Ingress Gateway:

* In a gateway.yaml file, add the necessary info for a gateway and virtual service.
* Gateway:
  + Sits at the edge of the mesh and managed incoming and outgoing calls
  + Exposes ports and implements load balancing
  + 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 name so only requests through the specified gateway are allowed
    - All other request will result in 404 response
  + 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 number) of the destination host (deployment label) 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
  + 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

Canary deployment(Traffic Shifting):

* 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/version, subsets are defined through destination rules
  + Can even add restriction for each destination, such as http header match, so the newer version can only be accessed by certain users
  + Another restriction is to assign weights to each destination
* Also need to update the destination rule file to add the new subset

Istio L7 routing feature:

* L7 routing feature is a combination of L7 routing and load balancing
* L7 routing is a traffic balancing feature that separated traffic basing of the type of application data
  + This allows application to be tuned to handle specific type of contents only, will help to increase efficiency
  + Allows different services to have different code in order to handle the different types of incoming data
  + Ex. One server serve only images, one serve only HTML/CSS/JS
* Traditional load balancer distribute the requests to servers based off algorithm
  + Requires all server to have the same content
* The combination of the 2 allows architect to design application that is highly optimized for specific type of content, while it is capable of distributing the request to even out the load

Fault Injection:

* Helps to test resiliency of the application
* In the virtual service config, can include a fault section under the http match section
  + Fault can be delay, abort, etc.
* There can be hard-coded timeout in the microservices level and the kubernetes service level

Circuit Breaker:

* Limits the impact of failures, latency spikes, and other undesirable effects
* Circuit breaking policies are set in the destination rule settings for the specific service
* The policies are checks to see if the current state of the service is “broken”
  + In the event that it is broken, the actions to take is also specified in the policy
  + The action could include pool ejection, meaning the service is removed from the pool and will not be accessed until it is back up and running
  + The duration of which the service is down is also specified in the policy along with policies that check for error, such as maxConnection, MaxPendingRequest, and Consecutive error, MaxRequestPerConnection
  + If any of these policies are triggered, the circuit is “open ” and any traffic that needs to access the open part of the circuit will fail, and trigger the back-up function in the app

Mirroring/Shadowing:

* Traffic mirroring is a concept where a copy or a portion of the live traffic is routed to another deployment in the service
* Very useful for code releases, allows testing for error, performance and results of the newly deployed code
* Mirroring feature is set in the virtual service, under route and same level as destination
* Mirrored requests are “fire and forget”, meaning all responses are discarded
* For example:
  + Assuming v1 is running and v2 is mirroring v1 and client is the service sending the calls
  + Use kubectl exec –it <client\_pod> -c <container\_name> -- sh –c ‘curl http://service:port’ to send calls for testing purpose
  + Use kubectl logs –f <v1\_pod> -c <container\_name> to check the logs for v1, should see the call made previously
    - For Istio, each pod usually has 2 containers, the actual one and the Envoy proxy
  + Also check the log for v2, should also see the call made previously
    - Even though all traffic is routed to v1