**CI & CD**

CI stands for continuous integration, CD stands for both continuous delivery and continuous deployment.

CI is a process where team members integrate their work continuously in a shared repo in which automated tests and automated builds also occur to validate their work.

CD (continuous delivery) basically ensures that the build of our app is in a deployable state. So after CI process, we need to make some checkings and this is the job of CD (continuous delivery) so that there is no error with incompatibilities. Continuous delivery may involve a manual step meaning that you could need to approve the change even though the checking was successful.

CD (continuous deployment) basically deploys our app automagically after all the tests.

Gitlab, Jenkins can be used to automate devops operations.

Gitlab pipelines have two main components:

* Job – describes the task needed to be done
* Stage – defines the order of the pipeline. It defines the order in which jobs will be completed.

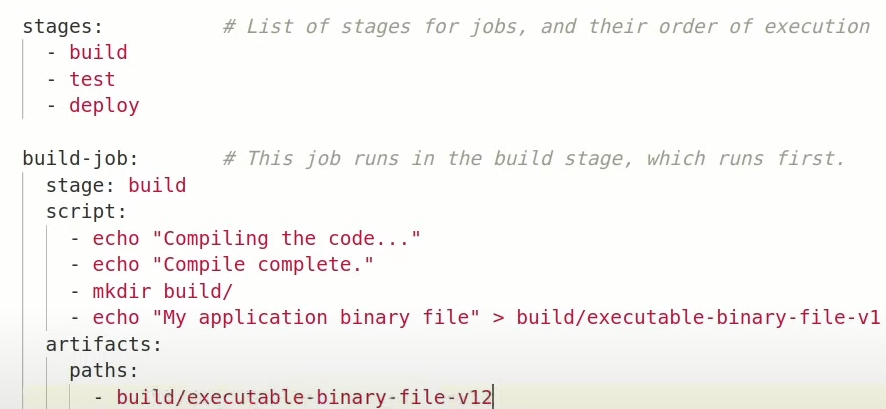
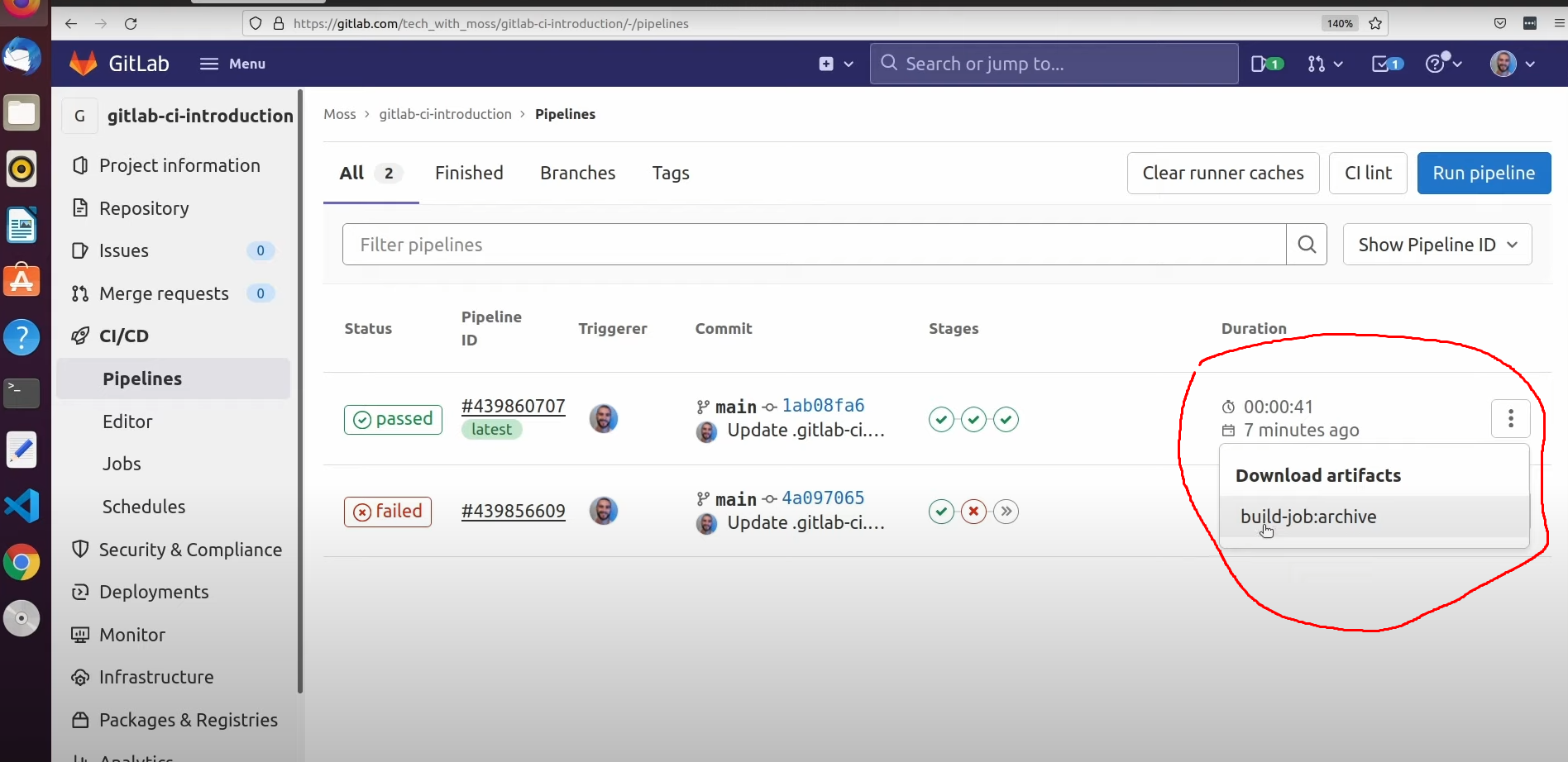
A pipeline is a set of instructions for a program to execute. A program that executes jobs in a gitlab pipeline is called a runner.

