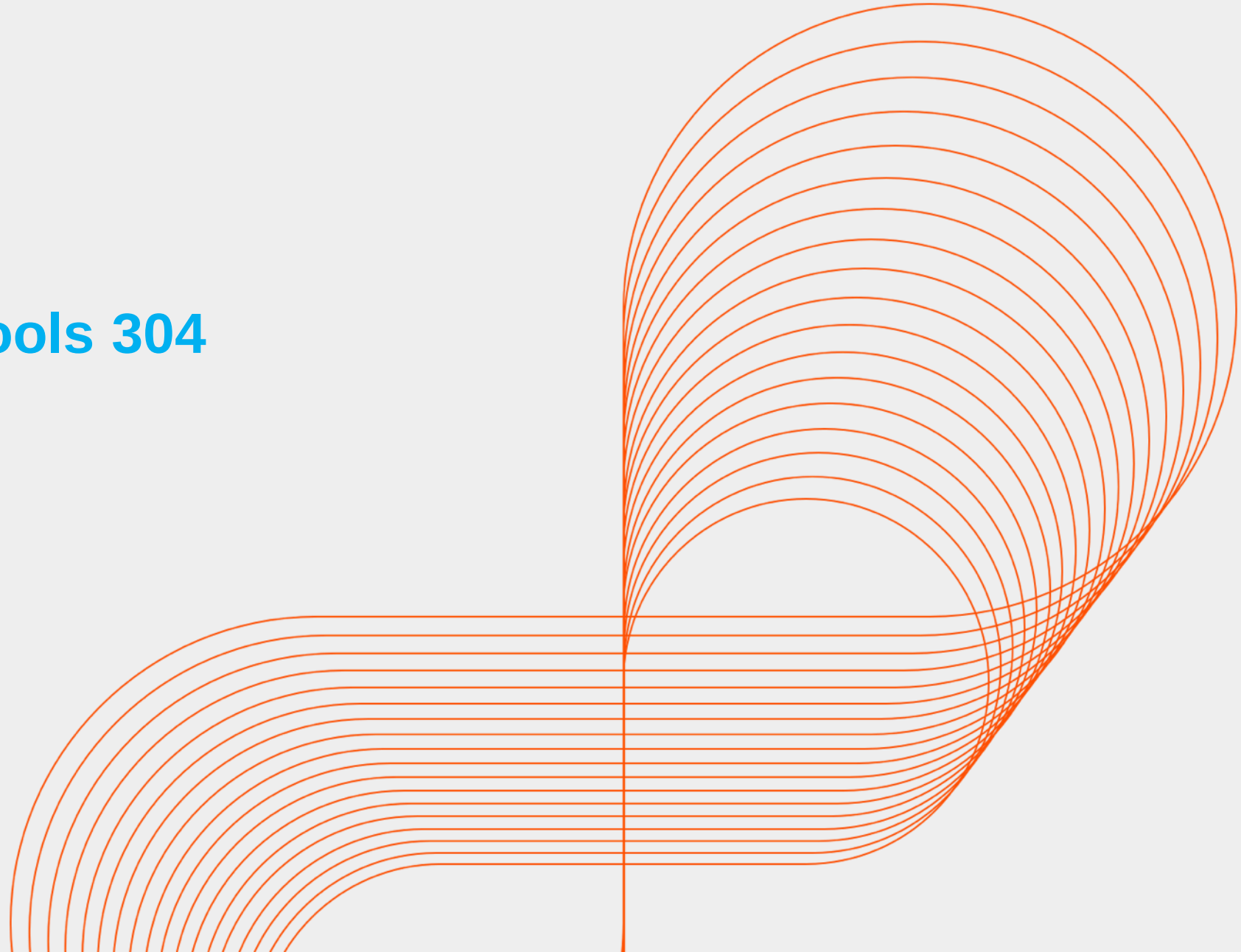
Persistent

Selfie Shots - Tools 304
Kubernetes

Persistent University



Services Overview

Key learning points :

