Kubernetes:

* Service discovery is done through DNS (Domain Name System) server
* A set of DNS record is created for each new service that’s created
* When looking up services in DNS server, run a DNS query for the service name and namespace
  + Each service is registered under the namespace
  + To access the service from the same namespace, look for <service\_name>
  + From another namespace, look for <service\_name>.<namespace>
  + The returned result for these look-ups is the cluster IP of the service
    - An alternative, though not recommended, is to access directly using cluster IP
    - Cluster IP is assigned randomly when service is created so it will be different every time. Would be better to access using DNS instead
* By default, the service name is metadata.name, but if a host name is specified, it takes precedence over the metadata name and becomes the service name

Istio Notes

* Service mesh: includes service discovery, load balancing, failure recovery , metrics and monitoring
* Istio provides control over the service mesh
* Deploy service with load balancing, service to service authentication, monitoring WITHOUT changing the service codes

Core Features

* Traffic management
  + Control the API calls between the services
  + Simplifies configuration for circuit breaker, timeouts and retries, making more reliable calls
* Security
  + Istio manages authentication, authorization, and encryption of service communications
  + Service communications are secured by default
* Observability
  + Robust tracing, monitoring, and logging

Pilot: Service Discovery using Kubernetes service registry, dynamic request routing, circuit breaker,

Citadel: authentication service, could upgrade unencrypted traffic and control service access, uses mTLS

Galley: API configuration for Istio

Envoy:

* A proxy sidecar that is injected into each pod as a separate container.
* Envoy intercepts all inbound and outbound api calls to and from the pod that it is injected in.
* Envoy then perform service discovery and update their load balancing pool accordingly.
* Failure recovery feature

Deploy:

* Do a build on all the projects that are being deployed
* Write a DockerFile or Docker-compose file for each project and in a cmd prompt window, navigate to the folder of the project and run docker build –t <image\_name:version> .
* Docker will then locate and run the DockerFile to make the image with the jar file that was built earlier.
* To deploy the image in Kubernetes as a deployment and service, there are two options:
  + First method is to run the command kubectl run <deployment\_name> --image=<image\_name:version> --port=<port> and this will create the deployment along with a pod. Then to expose the deployment as a service, run command kubectl expose deployment <deployment\_name> --type=”type”.
    - There are different type of services. The default type is ClusterIP, meaning it is only exposed internally and cannot be accessed from outside the VM. There is also “NodePort” type which is to expose externally and can be accessed from local browser. (Method for accessing will be described later). There is also “LoadBalancer” type with if the service has multiple deployments (ex. Multiple versions), the load balancer service will balance the incoming traffic and direct it to a certain version, depending on the rule specified for traffic routing.
  + The other method is to make a deployment.yaml file with the necessary information to make a deployment and a service with the imaged created earlier.
    - For making the deployment (kind = deployment) and the service (kind = service), specify name and label so they can be identified by other services with the label.
    - For the deployment section, the number of replicas can be set if one wishes to deploy multiple instances of the app. Also need to specify the name and version of the image that is to be used, along with the port for this container.
    - For the service section. Specify the deployment that is being exposed with spec.selector.app=<deployment label>. Also include the port and protocol for accessing the service.
* To use all the features in Istio, the Kubernetes cluster must have all the istio services implemented. To do so, download the istio folder and navigate to its directory. Set the Istioctl in the bin folder as path variable. Then install the istio crds (custom resource definition) with kubectl apply –f install/kubernetes/helm/istio/templates/crds.yml. Then to install the istio without mTLS authentication, run kubectl apply –f install/kubernetes/istio-demo.yml. Now, all the features should be installed in the istio-system namespace (can be accessed by adding the tag “–n istio-system” after commands)
* When using ingress gateway in deploying with Istio, the ingress host is (minikube ip) and port is found in (kubectl describe service istio-ingressgateway) under the port with name http2. The secured host is under the port with name https.
* Go to INGRESS\_HOST:INGRESS\_PORT/productpage to access the application. (/productpage is endpoints specified in by the ingress gateway yaml file)
* Istioctl kube-inject –f <input\_filename> -o <output\_filename> takes the input file and inject istio sidecar envoy into the service. Kubectl create –f <outpu\_filename> then takes the file and create a service with it.

Ingress Gateway:

* In a gateway.yaml file, add the necessary info for a gateway and virtual service.
* Gateway:
  + Use the istio default ingress controller (spec.selector.istio = ingressgateway).
  + Then set a port of 80 for http protocol. The reason for http protocol port to be 80 is because the default ingress gateway is configured to map port 31380 to port 80.
  + Later when accessing the gateway, go to browser and type https://(minikube ip):31380
* Virtual service:
  + Specify the gateway connected to the virtual service as the name of the gateway set up previously
  + Optionally add config for matching uri(can use exact or prefix)
  + Then add a route destination. With the host being the name of the service that this virtual service is calling, and port being the port of the same service
  + Refers to the destination rule to find the subset (version) of the destination host (deployment) that it is calling
* Destination Rules(Mixer feature):
  + Destination rules declare the subsets for the deployments, also referred to as the version for the deployments (pods).
  + It define the subset as the pods with the matching labels (names, version, etc.)
  + Destination acts as the reference for virtual services to determine the subset/version
* Run kubectl apply –f <gateway.yaml> and the gateway and virtual service will be configured.
* To see more about the gateway and virtual service that was just deployed, run kubectl get (gateways/virtualservices)
* Load balance is included in the istio-ingress gateway. The default load balancing method is round robin

Mixer:

* Flexible model for authorization policies and telemetry collections
* Envoy sidecar calls Mixer to perform precondition checks and to report telemetry
* Envoy sidecar has local cache so precondition checks can be done from cache
* Also uses buffer for outgoing telemetry so it calls Mixer less often
* Mixer separates backend systems (policies and telemetry) from the Istio system
* Mixer consists of many adapters to allow interaction with the backend infrastructure
* Mixer picks up attributes from the request, such as destination ip/service, source ip/service, path, size, time, etc.
* Attributes can be used in a expression for evaluation
* Uses configuration models to control the process authorization policy and telemetry collection

Denier(Mixer feature):

* A rule is added for with specs that searches for a specific call to services. Once a call that matches the specs is being executed, the denier will come in effect and interfere with the call
* The actions that can be taken by the denier includes returning a code and a message and it stops the call
* Instances: (still need to research)

Istio-Auth:

* Sends TLS certs to Envoy with each calls from API gateway to the services

Canary deployment:

* First deploy the version-1 of the application along with the gateway and virtual service for it.
* Also specify the destination rule to direct all traffic to verstion-1
* When version-2 is deployed and service is created, modify the virtual service to add a new route.destination
  + Now there are 2 destinations in the virtual service for the same host. To distinguish between the two, use a subset under each destination
  + Can even add restriction for each destination, such as http header match, so the newer version can only be accessed by certain users
  + Another method is to assign weights to each destination
* Also need to update the destination rule file to add the new subset

Destination Policy:

* Can be used to implement circuit breaker and pool ejection
* Specifies the destination and circuitBreaker method of that location in case it fails
  + Circuit breaker methods include consecutive errors (number of error before circuit breaker starts), sleep windows (shut down time), and more
* Citadel: authentication service, could upgrade unencrypted traffic and control service access, uses mTLS
* Galley: API configuration for Istio