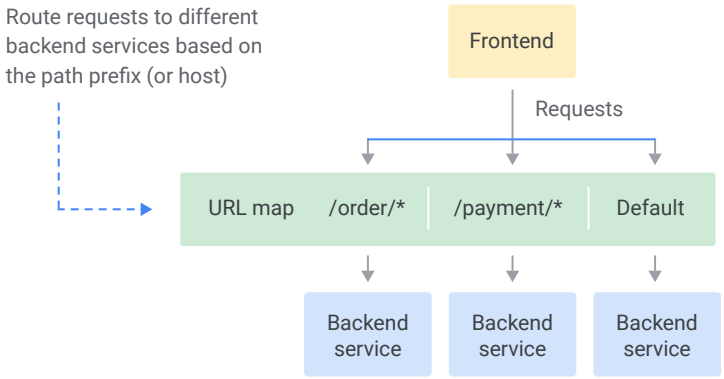
Examples of *backends* are:

- A Compute Engine instance group with virtual machines
- A Kubernetes service in Google Kubernetes Engine
- An external endpoint (the external HTTPS load balancer will proxy the requests). This last backend type can be a way to incrementally rebuild your application when you are still hosting parts on premise.

Cloud Storage buckets can be added to an URL map, but are not a backend service.

The URL map routes requests to backend services

Slide from "Application Development with Cloud Run"



One level closer to the client from the backend service is the URL map.

A *URL map* can match paths or hosts in the request to send traffic to different backend services.

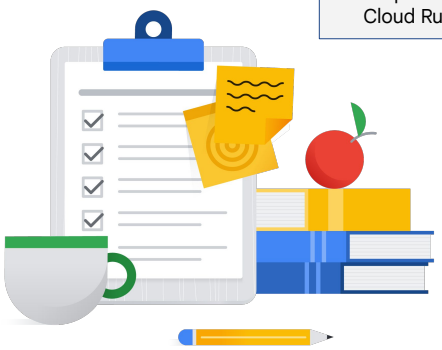
This is how you can combine different Cloud Run services with a bucket to host static assets, and even forward certain request to a service running on-premises in your datacenter.

The URL map is also useful to isolate parts of your application from each other. If you're running an ecommerce website, you don't want your entire application to stop working if the shopping cart breaks.

Summary

1. The three parts of the Global Load Balancer are the **frontend**, the **URL map** and the **backend services**
2. A **Managed Instance Group** can be a backend service, as well as other types of services in Google Cloud, e.g., a **Cloud Run service**
3. The **URL map** uses **host and path-based routing** to forward requests to backend services

Slide from "Application Development with Cloud Run"



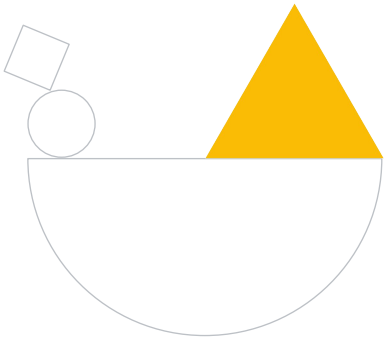
The three parts of the Global HTTPS Load Balancer are the frontend, the URL map and the backend services

A Cloud Run service can be a backend service, as well as other types of services in Google Cloud

The URL map uses host and path-based routing to forward requests to backend services

Managed Instance Group deployment patterns

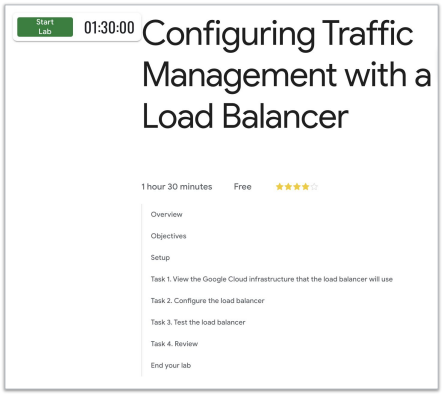
- Rolling updates
- Recreate strategy
- Load balancing review
- **Blue/Green**
- Canary testing
- A/B testing
- Shadow testing



Demo: Using traffic splitting with Blue/Green deployments

- This lab illustrates the Blue/Green deployments using traffic splitting
  - 2 separate instance groups are created
  - You configure a regional internal HTTP(S) load balancer to balance traffic between two backends
    - Blue (one instance group) and Green (the other)
- Traffic splitting based on weights is setup to send a certain % of traffic to the Green backend and the rest to the Blue backend

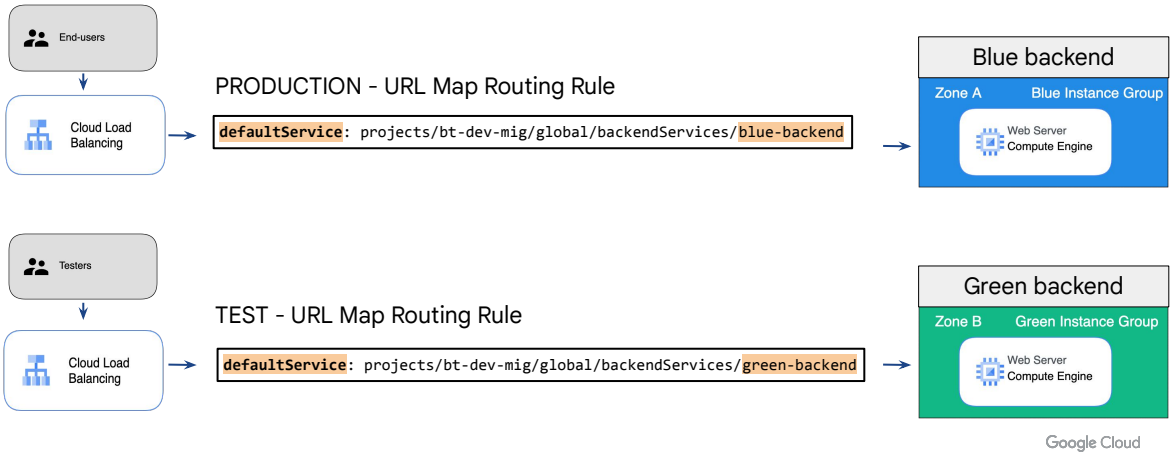
- Additional documentation on this process
  - https://cloud.google.com/load-balancing/docs/l7-internal/setting-up-traffic-management



https://partner.cloudskillsboost.google/catalog_lab/5795

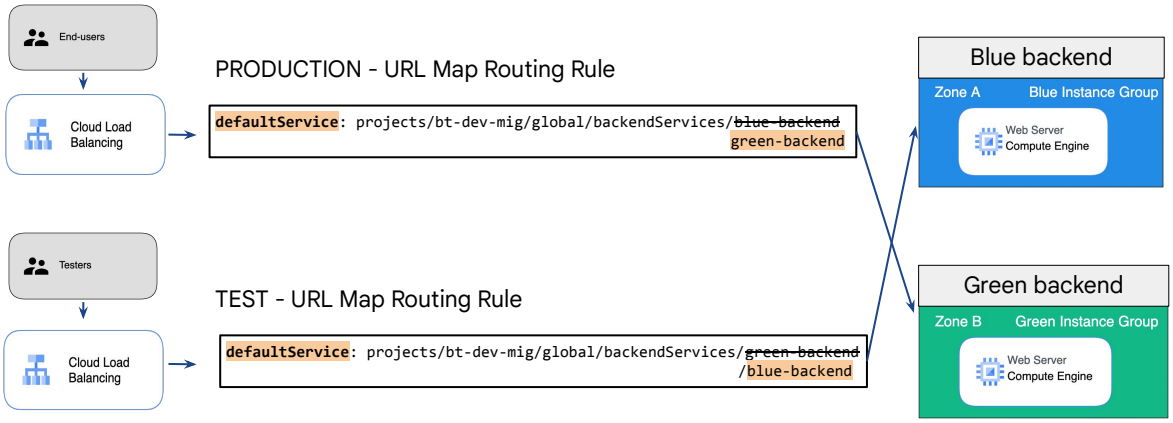
Blue/Green deployment with HTTPS Load balancing - Simplistic method

- Blue instance is production; green instance is test
  - Create a separate load balancer for green during testing

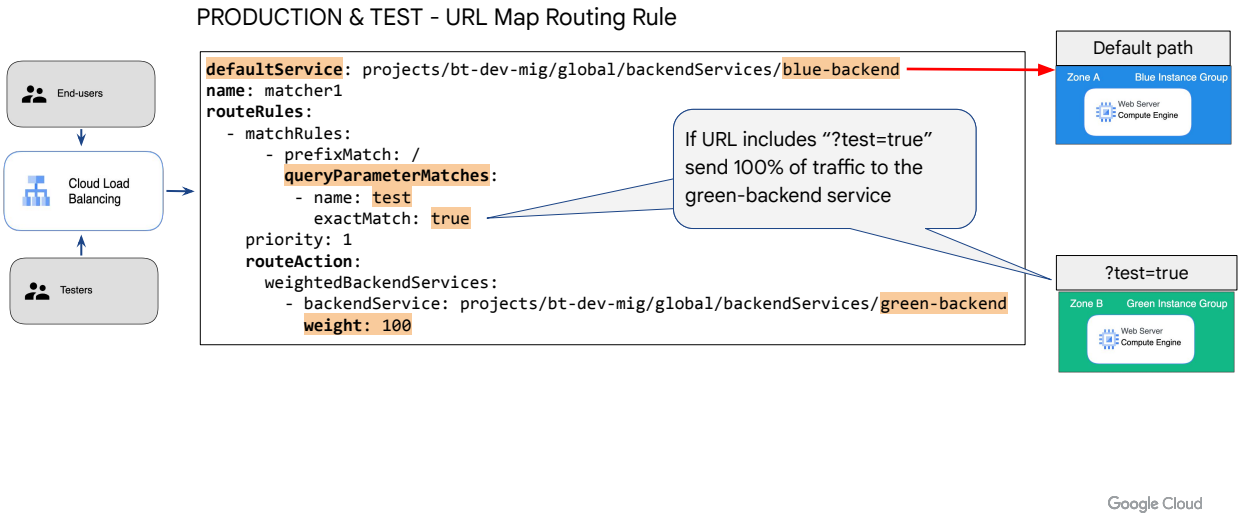


Blue/Green testing complete

- Update the backend services and the routing rules when testing is over
  - Easily roll-back if needed

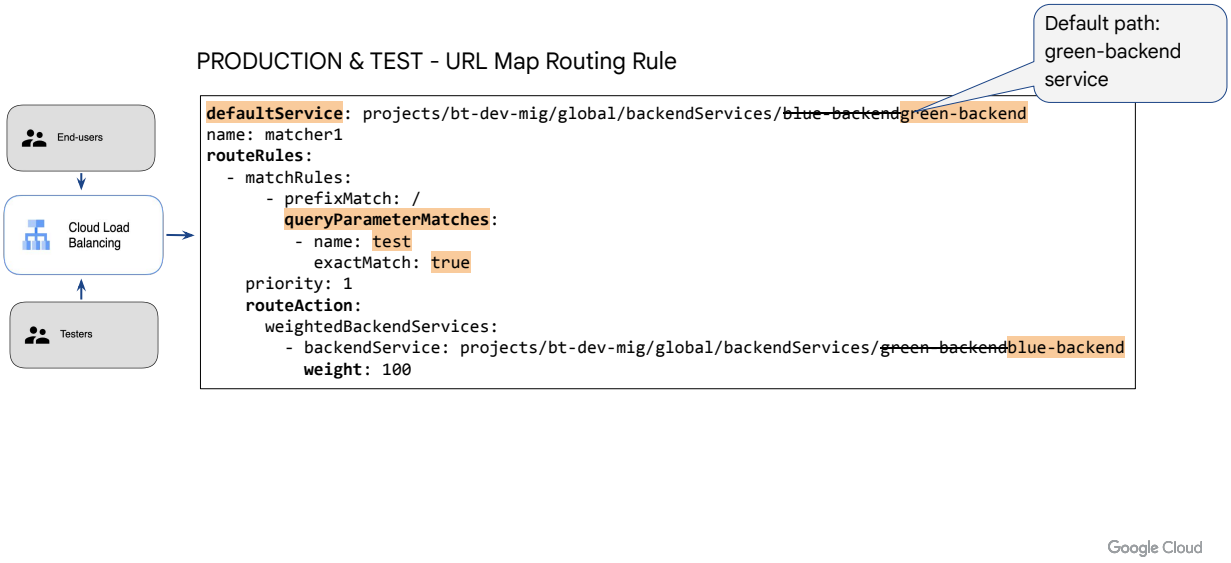


Alternative method: Blue/Green deployment with traffic splitting based on a query parameter



Tutorial: Request routing to a multi-region external HTTPS load balancer
https://cloud.google.com/load-balancing/docs/https/setting-up-https#gcloud_3

When testing is complete



Tutorial: Request routing to a multi-region external HTTPS load balancer
https://cloud.google.com/load-balancing/docs/https/setting-up-https#gcloud_3

Editing the routing rules with gcloud

- List the url-maps to get the name

```
gcloud compute url-maps list
```

- Edit the url-map and modify the code
  - The changes take effect immediately after saving the edits

```
gcloud compute url-maps edit [URL_MAP_NAME]
```

Opens routing rule in **vim** by default if using Cloud Shell

Use **export EDITOR=nano** to make a one time change

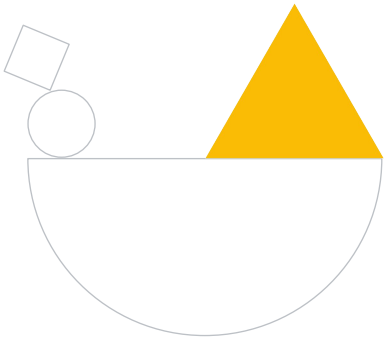
- Verify the changes
  - Will take several minutes to propagate

```
gcloud compute url-maps describe [URL_MAP_NAME]
```