- Services Overview
- Service Types

Persistent

Services Overview

- As touched on previously, the Kubernetes architecture is built on the concept of transient, decoupled objects connected together. Services are the agents which connect Pods together, or provide access outside of the cluster, with the idea that any particular Pod could be terminated and rebuilt. Typically using Labels, the refreshed Pod is connected and the microservice continues to provide the expected resource via an Endpoint object. Google has been working on **Extensible Service Proxy** (**ESP**), based off the **nginx** HTTP reverse proxy server, to provide a more flexible and powerful object than Endpoints, but **ESP** has not been adopted much outside of the **Google App Engine** or **GKE** environments.
- There are several different service types, with the flexibility to add more, as necessary. Each service can be exposed internally or externally to the cluster. A service can also connect internal resources to an external resource, such as a third-party database.
- The **kube-proxy** agent watches the Kubernetes API for new services and endpoints being created on each node. It opens random ports and listens for traffic to the **ClusterIP:Port**, and redirects the traffic to the randomly generated service endpoints.
- Services provide automatic load-balancing, matching a label query. While there is no configuration of this option, there is the possibility of session affinity via IP. Also, a headless service, one without a fixed IP nor load-balancing, can be configured.
- Unique IP addresses are assigned, and configured via the **etcd** database, so that Services implement **iptables** to route traffic, but could leverage other technologies to provide access to resources in the future.

Service Update Pattern

- Labels are used to determine which Pods should receive traffic from a service. As we have learned, labels can be dynamically updated for an object, which may affect which Pods continue to connect to a service.
- The default update pattern is for a rolling deployment, where new Pods are added, with different versions of an application, and due to automatic load balancing, receive traffic along with previous versions of the application.
- Should there be a difference in applications deployed, such that clients would have issues communicating with different versions, you may consider a more specific label for the deployment, which includes a version number. When the deployment creates a new replication controller for the update, the label would not match. Once the new Pods have been created, and perhaps allowed to fully initialize, we would edit the labels for which the Service connects. Traffic would shift to the new and ready version, minimizing client version confusion.

