Microservices architecture on Kubernetes

Feb , 2019 – Rakesh Gujjarlapudi
Senior Technologist

WHAT ARE MICROSERVICES

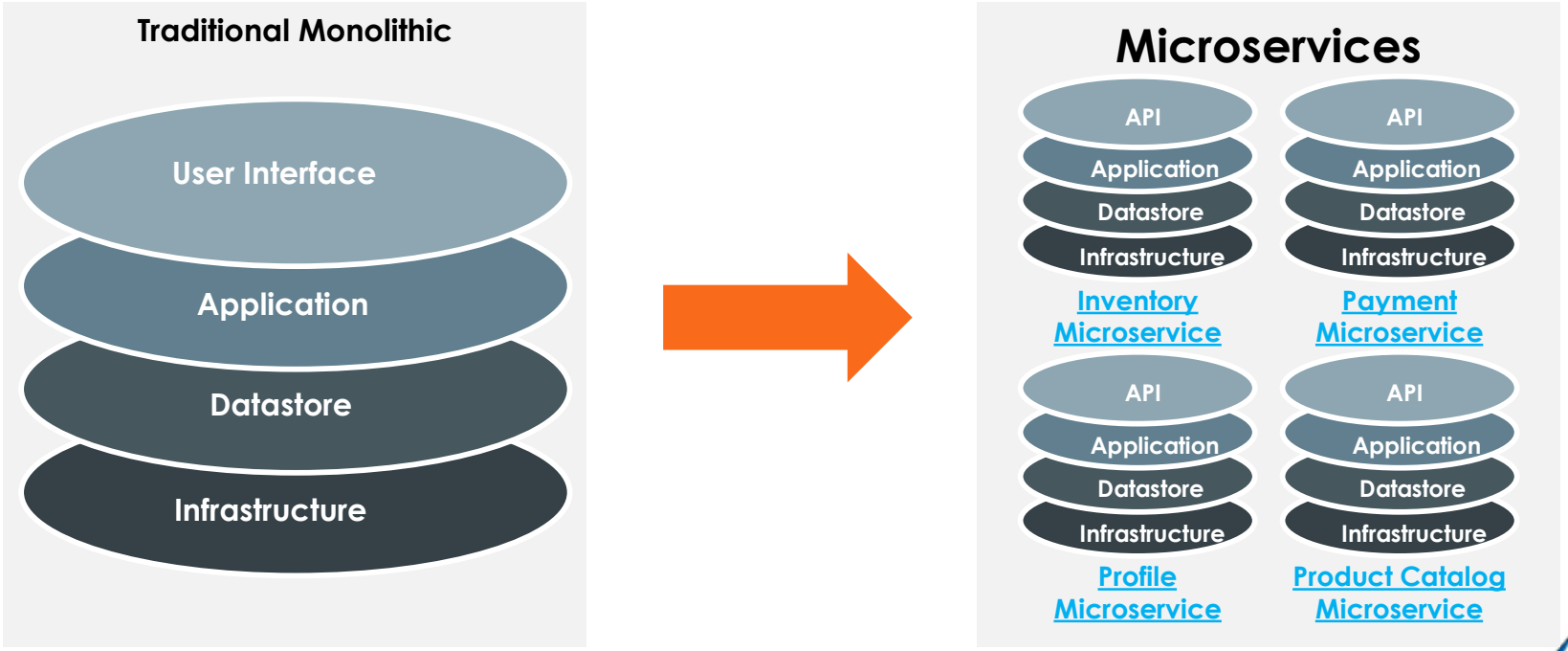
Minimal function services that are deployed separately but can interact together to achieve a broader use-case.

Monolithic Applications(Traditional)

- Single Monolithic Application
- Must Deploy Entire Application
- One Database for Entire Application
- Organized Around Technology Layers
- State In Each Runtime Instance
- One Technology Stack for Entire Application
- In-process Calls Locally, SOAP Externally

Microservices

- Many, Smaller Minimal Function Microservices
- Can Deploy Each Microservices Independently
- Each Microservices Often Has Its Own Datastore
- Organized Around Business Capabilities
- State is Externalized
- Choice Of Technology for Each Microservices
- REST Calls Over HTTP, Messaging or Binary