The default editor is vim. To change it to nano, enter export EDITOR=nano

Managed Instance Group deployment patterns

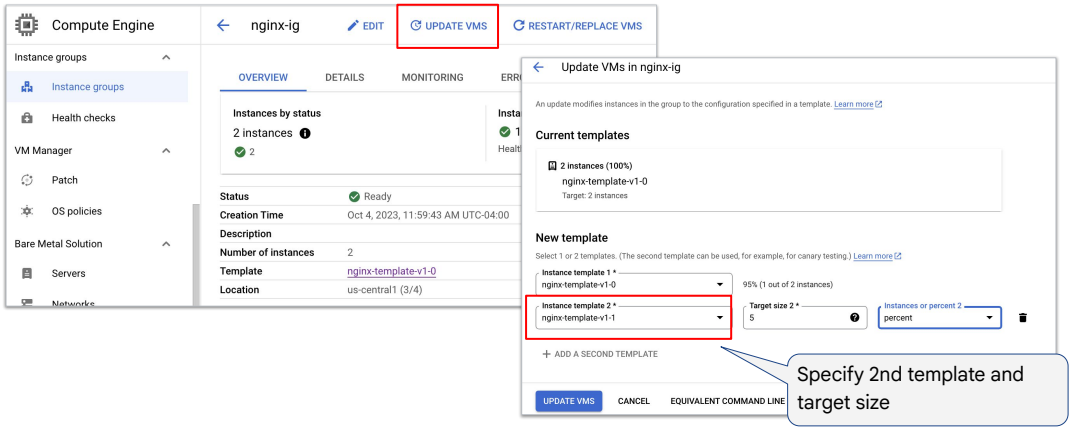
- Rolling updates
- Recreate strategy
- Load balancing review
- Blue/Green
- **Canary testing**
- A/B testing
- Shadow testing



Canary deployments with managed instance groups

- Two methods to implement canary deployments
  - Method 1 - create canaries ***within*** an existing MIG
  - Method 2 - Create a new MIG running the revised application and use the traffic splitting capability of the Load Balancer to split traffic between the original and the revised version (the canary)
    - Gradually increasing the traffic to the canary if all goes well
- Both methods are discussed in the following slides

Method 1: Create canaries within an existing MIG - Console



Method 1: Create canaries within an existing MIG - `gcloud`

- Create a new instance template for the canary
- Use the following rolling-update command to update one or more of the existing instances to the canary template

```
gcloud compute instance-groups managed rolling-action start-update MIG_NAME \
    --version=template=CURRENT_INSTANCE_TEMPLATE_NAME \
    --canary-version=template=NEW_TEMPLATE,target-size=SIZE \
    [--zone=ZONE | --region=REGION]
```

Size: the number or % of instances to apply this update to

- Monitor the canary for errors, etc
- To commit canary update, roll forward by issuing another **`rolling-action start-update`** command setting only the **`version`** flag and omitting the **`--canary-version`** flag.

```
gcloud compute instance-groups managed rolling-action start-update MIG_NAME \
    --version=template=NEW_TEMPLATE \
    [--zone=ZONE | --region=REGION]
```

Starting a canary update

Rolling back canaries within an existing MIG

- There is no explicit command for rolling back an update to a previous version
  - Make a new update request instead
    - Pass in the instance template to which to roll back

```
gcloud compute instance-groups managed rolling-action start-update MIG_NAME \
    --version=template=OLD_INSTANCE_TEMPLATE_NAME \
    --max-unavailable=100% \
    [--zone=ZONE | --region=REGION]
```

100% of your instances will be unavailable for a period of time

Starting a canary update

Method 2: Using Cloud Load Balancing traffic splitting for canary testing

- Example URL map routing rules

```
defaultService: projects/bt-dev-mig/global/backendServices/mig-1-backend
name: matcher1
routeRules:
  - matchRules:
      - prefixMatch: /
    priority: 1
    routeAction:
      weightedBackendServices:
        - backendService: projects/bt-dev-mig/global/backendServices/mig-1-backend
          weight: 90
        - backendService: projects/bt-dev-mig/global/backendServices/mig-2-backend
          weight: 10
```

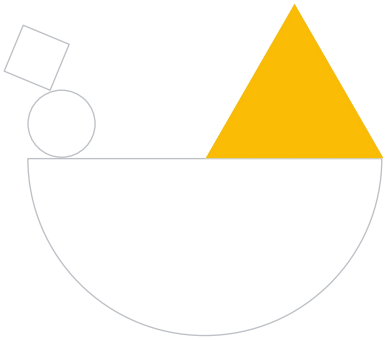
Production

Canary

- Monitor the canary for performance, error rates, etc.
- If all goes well, gradually increase the weights until 100% of traffic is served by the new version

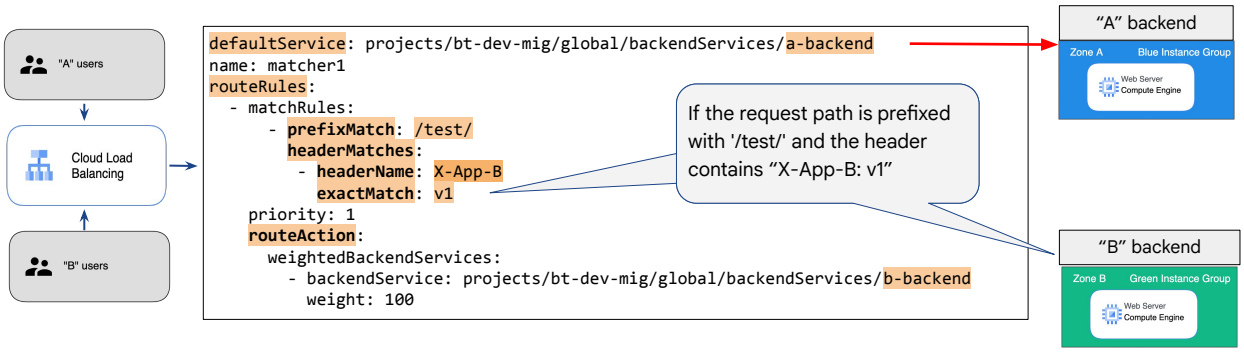
Managed Instance Group deployment patterns

- Rolling updates
- Recreate strategy
- Load balancing review
- Blue/Green
- Canary testing
- **A/B testing**
- Shadow testing



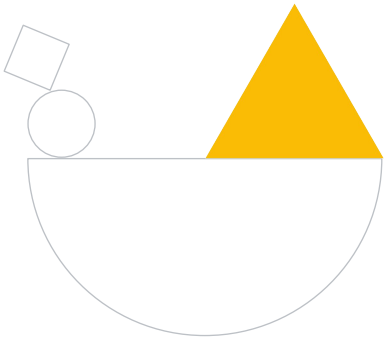
A/B testing with managed instance groups

- Similar to Blue/Green routing
  - Routing options include header based, path based, cookie based or query based routing



Managed Instance Group deployment patterns

- Rolling updates
- Recreate strategy
- Load balancing review
- Blue/Green
- Canary testing
- A/B testing
- **Shadow testing**

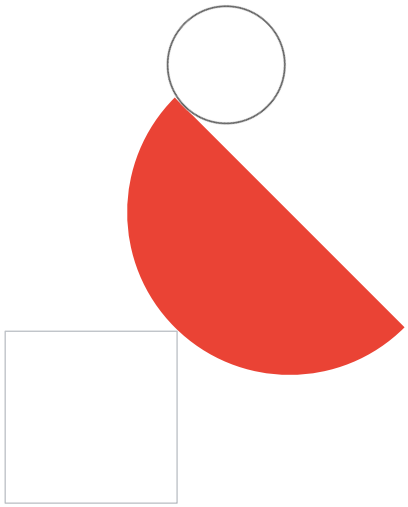


Shadow testing/mirroring with Cloud Load Balancing

- Traffic mirroring “mirrors” production traffic to a dedicated backend service
  - Useful for testing a new version of a backend service or debugging an existing one
- The mirror receives all the requests sent to production
  - Can't be configured to receive only a specific portion of requests
- The load balancer does not wait for response from the mirror

```
pathMatchers:
  - defaultService: global/backendServices/mig-1-backend
    name: matcher1
    routeRules:
      - matchRules:
          - prefixMatch: /
        priority: 1
        routeAction:
          weightedBackendServices:
            - backendService: global/backendServices/mig-1-backend
              weight: 100
          requestMirrorPolicy:
            backendService: global/backendServices/mig-2-backend
```

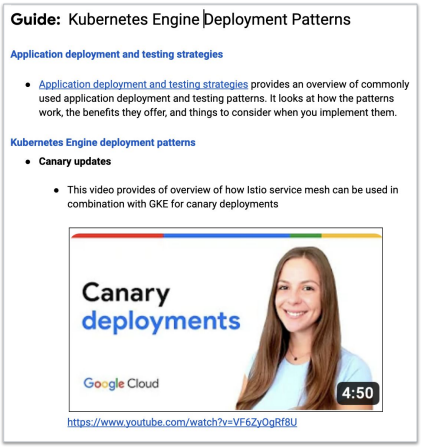
Google Kubernetes Engine deployment patterns



Guide: Kubernetes Engine Deployment Patterns

Topics covered include

- Overview of deployment patterns
- Rolling Updates
- Canary testing
- Service mesh



Kubernetes Engine Deployment Patterns

Review: Kubernetes deployments ensure that sets of Pods are running

- Create a configuration file (manifest) that describes the deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-deployment
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: my-app
        image: gcr.io/demo/my-app:1.0
        ports:
        - containerPort: 8080
```

Deploy 3 pods with this image

kubectl apply -f [DEPLOYMENT_FILE]

Create the deployment

Making changes to an existing deployment

- Method 1: Edit the configuration file, and update values. Then run

```
kubectl apply -f [DEPLOYMENT_FILE]
```

This method is used in this module

- Method 2: If changing a pod's image, run

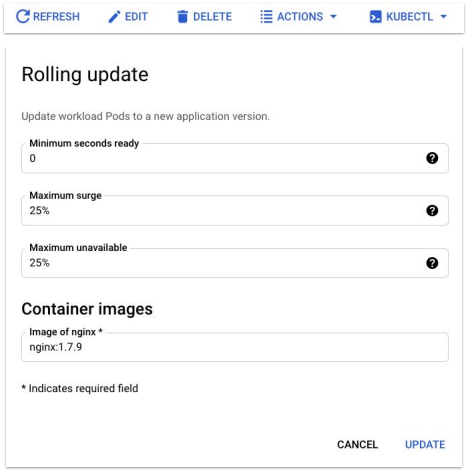
```
kubectl set image deployment [DEPLOYMENT_NAME] [IMAGE] [IMAGE]:[TAG]
kubectl set image deployments/my-app-deployment my-app=gcr.io/demo/my-app:3.0
```

- Method 3: If want to make a quick change and directly edit a resource (using vim, nano, etc.)

```
kubectl edit deployment [DEPLOYMENT_NAME]
kubectl edit deployment my-app-deployment
```

Making changes to an existing deployment

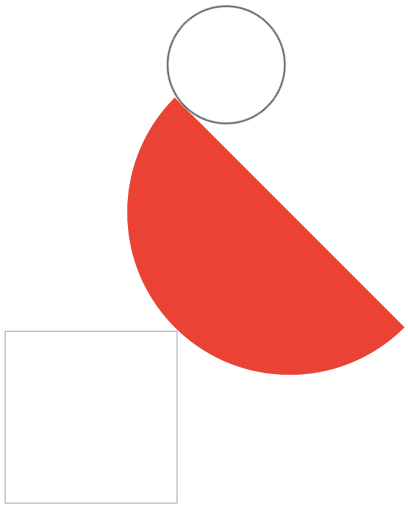
- Method 4: Use the Cloud Console



The last option for you to update a Deployment is through the Cloud Console. You can edit the Deployment manifest from the Cloud Console and perform a rolling update along with additional options.

Google Kubernetes Engine deployment patterns

- **Recreate strategy**
- Rolling updates
- Blue/Green
- Canary testing
- A/B testing
- Shadow testing



Applying a Recreate strategy

- Edit the existing manifest

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  strategy:
      type: Recreate
  template:
    spec:
      containers:
      - name: my-app
        image: us-central1-docker.pkg/my-app:2.0
        ports:
        - containerPort: 8080
```

Existing pods will be terminated and replaced with the new version

kubectl apply -f [DEPLOYMENT_FILE]