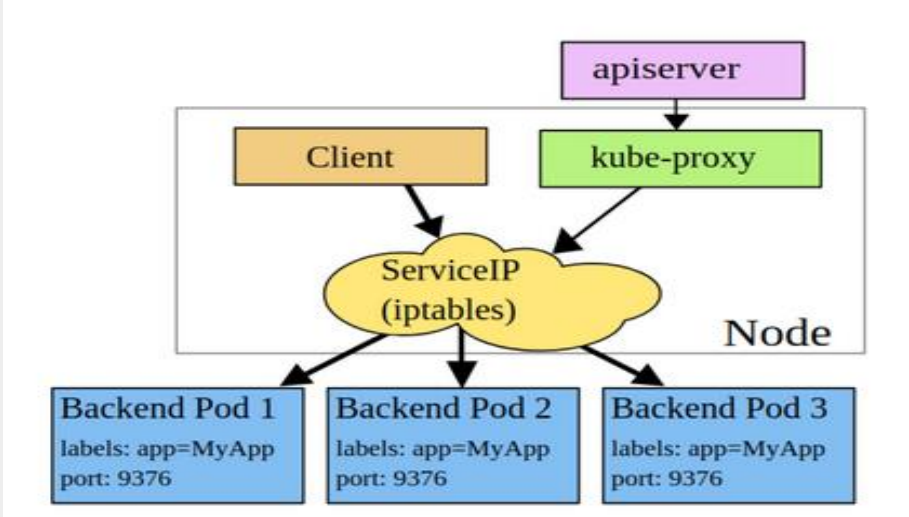
Service Types

- Services can be of the following types:
  - The **ClusterIP** service type is the default, and only provides access internally (except if manually creating an external endpoint). The range of ClusterIP used is defined via an API server startup option.
  - The **NodePort** type is great for debugging, or when a static IP address is necessary, such as opening a particular address through a firewall. The NodePort range is defined in the cluster configuration.
  - The **LoadBalancer** service was created to pass requests to a cloud provider like GKE or AWS. Private cloud solutions also may implement this service type if there is a cloud provider plugin, such as with **CloudStack** and **OpenStack**. Even without a cloud provider, the address is made available to public traffic, and packets are spread among the Pods in the deployment automatically.
  - A newer service is **ExternalName**, which is a bit different. It has no selectors, nor does it define ports or endpoints. It allows the return of an alias to an external service. The redirection happens at the DNS level, not via a proxy or forward. This object can be useful for services not yet brought into the Kubernetes cluster. A simple change of the type in the future would redirect traffic to the internal objects.

The **kubectl proxy** command creates a local service to access a ClusterIP. This can be useful for troubleshooting or development work.

kube-proxy

- The kube-proxy running on cluster nodes watches the API server service resources. It presents a type of virtual IP address for services other than ExternalName. The mode for this process has changed over versions of Kubernetes.
- In v1.0, services ran in userspace mode as TCP/UDP over IP or Layer 4. In the v1.1 release, the iptables proxy was added and became the default mode starting with v1.2.



- In the **iptables** proxy mode, **kube-proxy** continues to monitor the API server for changes in Service and Endpoint objects, and updates rules for each object when created or removed. One limitation to the new mode is an inability to connect to a Pod should the original request fail, so it uses a **Readiness Probe** to ensure all containers are functional prior to connection. This mode allows for up to approximately 5000 nodes. Assuming multiple Services and Pods per node, this leads to a bottleneck in the kernel.
- Another mode beginning in v1.9 is **ipvs**. While in beta, and expected to change, it works in the kernel space for greater speed, and allows for a configurable load-balancing algorithm, such as round-robin, shortest expected delay, least connection and several others. This can be helpful for large clusters, much past the previous 5000 node limitation. This mode assumes IPVS kernel modules are installed and running prior to **kube-proxy**.
- The **kube-proxy** mode is configured via a flag sent during initialization, such as **mode=iptables** and could also be **IPVS** or **userspace**

Accessing an Application with a Service

- The basic step to access a new service is to use **kubectl**.

```
$ kubectl expose deployment/nginx --port=80 --type=NodePort
$ kubectl get svc
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)  AGE
kubernetes   10.0.0.1     <none>        443/TCP  18h
nginx        10.0.0.112   <nodes>       80/TCP   5s
$ kubectl get svc nginx -o yaml
apiVersion: v1
kind: Service
...
spec:
    clusterIP: 10.0.0.112
    ports:
    - nodePort: 31230
...
```

- Open browser http://<Public IP>:31230

Accessing an Application with a Service

- The **kubectl expose** command created a service for the **nginx** deployment. This service used port 80 and generated a random port on all the nodes. A particular **port** and **targetPort** can also be passed during object creation to avoid random values. The **targetPort** defaults to the **port**, but could be set to any value, including a string referring to a port on a backend Pod. Each Pod could have a different port, but traffic is still passed via the name. Switching traffic to a different port would maintain a client connection, while changing versions of software, for example.
- The **kubectl get svc** command gave you a list of all the existing services, and we saw the **nginx** service, which was created with an internal cluster IP.
- The range of cluster IPs and the range of ports used for the random **NodePort** are configurable in the API server startup options.
- Services can also be used to point to a service in a different namespace, or even a resource outside the cluster, such as a legacy application not yet in Kubernetes.

Summary

At the end of this session, we see that you are now able to

- Explain Kubernetes services.
- Expose an application.
- Discuss the service types available.
- Start a local proxy.

Lab Exercise

- Expose deployments created as
  - ClusterIP service
  - NodePort service
  - LoadBalancer Service
- Deploy Kubernetes dashboard and access with local proxy.