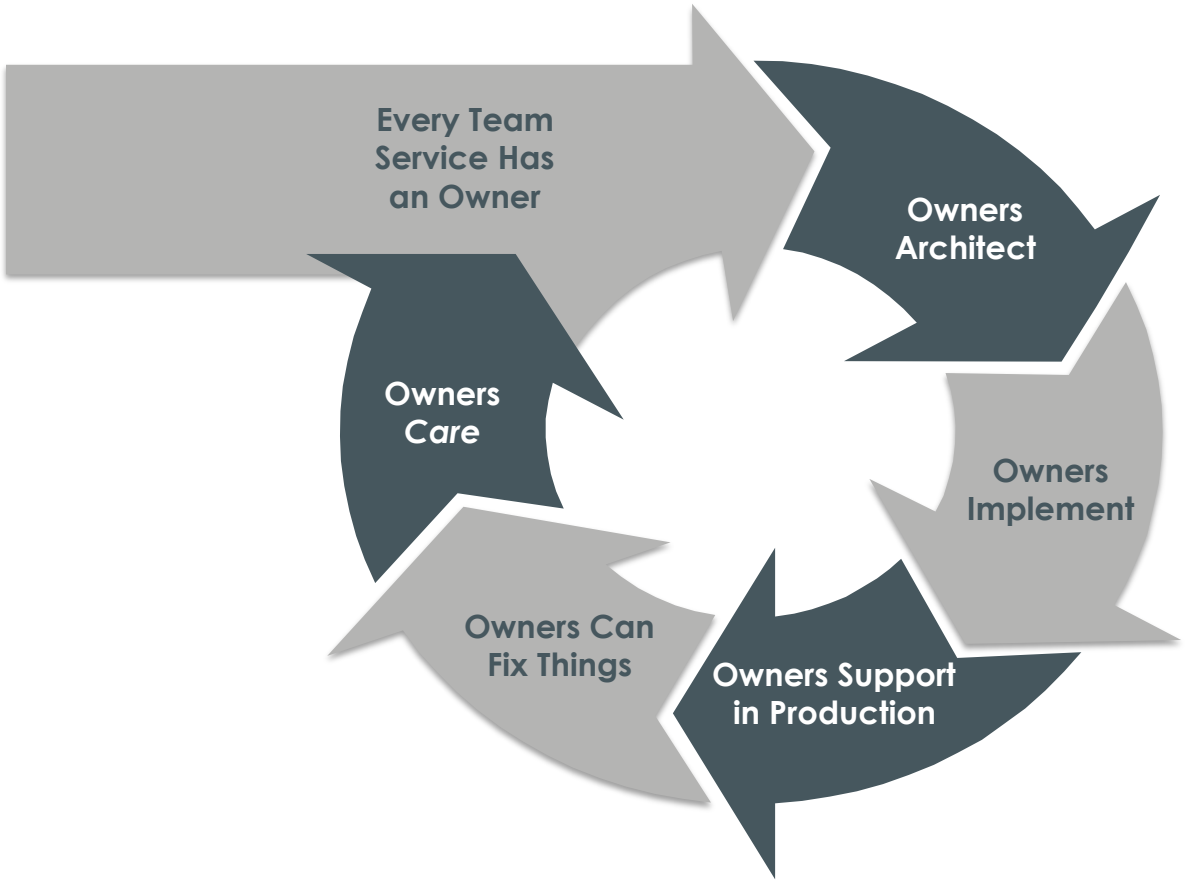


One Application

Many Small Microservices

MICROSERVICES OWNERSHIP

OWNERSHIP IS KEY TO THE SUCCESS OF MICROSERVICES



MICROSERVICES TRADEOFFS

Traditional App Development

Easier Deployment/Ops

- One big block of code, sometimes broken into semi-porous modules
- Complexity handled inside the big block of code
- Each big block is hard to develop but easy to deploy

Microservices

Easier Development

- Many small blocks of code, each developed and deployed independently
- Complexity encapsulated in each microservice
- Each microservice is easy to develop but hard to deploy

Common Microservice Adoption Use Cases

I want to **extend** my existing monolithic application by adding microservices on the periphery.

I want to build a **net new** microservices-style application from the ground up.

I want to **decompose** an existing modular application into a microservices-style application

SOA vs Microservices

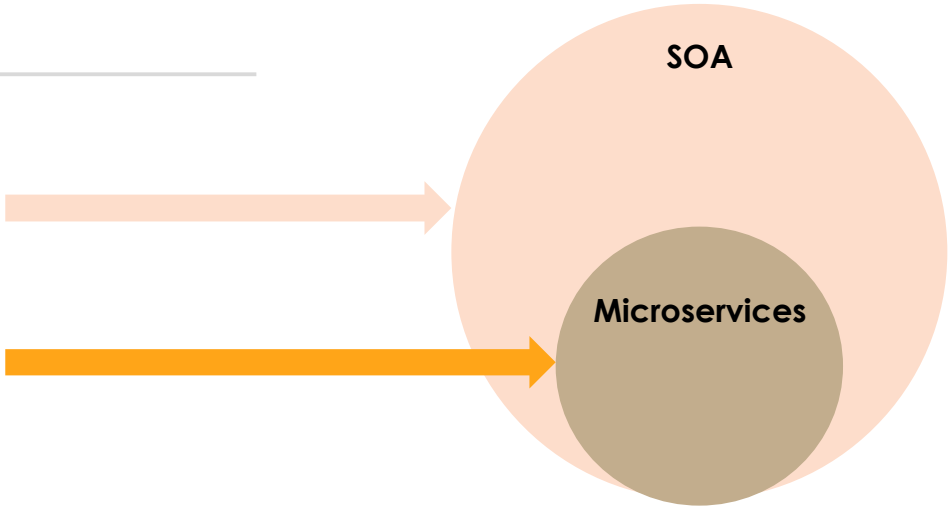
SOA is the general idea, where microservices are a very specific way of achieving it