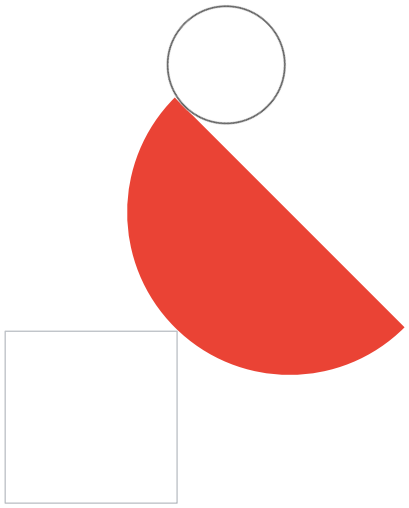
Update the deployment

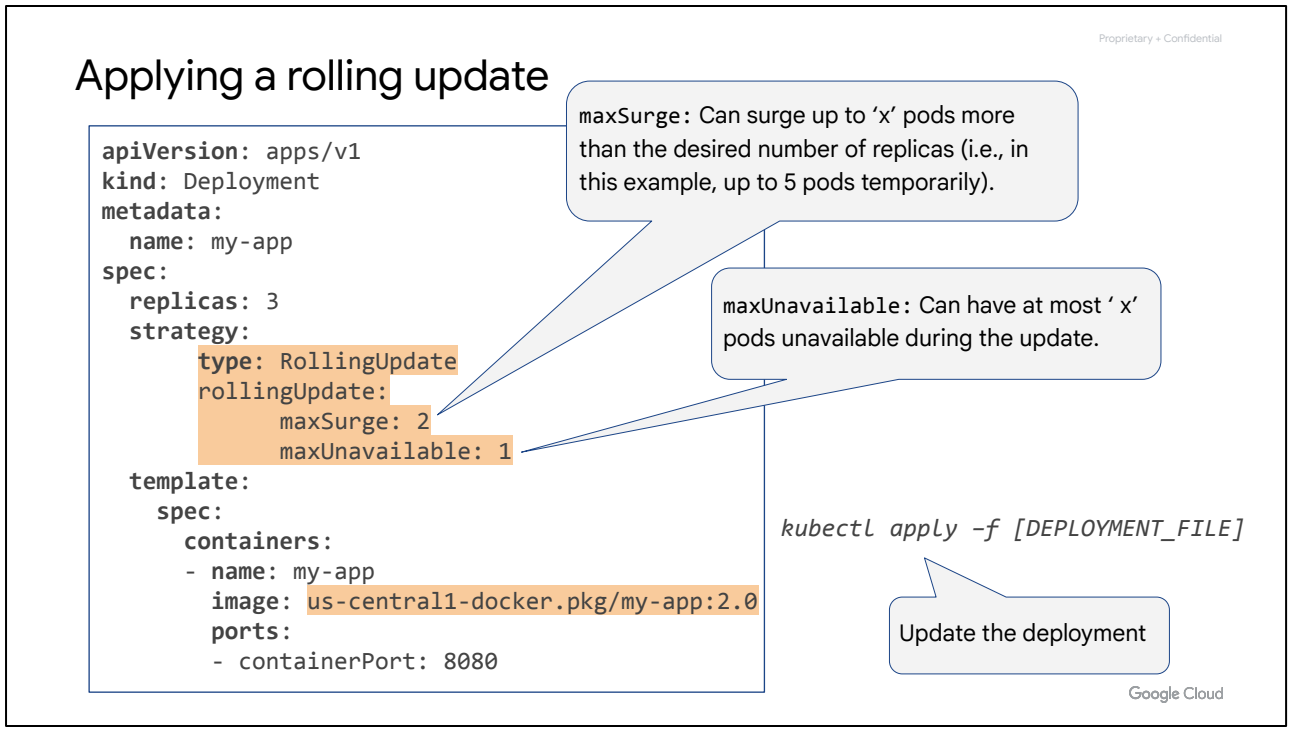
'Recreate' is a strategy type where all the old Pods are deleted before new Pods are created. This clearly affects the availability of your application, because the new Pods must be created and will not all be available instantly. For example, what if the contract of communication between parts of your application is changing, and you need to make a clean break? In such situations a continuous deployment strategy doesn't make sense. All the replicas need to change at once. That's when the Recreate strategy is recommended.

With this strategy, existing pods from the deployment are terminated and replaced with a new version. The application experiences downtime from the time the old version goes down until the new pods start successfully and start serving user requests.

Google Kubernetes Engine deployment patterns

- Recreate strategy
- **Rolling updates**
- Blue/Green
- Canary testing
- A/B testing
- Shadow testing





With Kubernetes, a rolling update can be performed to update images, configuration, labels, annotations, and resource limits/requests. Rolling updates replace the resource pods in a way designed for zero downtime. Rolling updates can be triggered by updating pod templates for:

- DaemonSets
- Deployments
- StatefulSets

For more details, see https://cloud.google.com/kubernetes-engine/docs/how-to/updating-apps.

When you make a change to a Deployment's Pod specification, such as changing the image version, an automatic update rollout happens. Again, note that these automatic updates are only applicable to the changes in Pod specifications.

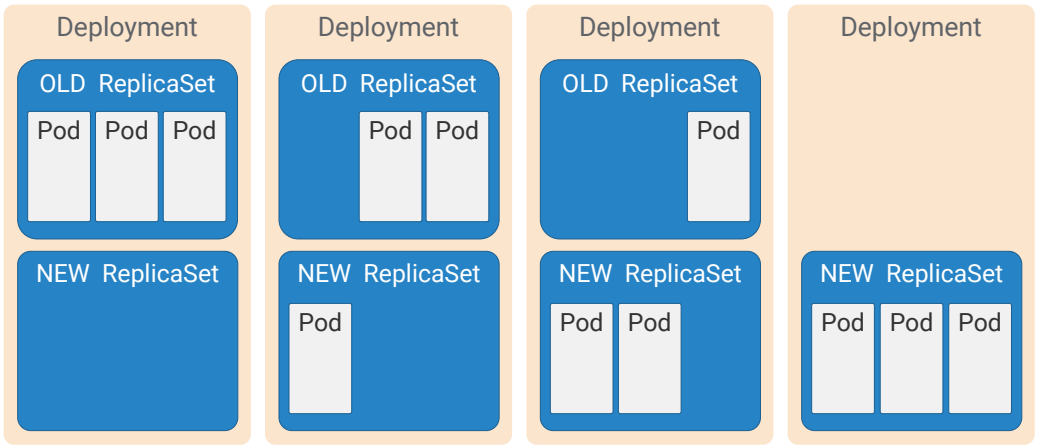
You can update a Deployment in different ways. One way is to use the kubectl 'apply' command with an updated Deployment specification YAML file. This method allows you to update other specifications of a Deployment, such as the number of replicas, outside the Pod template.

You can also use a kubectl 'set' command. This allows you to change the Pod template specifications for the Deployment, such as the image, resources, and

selector values.

Another way is to use a kubectl ‘edit’ command. This opens the specification file using the vim editor that allows you to make changes directly. Once you exit and save the file, kubectl automatically applies the updated file.

The process behind updating a Deployment



When a Deployment is updated, it launches a new ReplicaSet and creates a new set of Pods in a controlled fashion.

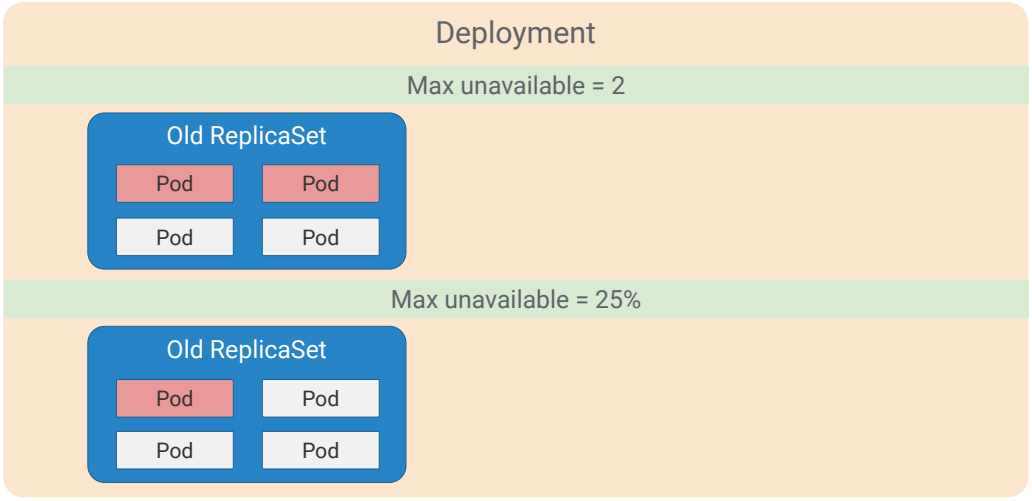
First, new Pods are launched in a new ReplicaSet.

Next, old Pods are deleted from the old ReplicaSet.

This is an example of a rolling update strategy also known as a ramped strategy.

Its advantage is that updates are slowly released, which ensures the availability of the application. However, this process can take time, and there's no control over how the traffic is directed to the old and new Pods.

Set parameters for your rolling update strategy

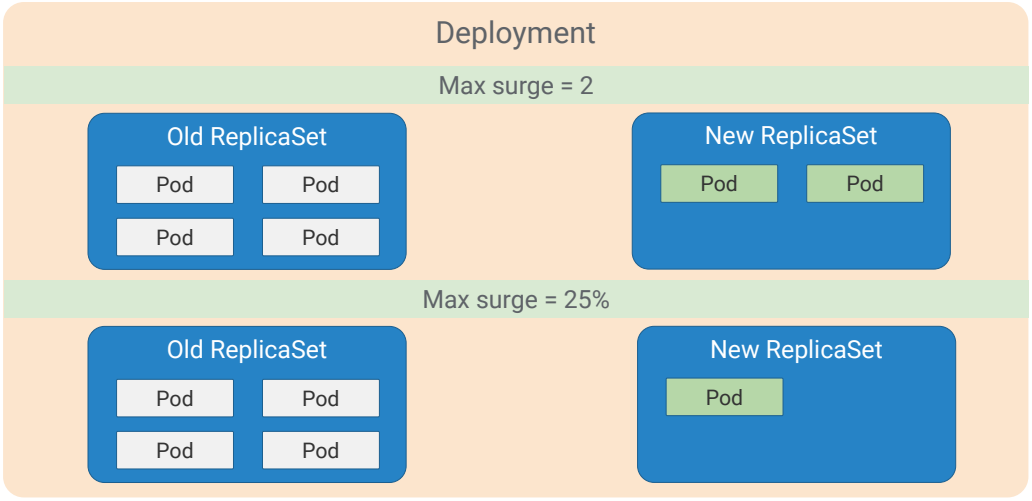


In a rolling update strategy, the max unavailable and max surge fields can be used to control how the Pods are updated. These fields define a range for the total number of running Pods within the Deployment, regardless of whether they are in the old or new ReplicaSets.

The max unavailable field lets you specify the maximum number of Pods that can be unavailable during the rollout process. This number can be either absolute or a percentage. For example, you can say you want to have no more than 2 Pods unavailable during the upgrade process.

Specifying max unavailable at 25% means you want to have at least 75% of the total desired number of Pods running at the same time. The default max unavailable is 25%.

Set parameters for your rolling update strategy



Max surge allows you to specify the maximum number of extra Pods that can be created concurrently in a new ReplicaSet. For example, you can specify that you want to add up to two new Pods at a time in a new ReplicaSet, and the Deployment controller will do exactly that.

You can also set max surge as a percentage. The Deployment controller looks at the total number of running Pods in both ReplicaSets, Old and New.

In this example, a Deployment with the desired number of Pods as 4 and a max surge of 25% will allow a maximum total of 5 Pods running at any given time between the old and new ReplicaSets. In other words, it'll allow 125% of the desired number of Pods, which is 5. Again, the default max surge is 25%.

Rolling back a Deployment

```
# Revert to the previous version
kubectl rollout undo deployment [DEPLOYMENT_NAME]

# Revert to a specific version
kubectl rollout undo deployment [DEPLOYMENT_NAME] --to-revision=2

# View rollout history
kubectl rollout history deployment [DEPLOYMENT_NAME]
```


You roll back using a kubectl ‘rollout undo’ command. A simple ‘rollout undo’ command will revert the Deployment to its previous revision.

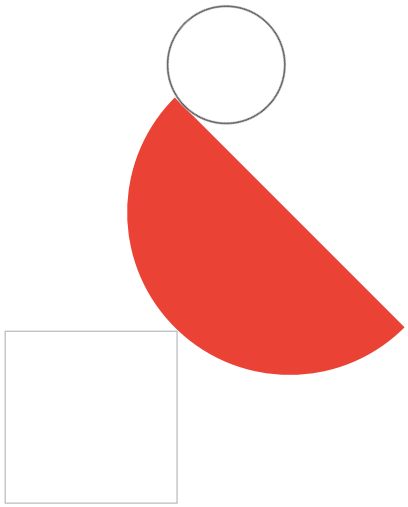
You roll back to a specific version by specifying the revision number.

If you’re not sure of the changes, you can inspect the rollout history using the kubectl ‘rollout history’ command.
The Cloud Console doesn’t have a direct rollback feature; however, you can start Cloud Shell from your Console and use these commands. The Cloud Console also shows you the revision list with summaries and creation dates.

By default, the details of 10 previous ReplicaSets are retained, so that you can roll back to them. You can change this default by specifying a revision history limit under the Deployment specification.

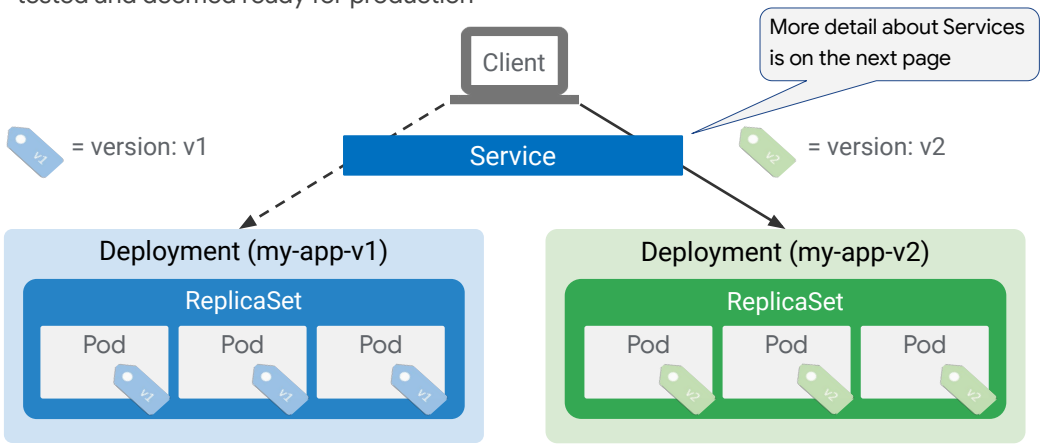
Google Kubernetes Engine deployment patterns

- Recreate strategy
- Rolling updates
- **Blue/Green**
- Canary testing
- A/B testing
- Shadow testing



Blue/green deployment strategy in Kubernetes Engine

- A service* routes traffic to the “blue” environment until the “green” environment has been tested and deemed ready for production



*A service in Kubernetes is different from the Cloud Load Balancer discussed in Managed Instance Groups

A blue/green deployment strategy is useful when you want to deploy a new version of an application and also ensure that application services remain available while the Deployment is updated.

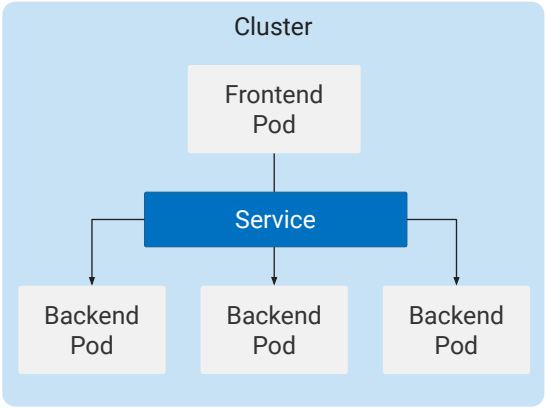
With a blue/green update strategy, a completely new Deployment is created with a newer version of the application. In this case, it’s my-app-v2.

When the Pods in the new Deployment are ready, the traffic can be switched from the old blue version to the new green version. But how can you do this?

This is where a Kubernetes Service is used. Services allow you to manage the network traffic flows to a selection of Pods. This set of Pods is selected using a label selector.

A service delivers traffic to a set of pods

- Pods are transient - are added/deleted as needed
  - There is no fixed IP address for an app to use to communicate with a pod
- A service* provides a static IP address and can serve traffic to the pods



*There are many types of services. Detailed discussion of these is beyond the scope of this module.

We haven't yet discussed how to locate and connect to the applications running in these Pods, especially as new Pods are created or updated by your Deployments. While you can connect to individual Pods directly, Pods themselves are transient. A Kubernetes Service is a static IP address that represents a Service, or a function, in your infrastructure. It's a stable network abstraction for a set of Pods that deliver that Service and, and it hides the ephemeral nature of the individual Pods.

Applying a Blue/Green deployment

Blue deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-blue
  labels:
    name: nginx
    version: "blue"
spec:
  replicas: 3
  selector:
    matchLabels:
      version: "blue"
  template:
    metadata:
      labels:
        version: "blue"
    spec:
      containers:
      - name: nginx
        image: nginx:1.17.1
        ports:
        - containerPort: 80
```

Green deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-green
  labels:
    name: nginx
    version: "green"
spec:
  replicas: 3
  selector:
    matchLabels:
      version: "green"
  template:
    metadata:
      labels:
        version: "green"
    spec:
      containers:
      - name: nginx
        image: nginx:1.17.2
        ports:
        - containerPort: 80
```

Services exist for both deployments during testing

- One going to the blue (production) deployment
- One going to the green (test) deployment
  - Testing occurs in the green environment

```
apiVersion: v1
kind: Service
metadata:
  name: prod-service
  labels:
    name: nginx
spec:
  ports:
    - name: http
      port: 80
      targetPort: 80
  selector:
    version: "blue"
  type: LoadBalancer
```

Service routes traffic to any pod with a key value pair of "version: blue"

prod-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: test-service
  labels:
    name: test-service
spec:
  ports:
    - name: http
      port: 80
      targetPort: 80
  selector:
    version: "green"
  type: LoadBalancer
```

Service routes traffic to any pod with a key value pair of "version: green"

test-service.yaml

When testing is successful

- Switch the production service to serve traffic to the green pods instead of the blue and vice versa
  - Update the service so that it serves pods with a “version: green” label only

```
kubectl patch service prod-service -p '{"spec":{"selector":{"version":"green"}}}'
```

```
apiVersion: v1
kind: Service
metadata:
  name: prod-service
  labels:
    name: nginx
spec:
  ports:
    - name: http
      port: 80
      targetPort: 80
  selector:
    version: "bluegreen"
  type: LoadBalancer
```

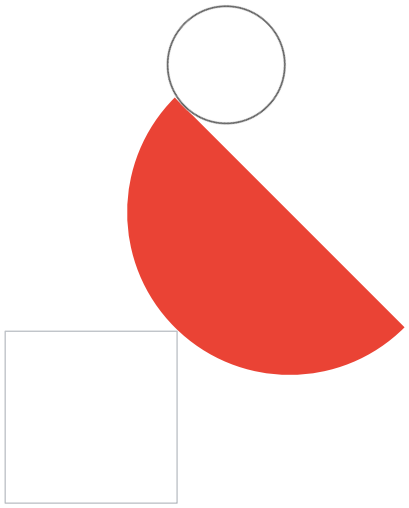
prod-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: test-service
  labels:
    name: test-service
spec:
  ports:
    - name: http
      port: 80
      targetPort: 80
  selector:
    version: "greenblue"
  type: LoadBalancer
```

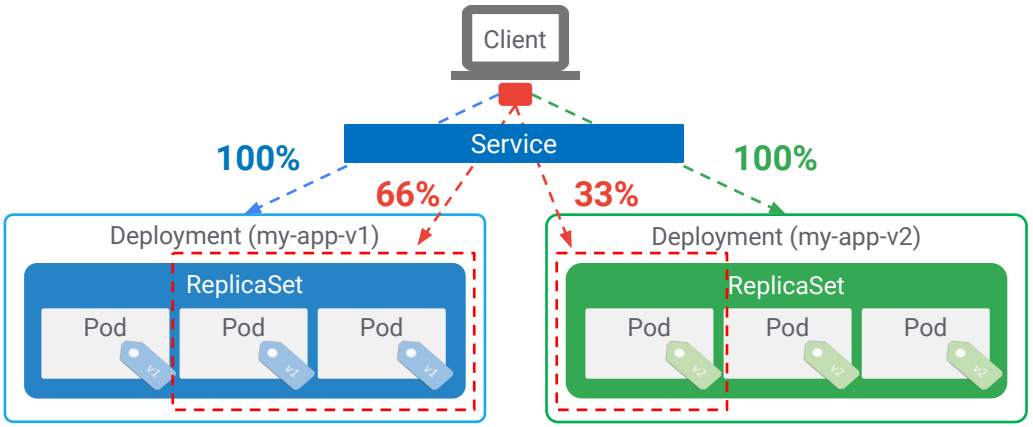
test-service.yaml

Google Kubernetes Engine Deployments

- Recreate strategy
- Rolling updates
- Blue/Green
- **Canary testing**
- A/B testing
- Shadow testing



Canary deployment is an update strategy where traffic is gradually shifted to the new version



The canary method is another update strategy based on the blue/green method, but traffic is *gradually* shifted to the new version. The main advantages of using canary deployments are that you can minimize excess resource usage during the update, and because the rollout is gradual, issues can be identified before they affect all instances of the application.

In this example, 100% of the application traffic is directed initially to my-app-v1 .

When the canary deployment starts, a subset of the traffic, 33% in this case, or a single pod, is redirected to the new version, my-app-v2, while 66%, or two pods, from the older version, my-app-v1, remain running.

When the stability of the new version is confirmed, 100% of the traffic can be routed to this new version. How is this done?

Applying a canary deployment

Existing deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-stable
spec:
  replicas: 9
  selector:
    matchLabels:
      app: my-app
      version: stable
  template:
    metadata:
      labels:
        app: my-app
        version: stable
    spec:
      containers:
      - name: my-app-container
        image: us-central1-docker.pkg/my-app:1.0
```

kubectl apply -f stable.yaml

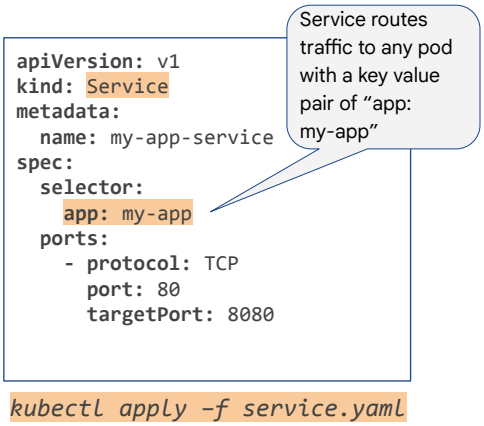
Canary deployment

```
apiVersion:
kind: Deployment
metadata:
  name: my-app-canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
      version: canary
  template:
    metadata:
      labels:
        app: my-app
        version: canary
    spec:
      containers:
      - name: my-app-container
        image: us-central1-docker.pkg/my-app:2.0
```

kubectl apply -f canary.yaml

The same service routes traffic to both sets of pods

- The stable version of the application (my-app:1.0) has 9 replicas
- The canary version (my-app:2.0) has 1 replica
- The service (my-app-service) routes traffic to both the stable and canary versions
  - 90% of traffic goes to the stable version
- For a more advanced way of routing traffic see
  - Anthos Service Mesh by example: Canary Deployments



In the blue/green update strategy covered previously, both the app and version labels were selected by the Service, so traffic would only be sent to the Pods that are running the version defined in the Service.

In a Canary update strategy, the Service selector is based only on the application label and does not specify the version. The selector in this example covers all Pods with the app:my-app label. This means that with this Canary update strategy version of the Service, traffic is sent to all Pods, regardless of the version label.

This setting allows the Service to select and direct the traffic to the Pods from both Deployments. Initially, the new version of the Deployment will start with zero replicas running. Over time, as the new version is scaled up, the old version of the Deployment can be scaled down and eventually deleted.

With the canary update strategy, a subset of users will be directed to the new version. This allows you to monitor for errors and performance issues as these users use the new version, and you can roll back quickly, minimizing the impact on your overall user base, if any issues arise.

However, the complete rollout of a Deployment using the canary strategy can be a slow process and may require tools such as Istio to accurately shift the traffic.

Anthos Service Mesh (ASM) provides advanced traffic management for Kubernetes

- Provides a uniform way to connect, manage, and secure microservices across on-premises, multi-cloud, and hybrid environments
  - **Built on top of Istio, an open-source service mesh**
- Features
  - Traffic Management: Advanced traffic routing, load balancing, and fault injection
  - Security: Mutual TLS (mTLS) for service-to-service communication, identity and access control
  - Observability: Integrated with monitoring, logging, and tracing tools like Cloud Monitoring and Cloud Logging.
  - Policies: Allows setting fine-grained policies for traffic, security, and access control

Anthos Service Mesh offers advanced traffic management features. It lets you easily control the flow of traffic and API calls between services with much greater granularity than standard Kubernetes. It also simplifies configuration of service-level properties like circuit breakers, timeouts, and retries, and makes it easy to set up important tasks like A/B testing, canary rollouts, and staged rollouts with percentage-based traffic splits. Beyond that, it provides out-of-box failure recovery features that help make your application more robust against failures of dependent services or the network.

ASM makes it easy to test Kubernetes deployments

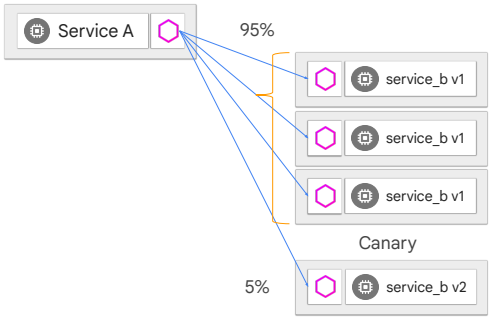
- Easy to set up tasks like:
  - A/B testing
  - Canary rollouts
  - Percentage-based traffic splits
  - Shadow testing/traffic mirroring
- Can configure service-level properties such as:
  - Request timeout - How long to wait for a request to complete before returning an error
  - Retries - Set a max number of retries to a backend
  - Circuit breakers - Prevent further requests to a service for a specified period if the error rate goes above a certain threshold
  - Fault injection - deliberately introduce failures to test a service's resilience

Canary deployment strategy with ASM/Istio

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-b
spec:
  hosts:
    - service-b
  http:
  - route:
    - destination:
        host: service-b
        subset:v1
      weight: 95
    - destination:
        host: service-b
        subset: v2
      weight: 5
```

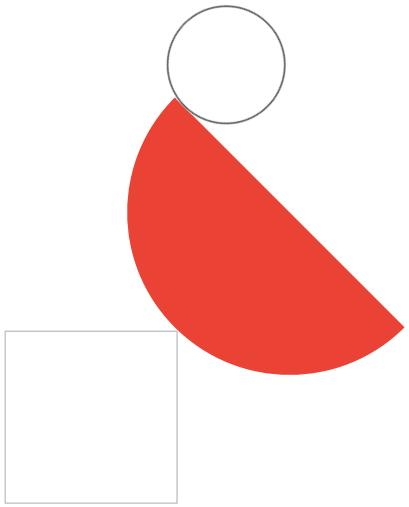
Virtual service provides fine-grained traffic routing capabilities

Weight defines the percent of traffic to send to a service



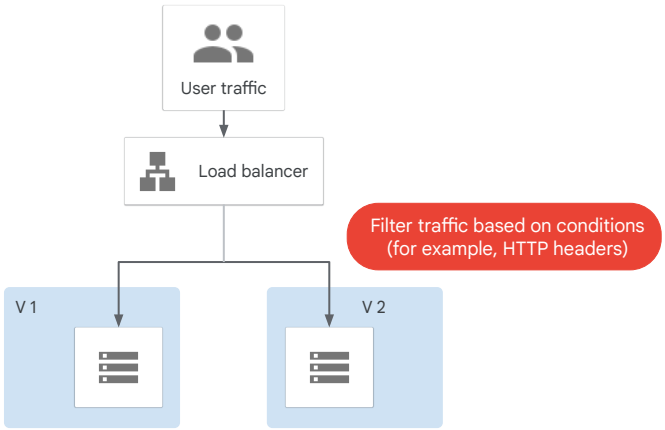
Google Kubernetes Engine deployment patterns

- Recreate strategy
- Rolling updates
- Blue/Green
- Canary testing
- **A/B testing**
- Shadow testing



A/B testing is used to measure the effectiveness of functionality in an application

- ASM can route traffic based on HTTP headers, URI paths, query parameters and more
  - For example,
    - Route traffic to V2 when the HTTP header `x-test-version` is set to `beta`, otherwise route traffic to v1



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.