All of the properties of SOA also apply to microservice
Keeping consumption of services separate from the provisioning of services
Separating infra management from the delivery of application capability
Separating teams and decoupling services

Implementation Differences

- Favors centralized orchestration
- Needlessly complicated by SOAP
- "Dumb endpoints, smart pipes"

- Favors distributed choreography
- REST + HTTP/S = simple
- "Smart endpoints, dumb pipes"



SOA vs Microservices Misconceptions

"Microservices removes the need for an Enterprise Service Bus"

Don't confuse the product with the pattern

"Microservices solves the problems of SOA"

Don't confuse improper SOA deployments as problems with SOA

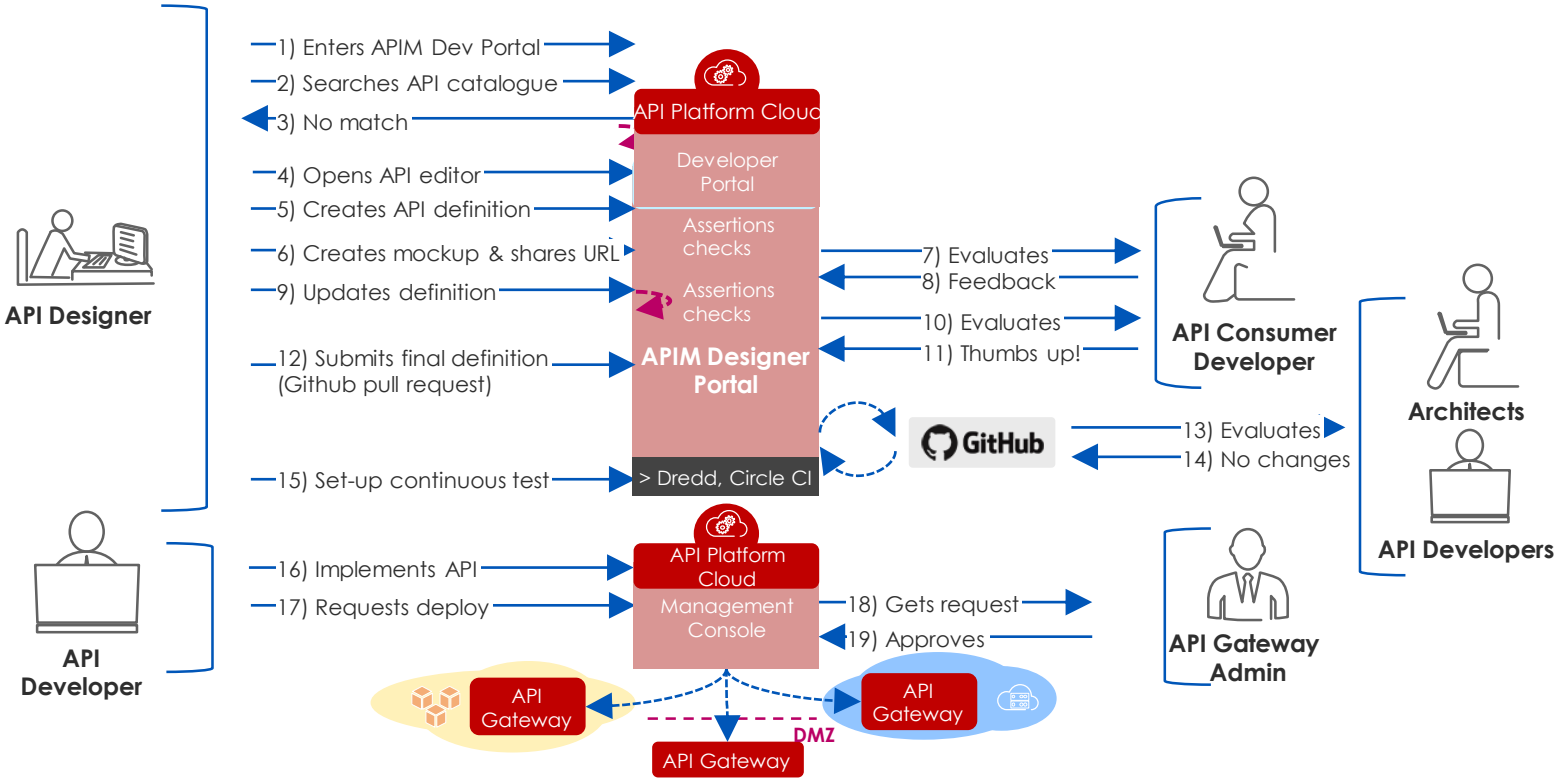
"Companies like Netflix and LinkedIn use microservices, so we should too"

Netflix and LinkedIn are in the platform business. Is that your business too?