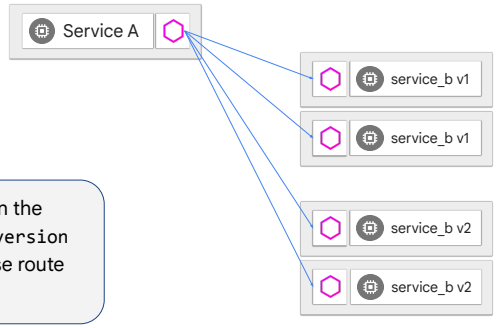
When you perform an A/B test, you route a subset of users to new functionality based on routing rules as shown in the example here. Routing rules often include factors such as browser version, user agent, geolocation, and operating system. After you measure and compare the versions, you update the production environment with the version that yielded better results.

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.

A/B deployment strategy with ASM/Istio

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-b
spec:
  hosts:
    - service-b
  http:
  - match:
    - headers:
        x-test-version:
          exact: beta
    route:
    - destination:
        host: service-b
        subset: v2
  - route:
    - destination:
        host: service-b
        subset: v1
```

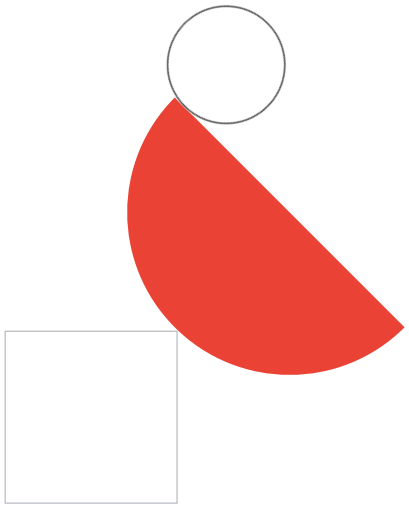
Route traffic to v2 when the HTTP header `x-test-version` is set to `beta`, otherwise route traffic to v1



The match condition checks if the HTTP header x-test-version is set to beta.
If the condition is met, traffic is routed to the new-service subset.
If the condition is not met (i.e., the header is not present or has a different value), the next route section is evaluated, which routes traffic to the old-service subset.

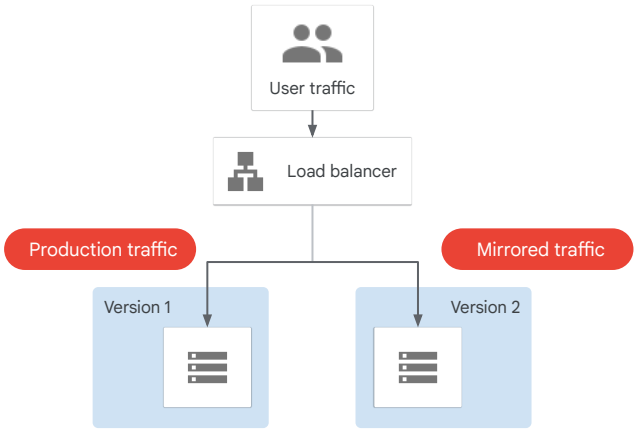
Google Kubernetes Engine deployment patterns

- Recreate strategy
- Rolling updates
- Blue/Green
- Canary testing
- A/B testing
- **Shadow testing**



Shadow testing allows you test new features without exposing users to potential issues

- Copy real-time traffic from production and send it to a staging or test environment
  - Test new features, configurations, or infrastructure changes without impacting users
- ASM/Istio traffic mirroring feature supports shadow testing



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 diagram shown illustrates. 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.

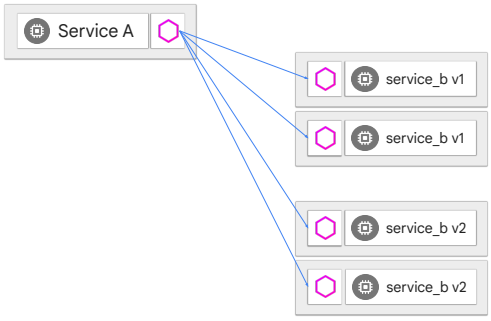
You need to ensure that the shadow tests do not trigger side effects that can alter the existing production environment or the user state.

Shadow testing has many key benefits. Because traffic is duplicated, any bugs in services that are processing shadow data have no impact on production. 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. Finally, there is a 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.

Shadow deployment strategy with ASM/Istio

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: service-b
spec:
  hosts:
    - service-b
  http:
  - route:
    - destination:
        host: service-b
        subset: v1
      weight: 100
    mirror:
      host: service-b
      subset: v2
    mirror_percent: 100
```

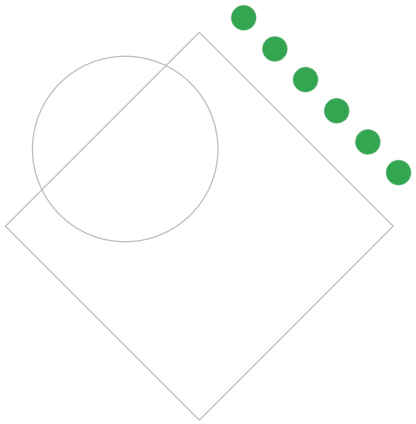
V2 will receive a copy of everything sent to V1



The route section specifies that 100% of the traffic should go to the v1 subset.
The mirror section specifies that the traffic should be mirrored to the v2 subset.
The mirror_percent specifies the percentage of traffic to be mirrored. In this example, we're mirroring 100% of the traffic.
With this configuration, all traffic going to service-b will be routed to old-service, and the same traffic will also be mirrored to new-service without affecting the response to the original requests. This allows you to observe the behavior of the new-service under real-world traffic conditions without impacting users.

Cloud Run deployment patterns

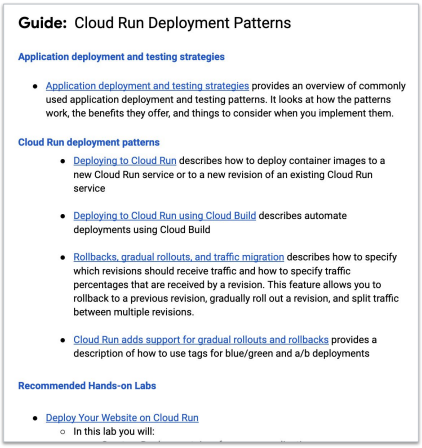
- **Cloud Run review**
- Canary testing
- Blue/Green and A/B testing



Guide: Cloud Run Deployment Patterns

Topics covered include

- Overview of deployment patterns
- Canary testing via traffic splitting
- Rolling back to a prior revision
- Using tags for Blue/Green deployments



Guide: Cloud Run Deployment Patterns

Application deployment and testing strategies

- Application deployment and testing strategies provides an overview of commonly used application deployment and testing patterns. It looks at how the patterns work, the benefits they offer, and things to consider when you implement them.

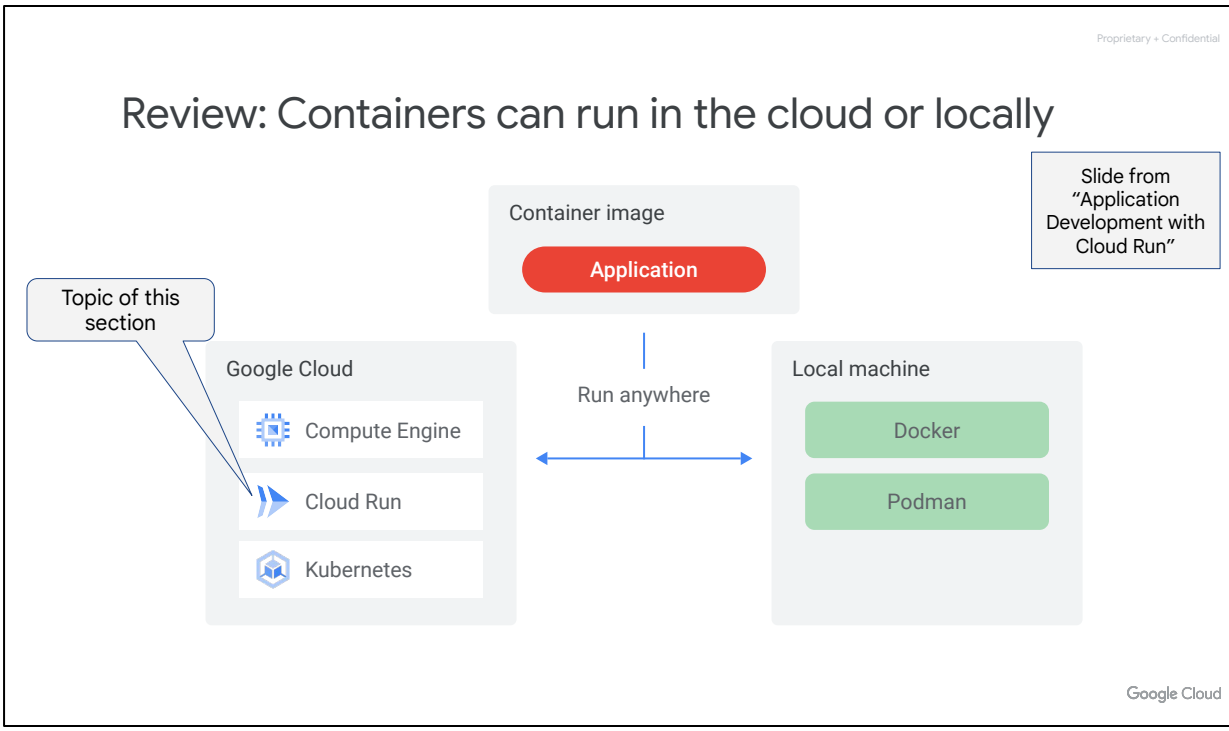
Cloud Run deployment patterns

- Deploying to Cloud Run describes how to deploy container images to a new Cloud Run service or to a new revision of an existing Cloud Run service
- Deploying to Cloud Run using Cloud Build describes automate deployments using Cloud Build
- Rollbacks, gradual rollouts, and traffic migration describes how to specify which revisions should receive traffic and how to specify traffic percentages that are received by a revision. This feature allows you to rollback to a previous revision, gradually roll out a revision, and split traffic between multiple revisions.
- Cloud Run adds support for gradual rollouts and rollbacks provides a description of how to use tags for blue/green and a/b deployments

Recommended Hands-on Labs

- Deploy Your Website on Cloud Run
  - In this lab you will:

Cloud Run Deployment Patterns



After your web application is packaged into a container image, you can run it anywhere. On Google Cloud, you can run containers on a Compute Engine in a virtual machine, or on a Kubernetes cluster, or on Cloud Run. On your local machine, you can use Docker or Podman (both are container runtimes).

Review: Cloud Run provides Containers-as-a-Service

- Serverless platform that runs individual containers
- Based on the open-source knative project (https://knative.dev/)
- Two deployment methods
  - Fully managed
    - Use when want Google Cloud to manage autoscaling, connectivity, high availability, etc.
  - Cloud Run for Anthos
    - Knative running on a GKE cluster
      - Use when need to:
        - Access VPC network
        - Tune size of compute engine instance, use GPUs, etc.
        - Are running knative containers on-premise or in other clouds and want a single pane of glass (Anthos) to manage them

Cloud Run
https://cloud.google.com/run

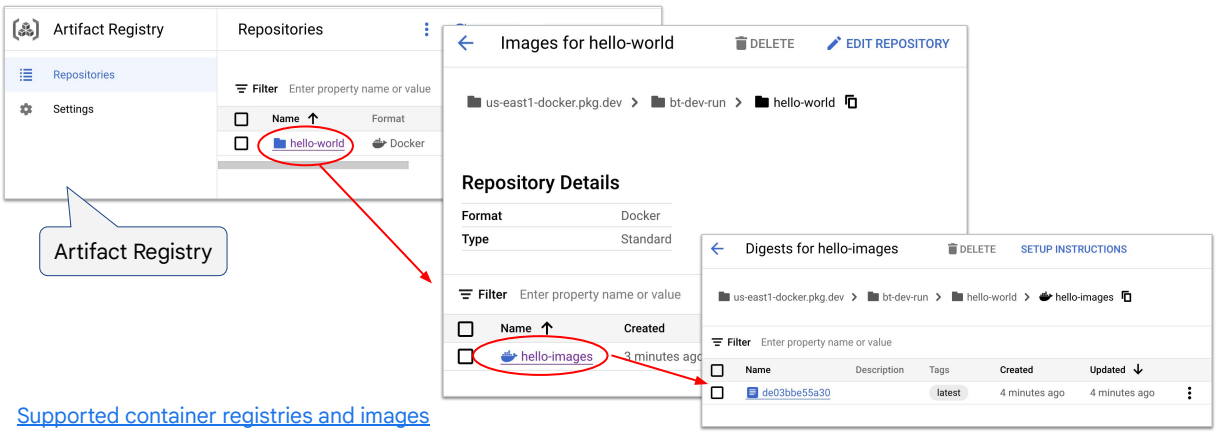
Knative
https://knative.dev/
https://cloud.google.com/knative

Cloud Run: What no one tells you about Serverless (and how it's done)
https://cloud.google.com/blog/topics/developers-practitioners/cloud-run-story-serverless-containers

Choosing between Cloud Run and Cloud Run for Anthos:
https://cloud.google.com/anthos/run/docs/choosing-a-platform

Cloud Run supports various container registries

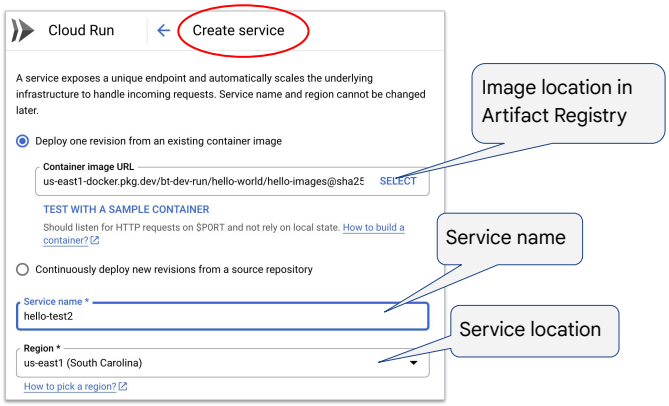
- Images can be stored in Artifact Registry (**recommended**), Container Registry (Deprecated), or Docker Hub



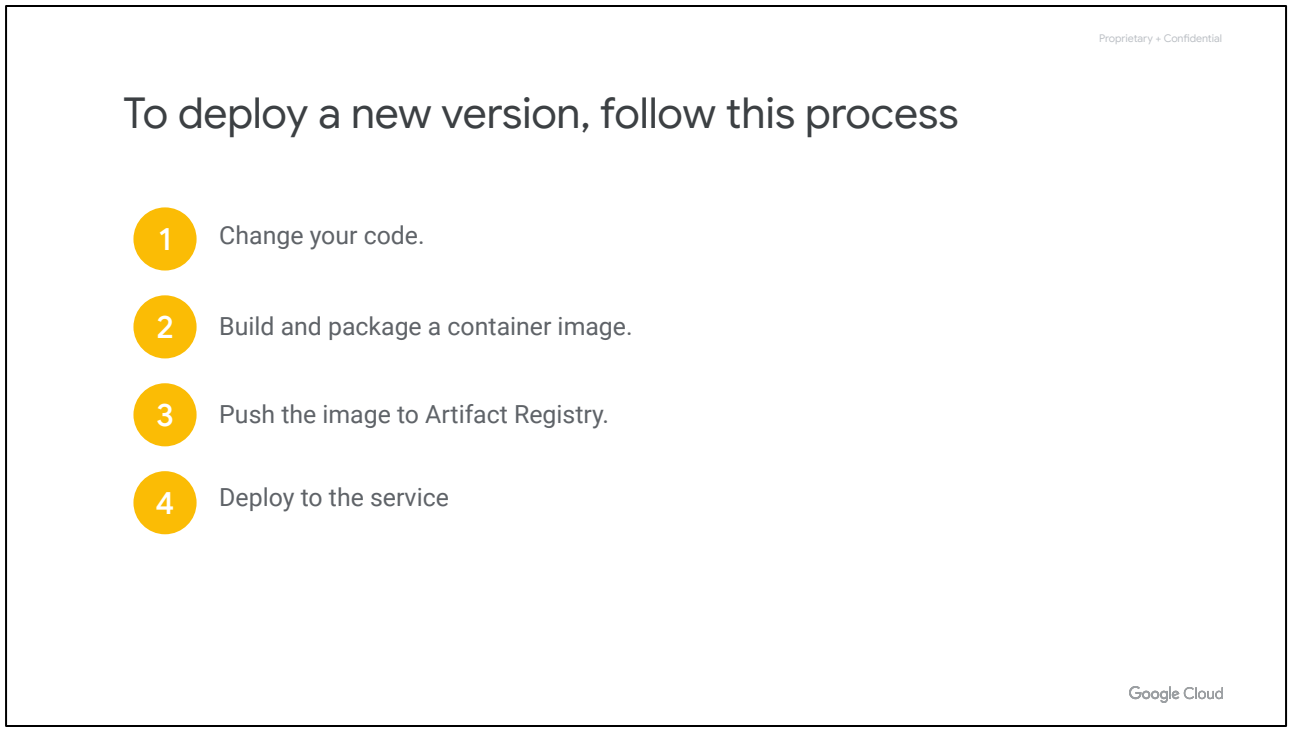
Supported container registries and images

Review: Deploying to Cloud Run creates a service

- Each service gets a unique and permanent URL that does not change over time as new revisions are deployed




```
gcloud run deploy hello-test2 \
--image us-east1-docker.pkg.dev/bt-dev-run/hello-world/hello-images:latest
```

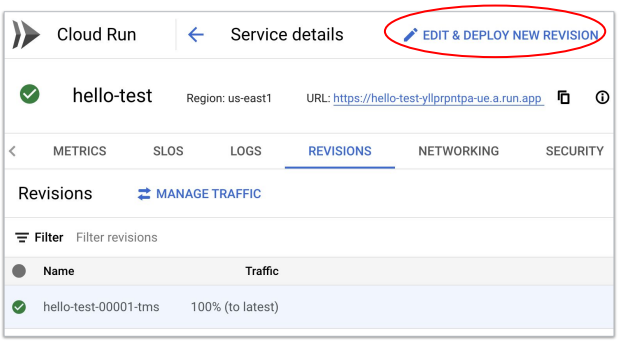


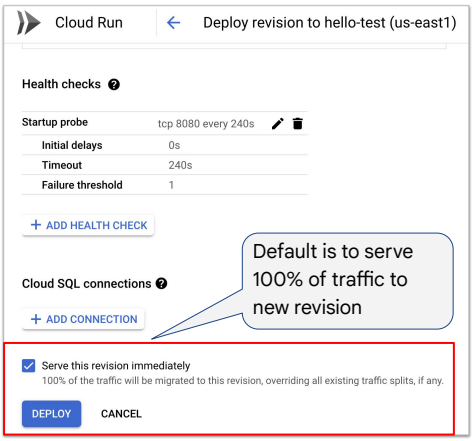
Let’s talk about the process you use to update your application on Cloud Run.

1. Change your application source code
2. Build and package your application into a container image
3. Push the container image to Artifact Registry
4. Deploy to the service

Application updates are created as new revisions

- When a new revision is deployed, choices are
  - Serve 100% of traffic
  - Serve no traffic

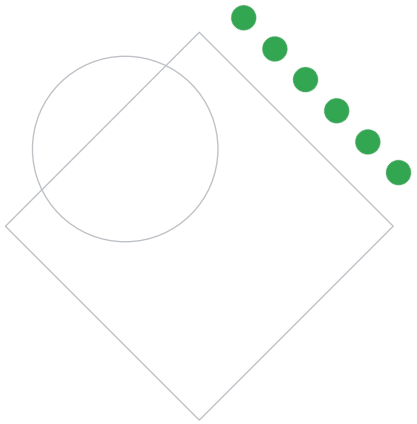





```
gcloud run deploy \
--image us-east1-docker.pkg.dev/bt-dev-run/hello-world/hello-images:latest \
--no-traffic
```

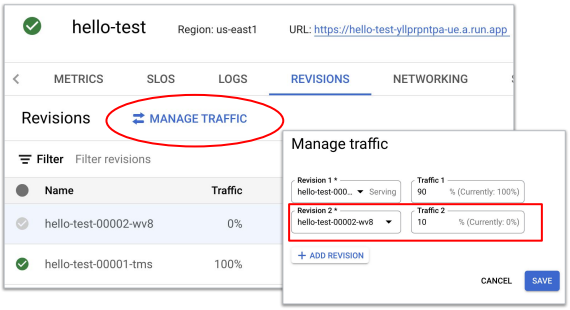
Cloud Run deployment patterns

- Cloud Run review
- **Canary testing**
- Blue/Green and A/B testing



Cloud Run canary testing

- Deploy a new revision
  - Send no traffic to it
- Use the traffic splitting feature to send a percentage of traffic to the new revision
  - Over time, repeat the step, sending more and more traffic to the canary



Send 10% of traffic to latest revision

```
gcloud run services update-traffic hello-test
--to-revisions hello-test-00002-wv8=10
```

Send all traffic to latest revision

```
gcloud run services update-traffic hello-test
--to-latest
```

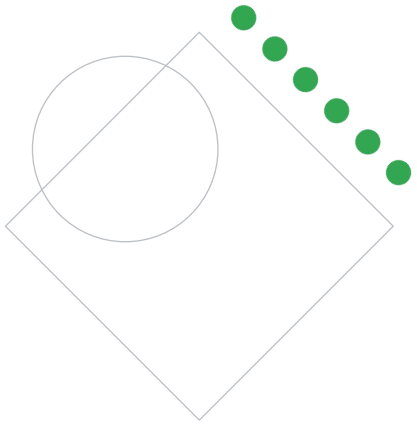
Roll-back to a specific revision

```
gcloud run services update-traffic hello-test
--to-revisions [REVISION_NAME]=100
```

Rollbacks, gradual rollouts, and traffic migration

Cloud Run deployment patterns

- Cloud Run review
- Canary testing
- **Blue/Green and A/B testing**



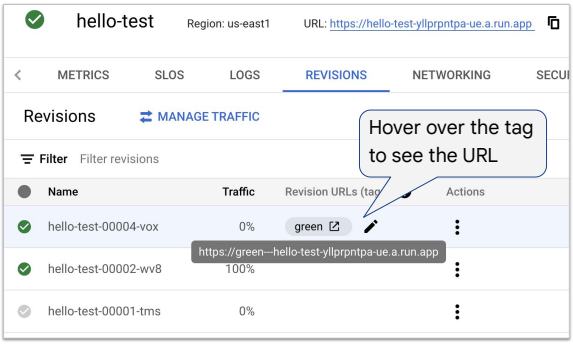
Cloud Run Blue/Green and A/B testing

- Assign a tag to a revision during deployment to access it at a specific URL without serving traffic

```
gcloud run deploy \
--image us-east1-docker.pkg.dev/bt-dev-run/hello-world/hello-images:latest \
--no-traffic --tag green
```

RESULT:

```
Service [hello-test] revision
[hello-test-00004-vox] has been deployed and is
serving 0 percent of traffic.
The revision can be reached directly at
https://green---hello-test-yllprpntpa-ue.a.run.
app
```



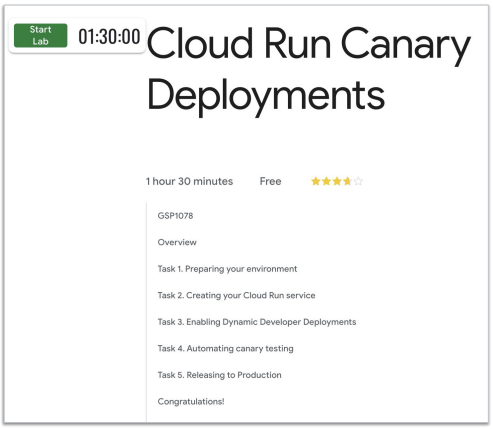
- To migrate traffic

```
gcloud run services update-traffic hello-test \
 --to-tags green=100
```

https://cloud.google.com/run/docs/rollouts-rollbacks-traffic-migration#tags

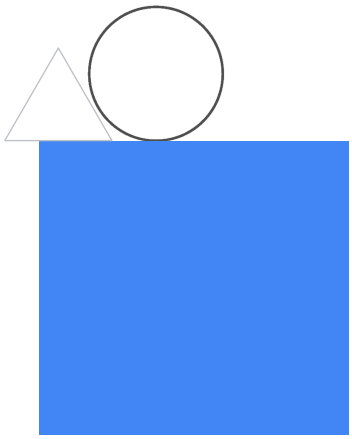
Using traffic splitting with Blue/Green deployments

- In this lab you will learn how to create a deployment pipeline for Cloud Run that
  - Executes a progression of code from developer branches to production
    - You will implement automated canary testing and percentage based traffic management



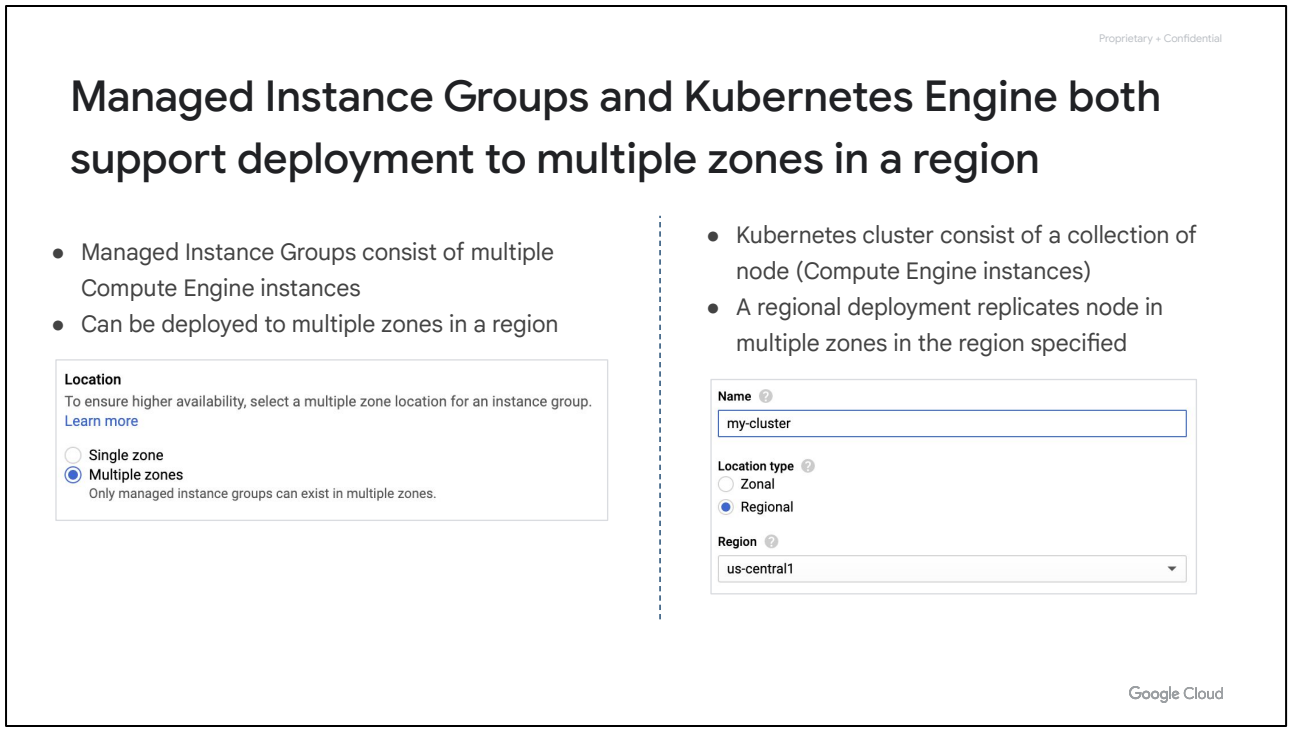
https://partner.cloudskillsboost.google/catalog_lab/5448

Patterns for scalable and resilient apps



Design for high availability

- Availability is a measure of the fraction of time that a service is usable.
  - Often used as a key indicator of overall service health
- Highly available architectures aim to maximize service availability, typically through physically distributing redundant resources
- Google Cloud services are available in locations across the globe
  - Locations are divided into regions and zones
  - How you deploy your app across these regions and zones affects the availability, latency, and other properties of your app



The multiple zone MIG and GKE cluster ensures that VMs are evenly distributed across the available zones.