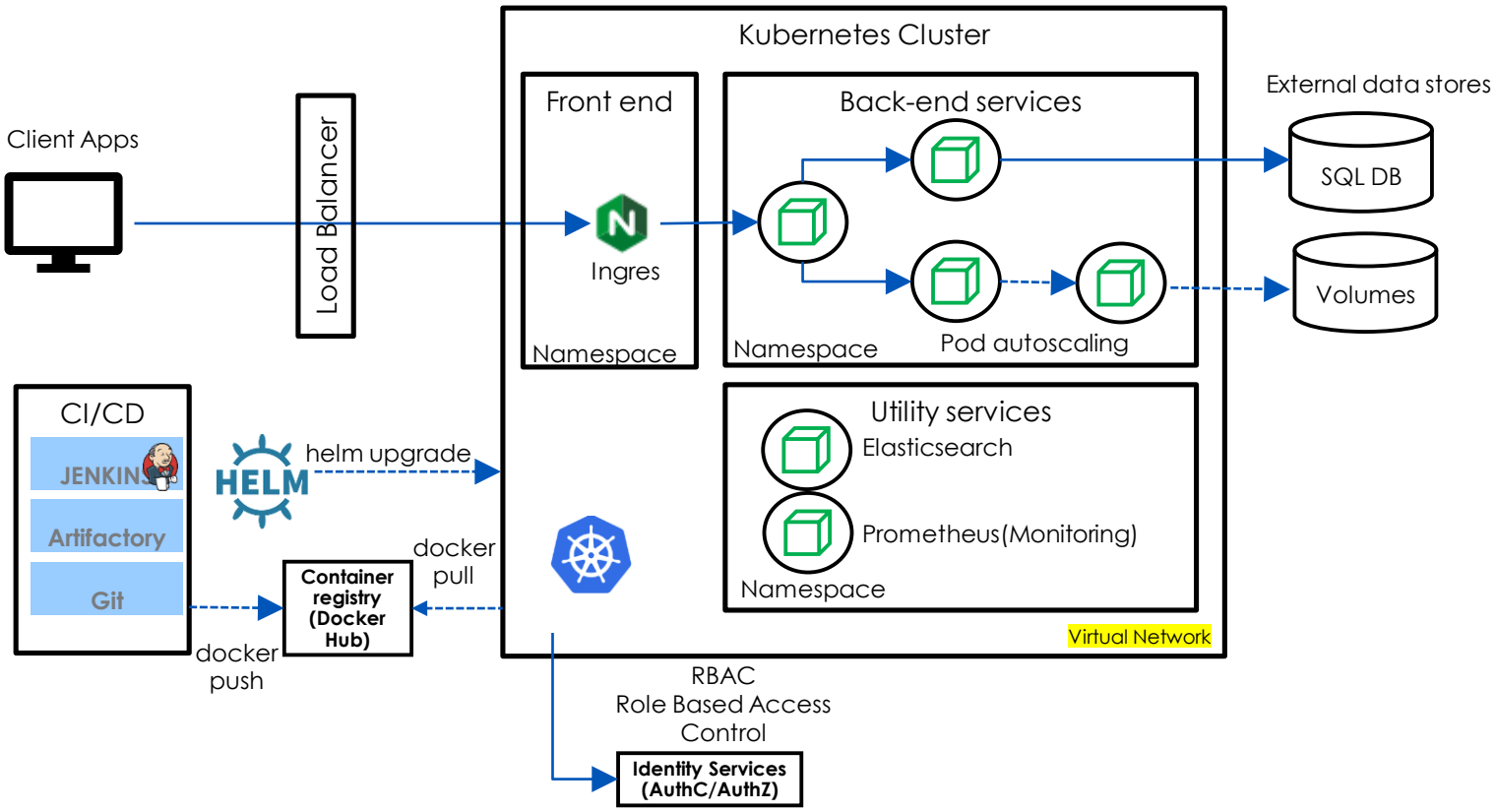
"We must choose microservices, or SOA"

Use both

API {First} Design Pattern – Microservices



MICROSERVICES - REFERENCE ARCHITECTURE WITH KUBERNETES



MICROSERVICES - REFERENCE ARCHITECTURE WITH KUBERNETES

Kubernetes cluster - Responsible for deploying the Kubernetes cluster and for managing the Kubernetes masters. You only manage the agent nodes.

Kubernetes Virtual network - Create the virtual network, which lets you control things like how the subnets are configured, on-premises connectivity, and IP addressing.

Kubernetes Ingress - An ingress exposes HTTP(S) routes to services inside the cluster. Its a collection of routing rules that govern how external users access services running in a Kubernetes cluster

External data stores - Microservices are typically stateless and write state to external data stores

Identity(AuthC/AuthZ) - To create and manage user authentication in client applications.

Container Registry(Docker Hub) - To store private Docker images, which are deployed to the cluster.

CI/CD Pipelines - DevOps Services and runs automated builds, tests, and deployments. CI/CD solutions such as Jenkins, Artifactory, Git

Helm - Helm is as a package manager for Kubernetes — a way to bundle Kubernetes objects into a single unit that you can publish, deploy, version, and update.