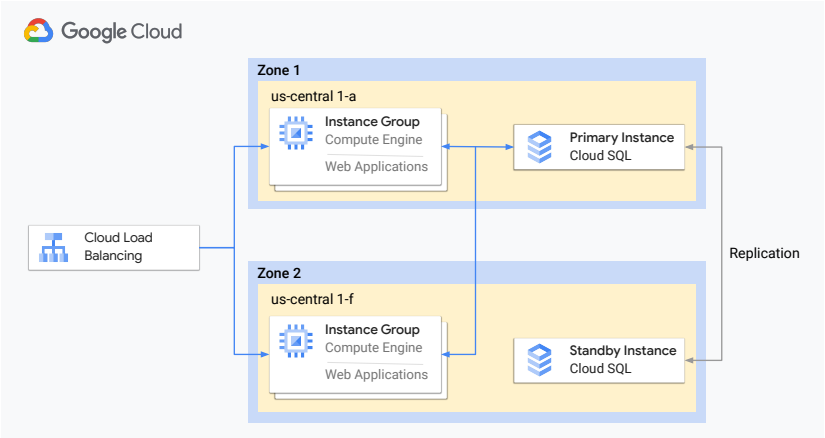
It is possible to use multiple Google Kubernetes Engine clusters and/or MIGs to host applications for better resilience, scalability, isolation, and compliance. In addition, users expect low-latency access to applications from anywhere around the world. It is possible to configure Cloud Load Balancing for both MIG and GKE environments. This allows traffic to be served from region closest to the end-user using a single anycast IP address, taking advantage of Google Cloud's 180+ Points of Presence and global network.

For more information about how Cloud Load Balancing handles cross-region traffic for GKE, see https://cloud.google.com/kubernetes-engine/docs/how-to/multi-cluster-ingress.

Istio can also be used to integrate multiple clusters in different regions. For more details, see https://istio.io/docs/examples/platform/gke/#create-the-gke-clusters.

Highly available Managed Instance Group example

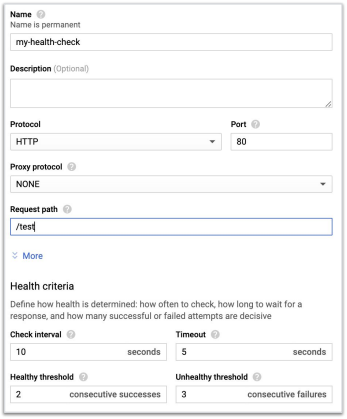
- Deploy multiple servers to multiple zones in a region
- If using a backend database
  - Create a failover database in another zone or use a distributed database like Firestore or Spanner



When you're using Compute Engine, for higher availability you can use a regional instance group, which provides built-in functionality to keep instances running. Use auto-healing with an application health check and load balancing to distribute load. For data, the storage solution selected will affect what is needed to achieve high availability. For Cloud SQL, the database can be configured for HA, which provides data redundancy and a standby instance of the database server in another zone.

Utilize health checks to ensure the availability of applications and services

- Heath checks are supported by the following services
  - Managed Instance Groups
  - Cloud Run
  - Kubernetes Engine (node level and pod level)
- Probes verify that the service is responsive
- If health check fails, a new instance is created to replace the unresponsive one.
- Load balancers also use health checks to ensure that they send requests only to healthy instances.



Managed Instance Group health check

The test endpoint is there to indicate that the service is available and ready to accept requests. A challenge here is that if dependent services are not available, this can lead to a cascading failure effect.

Create scalable services to adjust capacity to meet demand

- Scalability is the measure of a system's ability to handle varying amounts of work by adding or removing resources from the system
- Google Cloud provides products and features to help you build scalable, efficient apps:
  - Compute Engine virtual machines and Google Kubernetes Engine (GKE) clusters integrate with autoscalers that let you grow or shrink resource consumption based on metrics that you define.
  - Google Cloud's serverless platform provides managed compute, database, and other services that scale quickly from zero to high request volumes, and you pay only for what you use.
  - Database products like BigQuery, Cloud Spanner, and Cloud Bigtable can deliver consistent performance across massive data sizes.
  - Cloud Monitoring provides metrics across your apps and infrastructure, helping you make data-driven scaling decisions.

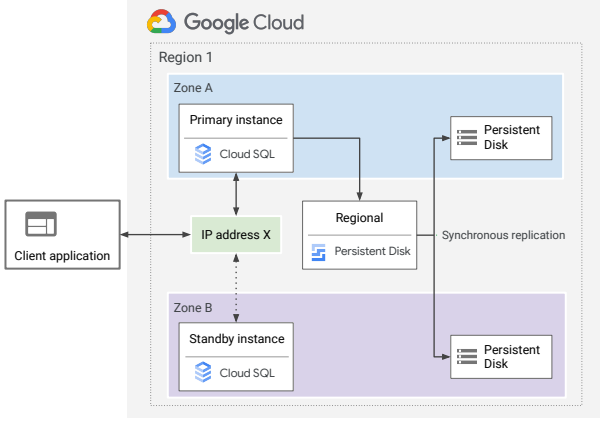
Favor managed services

- Use managed services to consume parts of your application stack as services
  - Rather than independently installing, supporting, and operating all parts of your application stack
- For example, use a MySQL database provided by Cloud SQL instead of installing and managing a MySQL database on virtual machines (VMs)
  - Google Cloud manages data replication, backups, and the underlying infrastructure
    - You spend less time managing infrastructure, and more time on improving the reliability of your app.
- Managed services offer built-in redundancy, helping you meet your availability goals
  - Some are deployed to a specific region and are automatically redundant across all the zones within that region
  - Others offer multi-regional availability and can withstand the loss of an entire region

https://cloud.google.com/architecture/scalable-and-resilient-apps#favor_managed_services

If using Cloud SQL, create a failover replica for high availability

- Replica will be created in another zone in the same region as the database.
- Will automatically switch to the failover if the primary instance is unavailable.
- Doubles the cost of the database.



The HA configuration is available for all three Cloud SQL offerings: MySQL, PostgreSQL, and SQL Server. Full details for the configuration are here: https://cloud.google.com/sql/docs/mysql/high-availability

The primary instance and standby share the same IP address so that in the event of failure, clients will seamlessly be redirected to the standby.

Spanner and Firestore can be deployed in 1 or multiple regions

Database	Availability SLA
Firestore single region	99.99%
Firestore multi-region	99.999%
Spanner single region	99.99%
Spanner multi-region (nam3)	99.999%

Both Firestore and Spanner offer single and multi-region deployment. Multi-region locations can withstand the loss of entire regions and maintain availability without losing data. However, multi-region deployments also cost more money. Which you choose ties back to your application requirements and service-level objectives.

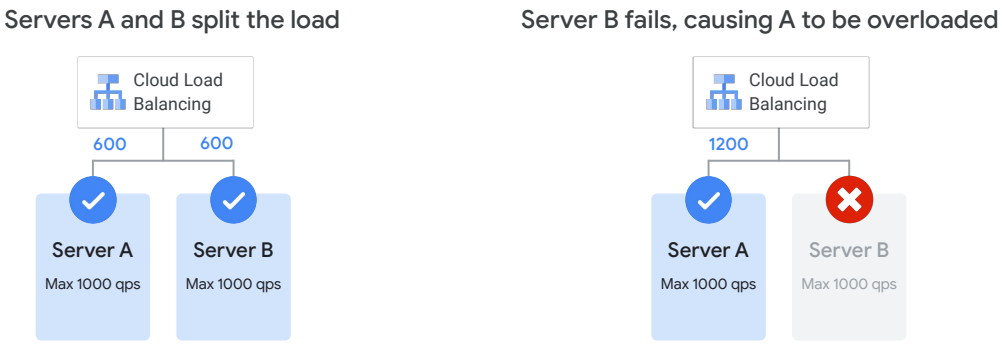
Implement caching

- A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer
- Supports improved scalability by reducing reliance on disk-based storage
- Request latencies to the storage layer are reduced since requests are served from memory
  - Reduces the load on services that are downstream of your app, especially databases
- Caching can also increase resiliency by supporting techniques like graceful degradation
  - If the underlying storage layer is overloaded or unavailable, the cache can continue to handle requests
- Cloud Memorystore is a fully managed service that is powered by the Redis or Memcached in-memory datastore
  - Can be deployed in a high-availability configuration that provides cross-zone replication and automatic failover.

https://cloud.google.com/architecture/scalable-and-resilient-apps#implement_caching

Beware of cascading failures

Cascading failures occur when one system fails, causing others to be overloaded, such as a message queue becoming overloaded because of a failing backend.



Cascading failures often occurs when a failure in one part of the system increases the probability that other portions of the system will fail. There are a few typical scenarios that account for the majority of cascading failures:

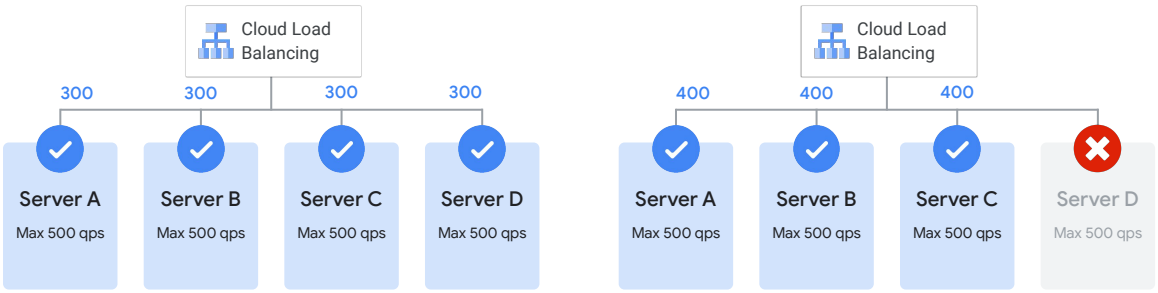
Server overload: When one server fails and others become overloaded by handling the capacity the failed server would handle under normal circumstances.

Resource exhaustion: CPU, memory, threads, and file descriptors all can cause a server to become unstable and crash.

Some strategies for preventing these are discussed on the next slide.

To avoid cascading failures

- Use health checks in Compute Engine or readiness and liveliness probes in Kubernetes to detect and then repair unhealthy instances.
- Ensure that new server instances start fast and ideally don't rely on other backends/systems to start up.



Cascading failures may be handled by support from the deployment platform with, for example, Kubernetes and the desired state. It is always important to have an efficient startup sequence for services, and they should start up even if dependencies are not available but just not make themselves available to process requests.

Other strategies should also be considered. For instance, to prevent server overload, consider serving degraded results, load shedding, or graceful degradation. Servers should try to prevent themselves from becoming overloaded. Also consider implementing rate limiting on services to prevent overload.

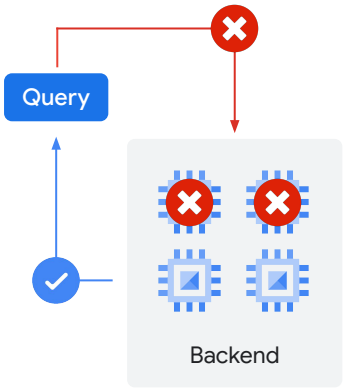
The importance of good capacity planning cannot be overestimated. The load at which a service will fail should be known so that the deployment can be suitably provisioned.

Query of death overload

- Term used to describe a query that, when executed, can cause a system to crash, hang, or become unresponsive
  - The query may consume excessive resources, run for an extended period, or process an enormous amount of data, leading to high costs or system instability

Problem

Business logic error shows up as overconsumption of resources, and the service overloads.



Solution

Requires good monitoring, observability, and logging to determine the root cause of the problem. Ensure that notification of these issues gets back to the developers.

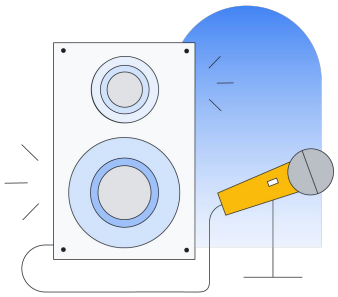
In the query of death, a request is made to a service that may cause a failure in the service. This is referred to as the query of death because the error manifests itself as overconsumption of resources, but in reality is due to an error in the business logic itself. This can be difficult to diagnose and requires good monitoring, observability, and logging to determine the root cause of the problem.

Retry overload failure

- Challenge with issuing retries is that you may not know why the first request failed
  - Is it because the service you called is overloaded?
    - If this is the case, a retry can add more load to an already overloaded service
      - Callers may start issuing retries because you have not replied in a given time period
        - Your attempt to build in resilience actually caused system instability

Problem

You try to make the system more reliable by adding retries, and instead you create the potential for an overload.



Solution

Consider the truncated exponential backoff or the circuit breaker patterns to protect the service from too many retries

The challenge with issuing a retry is that you do not know why the first request failed. Is it because the service you called is overloaded? If this is the case, the chances are that a retry can add more load to an already overloaded service. Not only that, your callers may start issuing retries to you because you have not replied in a given time period, and as a result trying to build in resilience actually causes system instability. If retries are to be used, some recommendations are on the next slide.

Use truncated exponential backoff pattern to avoid retry overload

If service invocation fails, try again:

- Continue to retry, but wait a while between attempts
- Wait a little longer each time the request fails
- Set a maximum length of time and a maximum number of requests
- Eventually, give up

Example:

- Request fails; wait 1 second + random_number_milliseconds and retry
- Request fails; wait 2 seconds + random_number_milliseconds and retry
- Request fails; wait 4 seconds + random_number_milliseconds and retry
- And so on, up to a maximum_backoff time
- Continue waiting and retrying up to some maximum number of retries

Always use exponential backoff when scheduling retries. Retries should be randomly distributed over the retry window; otherwise, retries can be scheduled at the same time from different clients, which causes more load on the struggling service.