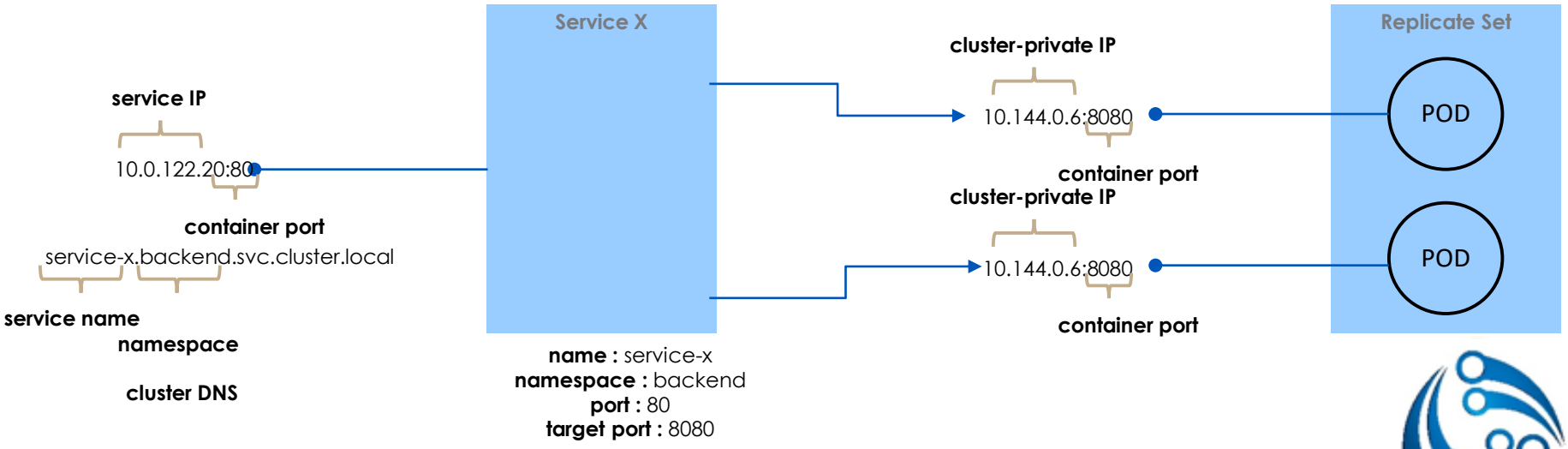
KUBERNETES MICROSERVICES DESIGN CONSIDERATIONS

The Kubernetes Service object is a natural way to model microservices in Kubernetes.
Microservices typically communicate through well-defined APIs, and are discoverable through some form of service discovery.
The Kubernetes Service object provides a set of capabilities that match these requirements:

- IP address. The Service object provides a static internal IP address for a group of pods (ReplicaSet). As pods are created or moved around, the service is always reachable at this internal IP address.
- Load balancing. Traffic sent to the service's IP address is load balanced to the pods.
- Service discovery. Services are assigned internal DNS entries by the Kubernetes DNS service. That means the API gateway can call a backend service using the DNS name. The same mechanism can be used for service-to-service communication. The DNS entries are organized by namespace, so if your namespaces correspond to bounded contexts, then the DNS name for a service will map naturally to the application domain.

The actual mapping to endpoint IP addresses and ports is done by kube-proxy, the Kubernetes network proxy.

Conceptual relation between services and pods.



KUBERNETES MICROSERVICES AND API GATEWAY

API gateway sits between external clients and the microservices.(general microservices design pattern).
It acts as a reverse proxy, routing requests from clients to microservices.
It may also perform various cross-cutting tasks such as authentication, SSL termination, and rate limiting.

Functionality provided by API gateway:

- **Gateway Routing:** Routing client requests to the right backend services. This provides a single endpoint for clients, and helps to decouple clients from services.
- **Gateway Aggregation:** Aggregation of multiple requests into a single request, to reduce chattiness between the client and the backend.
- **Gateway Offloading:** A gateway can offload functionality from the backend services, such as SSL termination, authentication, IP whitelisting, or client rate limiting (throttling).

Example : Most common implementation is to deploy an edge router or reverse proxy, such as Nginx, HAProxy, or raefik, inside the cluster.

Kubernetes Ingress resource type abstracts the configuration settings for a proxy server. It works in conjunction with an ingress controller, which provides the underlying implementation of the Ingress. There are ingress controllers for Nginx, HAProxy, Traefik, and Application Gateway.

The ingress controller handles configuring the proxy server. Often these require complex configuration files, which can be hard to tune if you aren't an expert, so the ingress controller is a nice abstraction. In addition, the Ingress Controller has access to the Kubernetes API, so it can make intelligent decisions about routing and load balancing. Example, the Nginx ingress controller bypasses the kube-proxy network proxy.

For complete control over the settings, bypass the abstraction and configure the proxy server manually.

Note : A reverse proxy server is a potential bottleneck or single point of failure, so always deploy at least two replicas for high availability.

KUBERNETES MICROSERVICES AND DATA STORAGE

- In a microservices architecture, services should not share data storage. Each service should own its own private data in a separate logical storage, to avoid hidden dependencies among services. The reason Is to avoid unintentional coupling between services, which can happen when services share the same underlying data schemas.

- When services manage their own data stores, they can use the right data store for their particular requirements.

- Avoid storing persistent data in local cluster storage, because that ties the data to the node.

- Instead, use an external service such as SQL Database or mount a persistent volume using storage disks or storage files.

- Use storage files if the same volume needs to be shared by multiple pods.

KUBERNETES MICROSERVICES AND NAMESPACES

Use namespaces to organize services within the cluster.
Every object in a Kubernetes cluster belongs to a namespace.
By default, when you create a new object, it goes into the default namespace.
Good practice to create namespaces that are more descriptive to help organize the resources in the cluster.

Advantages :
Namespaces help prevent naming collisions. When multiple teams deploy microservices into the same cluster, with possibly hundreds of microservices, it gets hard to manage if they all go into the same namespace.

Constraints:
Apply resource constraints to a namespace, so that the total set of pods assigned to that namespace cannot exceed the resource quota of the namespace.
Apply policies at the namespace level, including RBAC and security policies.
For a microservices architecture, considering organizing the microservices into bounded contexts, and creating namespaces for each bounded context.

Example : All microservices related to the "Order Fulfillment" bounded context could go into the same namespace. Alternatively, create a namespace for each development team.

Place utility services into their own separate namespace.
For example, you might deploy Elasticsearch or Prometheus for cluster monitoring, or Tiller for Helm.

KUBERNETES MICROSERVICES AND SCALABILITY

Kubernetes supports scale-out at two levels:

- Scale the number of pods allocated to a deployment.
- Scale the nodes in the cluster, to increase the total compute resources available to the cluster.

Use autoscaling to scale out pods and nodes, to minimize the chance that services will become resource starved under high load.
An autoscaling strategy must take both pods and nodes into account. If you just scale out the pods, eventually you will reach the resource limits of the nodes.

Pod autoscaling:
Horizontal Pod Autoscaler (HPA) scales pods based on observed CPU, memory, or custom metrics.
To configure horizontal pod scaling, you specify a target metric (for example, 70% of CPU), and the minimum and maximum number of replicas. Load test services to derive these numbers.
A side-effect of autoscaling is that pods may be created or evicted more frequently, as scale-out and scale-in events happen. Mitigate the effects of this use readiness probes to let Kubernetes know when a new pod is ready to accept traffic. Use pod disruption budgets to limit how many pods can be evicted from a service at a time.

Cluster autoscaling
The cluster autoscaler scales the number of nodes. If pods can't be scheduled because of resource constraints, the cluster autoscaler will provision more nodes. HPA looks at actual resources consumed or other metrics from running pods, the cluster autoscaler is provisioning nodes for pods that aren't scheduled yet. Therefore, it looks at the requested resources, as specified in the Kubernetes pod spec for a deployment. Use load testing to fine-tune these values. You can't change the VM size after you create the cluster, so you should do some initial capacity planning to choose an appropriate VM size for the agent nodes when you create the cluster.

KUBERNETES MICROSERVICES AND AVAILABILITY CONSIDERATIONS

Health probes - Kubernetes defines two types of health probe that a pod can expose:

Readiness probe: Tells Kubernetes whether the pod is ready to accept requests.
Liveness probe: Tells Kubernetes whether a pod should be removed and a new instance started.

A service has a label selector that matches a set of (zero or more) pods. Kubernetes load balances traffic to the pods that match the selector. Only pods that started successfully and are healthy receive traffic. If a container crashes, Kubernetes kills the pod and schedules a replacement.

A pod may not be ready to receive traffic, even though the pod started successfully. For example, there may be initialization tasks, where the application running in the container loads things into memory or reads configuration data. To indicate that a pod is healthy but not ready to receive traffic, define a readiness probe.

Liveness probes handle the case where a pod is still running, but is unhealthy and should be recycled. For example, suppose that a container is serving HTTP requests but hangs for some reason. The container doesn't crash, but it has stopped serving any requests. If you define an HTTP liveness probe, the probe will stop responding and that informs Kubernetes to restart the pod.

KUBERNETES MICROSERVICES AND AVAILABILITY CONSIDERATIONS

Considerations when designing probes:

- If code has a long startup time, liveness probe may report failure before the startup completes. Prevent this by using the **initialDelaySeconds** setting, which delays the probe from starting.

- A liveness probe helps when restarting the pod is likely to restore it to a healthy state. Use a liveness probe to mitigate failures against memory leaks or unexpected deadlocks,
  Note : Restarting a pod might end up in the pods immediately fail again(Deadlock).

- Use readiness probes to check dependent services.