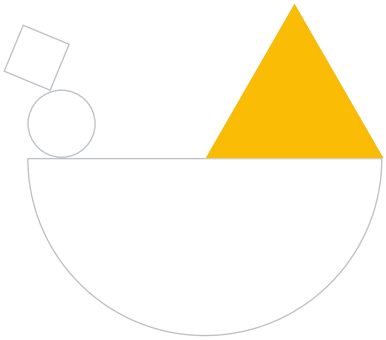
Use the circuit breaker pattern to protect the service from too many retries

- Plan for degraded state operations
- If a service is down and all its clients are retrying, the increasing number of requests can make matters worse.
  - Protect the service behind a proxy that monitors service health (*the circuit breaker*)
  - If the service is not healthy, don't forward requests to it.
- If using GKE, leverage Anthos Service Mesh/Istio to automatically implement circuit breakers
- If using Managed Instance Groups with Load Balancing, use Google Cloud Monitoring along with Cloud Functions or Cloud Run to detect anomalies or increased error rates
  - When certain conditions are met, automated actions (like adjusting traffic policies or sending alerts) can be triggered

The circuit breaker pattern https://martinfowler.com/bliki/CircuitBreaker.html is a recognized solution to preventing too many retries. It is built into some infrastructure; for example, Istio. Also, several libraries in different languages provide support. For instance, in Java, the Microprofile and Spring frameworks provide support and libraries such as resilience4j (https://github.com/resilience4j/resilience4j)

Similar libraries are available for many other languages.

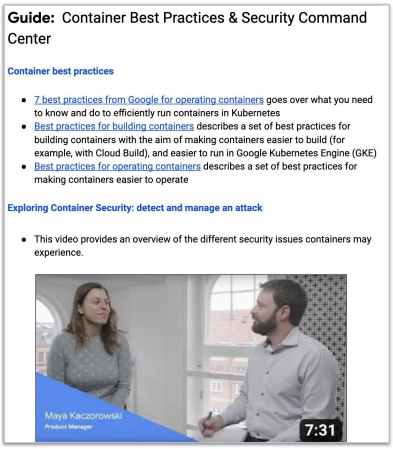
Container Best Practices & Security Command Center



Guide: Container Best Practices & Security Command Center

Topics covered include

- Security Command Center (SCC) overview
- Using SCC to detect web app vulnerabilities
- Using SCC to detect container vulnerabilities
- Using Binary Authorization to ensure only trusted images are deployed to containers
- Container best practices



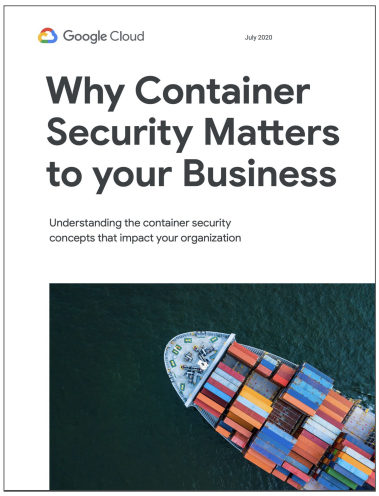
Container Best Practices & Security Command Center

Container Security

Slide from Module 6

Shared Responsibility Model

- The security of cloud services is a shared responsibility between the Google and the user
- Security and incident response teams need to understand what they are responsible for hardening and protecting vs Google's responsibility



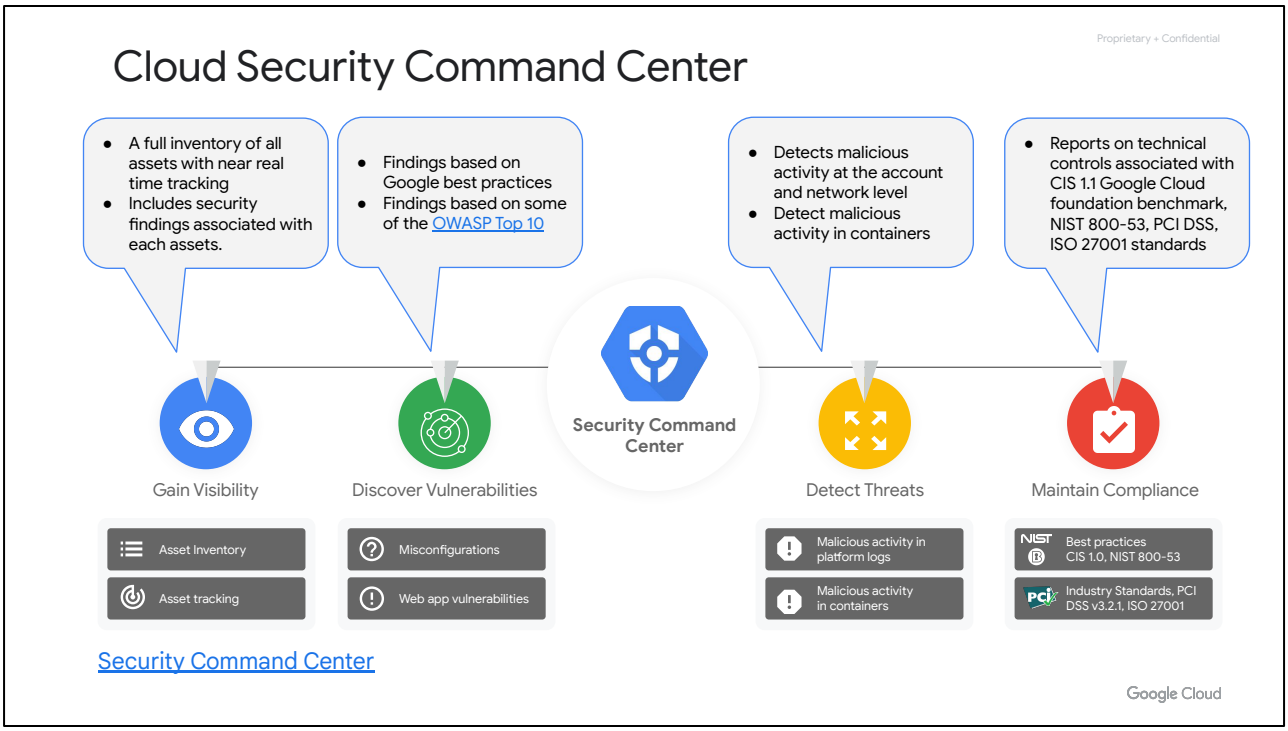
https://services.google.com/fh/files/misc/why_container_security_matters_to_your_business.pdf

Key security concerns related to containers

- Safe Origin
  - How do you ensure that all container images running in the cluster come from an approved source?
- Consistency and Validation
  - How do you ensure that all desired validation steps were completed successfully for every container build and every deployment?
- Integrity
  - How do you ensure that containers were not modified before running after their provenance was proven?

Unvalidated container images present several risks

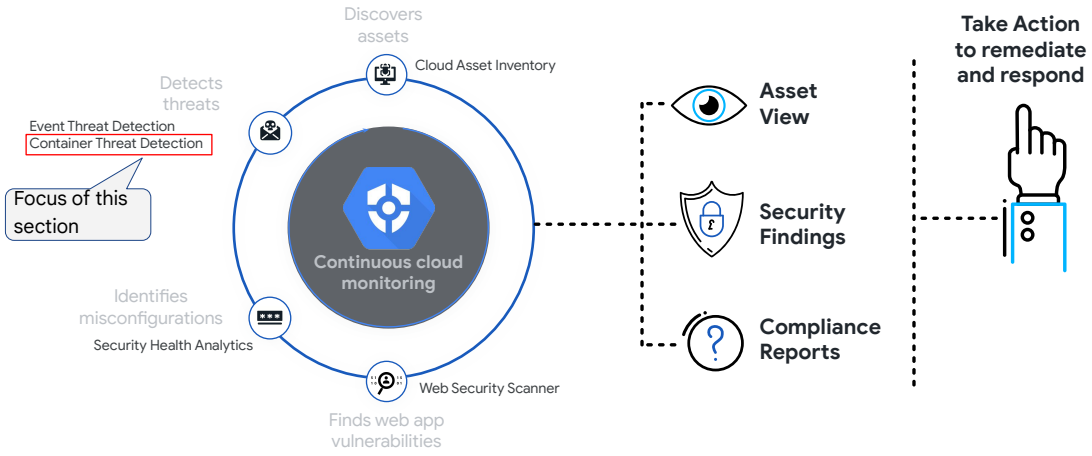
- A malicious actor with a compromised a container may be able to obtain sufficient cluster privileges to launch other containers from an unknown source without enforcement
- An authorized user with the permissions to create pods may be able to accidentally or maliciously run an undesired container directly inside a cluster.
- An authorized user may accidentally or maliciously overwrite a docker image tag with a functional container that has undesired code silently added to it
  - Kubernetes will pull and deploy that container as a part of a deployment automatically



Security Command Center helps you

- Understand the number of projects you have, what resources are deployed, and manage which service accounts have been added or removed.
- Identify security misconfigurations and compliance violations and resolve them by following actionable recommendations.
- Uncover threats targeting your resources using logs and powered by threat intelligence
- Use kernel-level instrumentation to identify potential compromises of containers.

Security Command Center - How it works



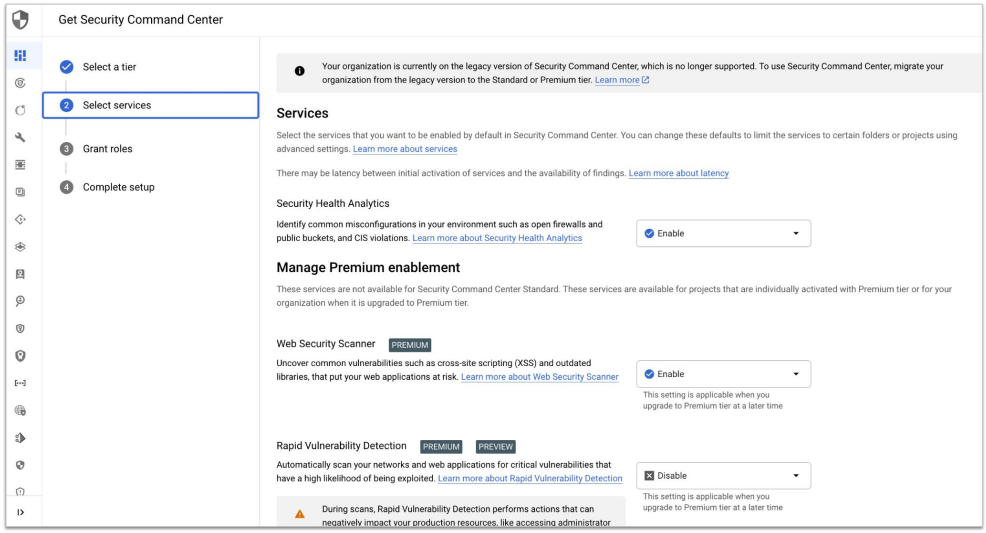
[Security Command Center Evaluation Guide]

SCC Container Threat Detection detects runtime attacks in near real time

- Continuously monitors the state of Container-Optimized OS node images
  - Compute Engine and Google Kubernetes engine
- Evaluates all changes and remote access attempts. Detects things such as:
  - Added Binary Executed
    - Detects if a binary not part of the original container image was executed
  - Added Library Loaded
    - Detects if a library not part of the original container image was loaded.
  - Malicious Script Executed
    - Uses NLP techniques to evaluate executed bash scripts and identify them as malicious.
  - Malicious URL Observed
    - Checks URLs in running processes against unsafe web resources maintained by the Google Safe Browsing service

Container Threat Detection conceptual overview

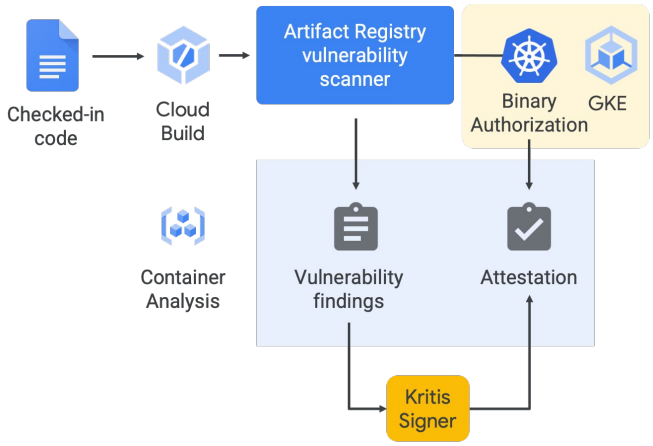
Accessing Security Command Center



SCC detects compromised containers after they have been deployed.
Wouldn't it better to detect issues before they were deployed?

Securing workloads with Binary Authorization

- Enforce deploying only trusted containers into GKE
  - Enable Binary Authorization on your GKE cluster
  - Add a policy that requires signed images
  - When an image is built by Cloud Build an “attestor” verifies that it was from a trusted repository (Cloud Source Repositories, for example)
  - Artifact Registry includes a vulnerability scanner that scans containers



Binary Authorization is a Google Cloud managed service that works closely with GKE to enforce deploy-time security controls to ensure that only trusted container images are deployed. With Binary Authorization you can allowlist container registries, require images to be signed by trusted authorities, and centrally enforce those policies. By enforcing this policy, you can gain tighter control over your container environment by ensuring only approved and/or verified images are integrated into the build-and-release process.

Demo: Google Kubernetes Engine Security: Binary Authorization

- This lab deploys a Kubernetes Engine Cluster with the Binary Authorization feature enabled demonstrates how to allowlist approved container registries, and walks you through the process of creating and running a signed container.

Binary Authorization is a Google Cloud managed service that works closely with GKE to enforce deploy-time security controls to ensure that only trusted container images are deployed. With Binary Authorization you can allowlist container registries, require images to be signed by trusted authorities, and centrally enforce those policies. By enforcing this policy, you can gain tighter control over your container environment by ensuring only approved and/or verified images are integrated into the build-and-release process.


Google Cloud