Example, if a pod has a dependency on a database, the liveness probe might check the database connection.
Note :An external service might be temporarily unavailable for some reason. That will cause the readiness probe to fail for all the pods in your service, causing all of them to be removed from load balancing, and creating cascading failures upstream.
Better design pattern is to implement retry handling within service, so that your service can recover correctly from transient failures.

- **Resource constraints**

Resource contention can affect the availability of a service. Define resource constraints for containers, so that a single container cannot overwhelm the cluster resources (memory and CPU).
Note: For non-container resources, such as threads or network connections, use the Bulkhead Pattern to isolate resources.

Note : Use resource quotas to limit the total resources allowed for a namespace. That way, the front end can't starve the backend services for resources or vice-versa.

KUBERNETES MICROSERVICES AND SECURITY CONSIDERATIONS

- **Role** is a set of permissions that apply within a namespace. Permissions are defined as verbs (get, update, create, delete) on resources (pods, deployments, etc.).
- **RoleBinding** assigns users or groups to a Role.

- **Role based access control (RBAC) -** Kubernetes RBAC controls permissions to the Kubernetes API.

Example: Creating pods and listing pods are actions that can be authorized (or denied) to a user through RBAC. To assign Kubernetes permissions to users, you create roles and role bindings:

ClusterRole object, which is like a Role but applies to the entire cluster, across all namespaces. To assign users or groups to a ClusterRole, create a ClusterRoleBinding.

Kubernetes can integrate with AD for user authentication when you create an Kubernetes cluster,

Kubernetes cluster actually has two types of credentials for calling the Kubernetes API server: cluster user and cluster admin. The cluster admin credentials grant full access to the cluster. The cluster administrator can use this kubeconfig to create roles and role bindings.

Considerations

- Who can create or delete an Kubernetes cluster?
- Who can administer a cluster?
- Who can create or update resources within a namespace?
- It's a good practice to scope Kubernetes RBAC permissions by namespace, using Roles and RoleBindings, rather than ClusterRole and ClusterRoleBinding.

KUBERNETES MICROSERVICES AND SECRETS MANAGEMENT AND APPLICATION CREDENTIALS

Applications and services often need credentials that allow them to connect to external services like SQL Database. The challenge is to keep these credentials safe and not leak them.

Key Vault – In Kubernetes mount one or more secrets from Key Vault as a volume. The volume reads the secrets from Key Vault. The pod can then read the secrets just like a regular volume. See Kubernetes-KeyVault-FlexVolume project on GitHub.

HashiCorp Vault. Kubernetes applications can authenticate with HashiCorp Vault using AD. Deploy Vault itself to Kubernetes, but it's recommend to run it in a separate dedicated cluster from your application cluster.

Kubernetes secrets. Easiest to configure but has some challenges. Secrets are stored in etcd, which is a distributed key-value store. Kubenetes encrypts etcd at rest.

Pod and container security

Don't run containers in privileged mode. Privileged mode gives a container access to all devices on the host. You can set Pod Security Policy to disallow containers from running in privileged mode.

When possible, avoid running processes as root inside containers. Containers do not provide complete isolation from a security standpoint, so it's better to run a container process as a non-privileged user.

Store images in a trusted private registry(Docker Trusted Registry). Use a validating admission webhook in Kubernetes to ensure that pods can only pull images from the trusted registry.

Scan images for known vulnerabilities, using a scanning solution such as **Twistlock** and **Aqua.**

A container image is built up from layers. The base layers include the OS image and application framework images, such as ASP.NET Core or Node.js. The base images are typically created upstream from the application developers, and are maintained by other project maintainers. When these images are patched upstream, it's important to update, test, and redeploy your own images, so that you don't leave any known security vulnerabilities.

KUBERNETES MICROSERVICES – DEPLOYMENT (CI/CD) CONSIDERATIONS

Goals of a robust CI/CD process for a microservices architecture:

- Each team can build and deploy the services that it owns independently, without affecting or disrupting other teams.
- Before a new version of a service is deployed to production, it gets deployed to dev/test/QA environments for validation. Quality gates are enforced at each stage.
- A new version of a service can be deployed side-by-side with the previous version.
- Sufficient access control policies are in place.
- Trust the container images that are deployed to production.

Isolation of environments(Dev/Test/QA/Prod)
Kubernetes, you have a choice between physical isolation and logical isolation. Physical isolation means deploying to separate clusters. Logical isolation makes use of namespaces and policies.
Recommendation is to create a dedicated production cluster along with a separate cluster for your dev/test environments. Use logical isolation to separate environments within the dev/test cluster. Services deployed to the dev/test cluster should never have access to data stores that hold business data.

Helm
Consider using Helm to manage building and deploying services. Some of the features of Helm that help with CI/CD include:

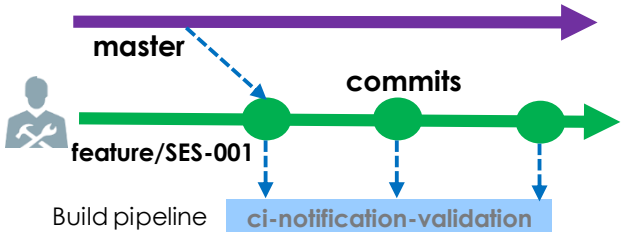
- Organizing all of the Kubernetes objects for a particular microservice into a single Helm chart.
- Deploying the chart as a single helm command, rather than a series of kubectl commands.
- Tracking updates and revisions, using semantic versioning, along with the ability to roll back to a previous version.
- The use of templates to avoid duplicating information, such as labels and selectors, across many files.
- Managing dependencies between charts.
- Publishing charts to a Helm repository, such as Azure Container Registry, and integrating them with the build pipeline.

KUBERNETES & MICROSERVICES – CICD WORKFLOW

Prerequisites

- The source control repository is monorepo, with folders organized by microservice.
- The team's branching strategy is based on trunk-based development.
- The team uses Jenkins Pipelines to run the CI/CD process.
- The team uses namespaces in container registry to isolate images that are approved for production from images that are still being tested.

Developer is working on a microservice called Notification Service. While developing a new feature, the developer checks code into a feature branch.



Pushing commits to this branch tiggers a CI build for the microservice. By convention, feature branches are named feature/*. The build definition file includes a trigger that filters by the branch name and the source path. Using this approach, each team can have its own build pipeline.

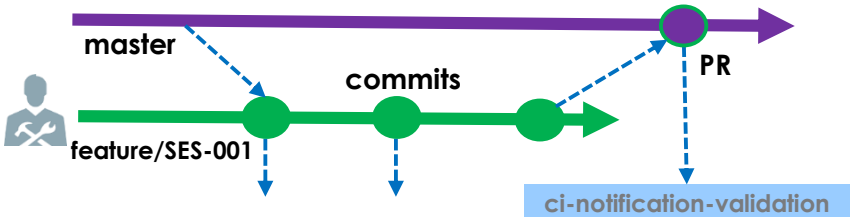
```
trigger:
  batch: true
  branches:
    include:
    - master
    - feature/*

    exclude:
    - feature/experimental/*

  paths:
     include:
     - /src/XXX/notification/
```

At this point in the workflow, the CI build runs some minimal code verification:

- Build code
- Run unit tests

KUBERNETES & MICROSERVICES – CICD WORKFLOW

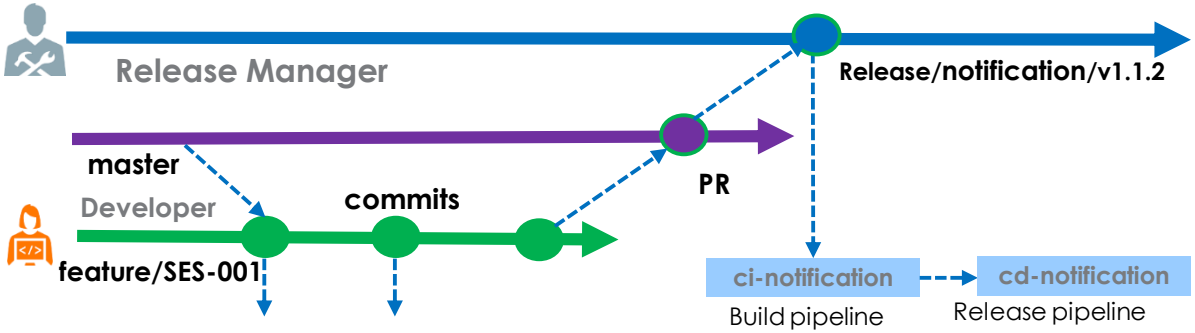
The idea here is to keep the build times short so the developer can get quick feedback.
When the feature is ready to merge into master, the developer opens a PR.
This triggers another CI build that performs some additional checks:
Build code
Run unit tests
Build the runtime container image
Run vulnerability scans on the image



Build pipeline

Note : Define policies to protect branches. For example, the policy could require a successful CI build plus a sign-off from an approver in order to merge into master.

KUBERNETES & MICROSERVICES – CICD WORKFLOW



Note : Manual approval steps can be completely automated if preferred

Team is ready to deploy a new version of the Notification service.
Release manager creates a branch from master with this naming pattern: release/<microservice name>/<semver>.
For example, release/notification/v1.1.2. This triggers a full CI build that runs all the previous steps plus:

- Push the Docker image to Container Registry. The image is tagged with the version number taken from the branch name.
- Run helm package to package the Helm chart
- Push the Helm package to Container Registry by running helm push.

Assuming this build succeeds, it triggers a deployment process using an Pipelines release pipeline.

- Run **helm upgrade** to deploy the Helm chart to a QA environment.
- An approver signs off before the package moves to production.
- Re-tag the Docker image for the production namespace in Container Registry.
- For example, if the current tag is **myrepo.cr.io/notification:v1.1.2**, the production tag is **myrepo.cr.io/prod/notification:v1.1.2**.
- Run **helm upgrade** to deploy the Helm chart to the production environment.