In GitLab CI, Runners run your yaml. A runner is an isolated (virtual) machine that picks up builds through the coordinator API of GitLab CI. A runner can be specific to a certain project or serve any project in GitLab CI. A runner that serves all projects is called a shared runner.

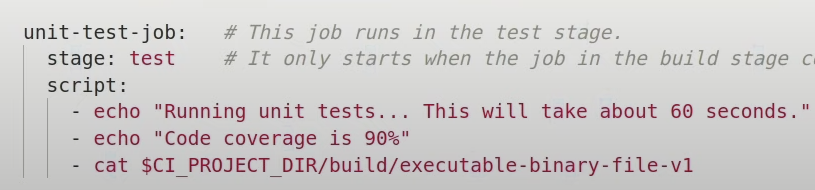
If we don’t have any stages then jobs will be performed in parallel.

One important difference of Gitlab from Jenkins is that jobs run independently of each other and have a fresh environment in each job. Artifacts passed between jobs is controlled using the artifacts and dependencies keywords.

“artifacts” keyword allows us to specify files that should be shared with downstream jobs in the pipeline. This keyword also allows us to download those specified files to our local machine after the pipeline has completed. For example, if a job generates a test report, we may want to download that and review it after the pipeline is completed



Now after completing **the script** section the gitlab runner that is executing this “build-job” job will upload the executable-binary-file-v1 to gitlab and then any downstream jobs like the below one “unit-test-job” job will first download any available artifacts from gitlab before executing its script 🡺

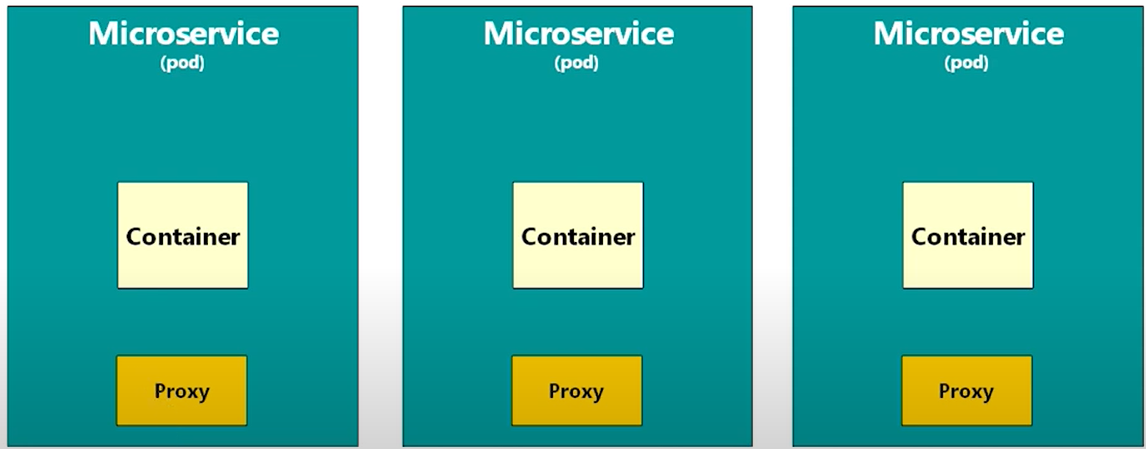


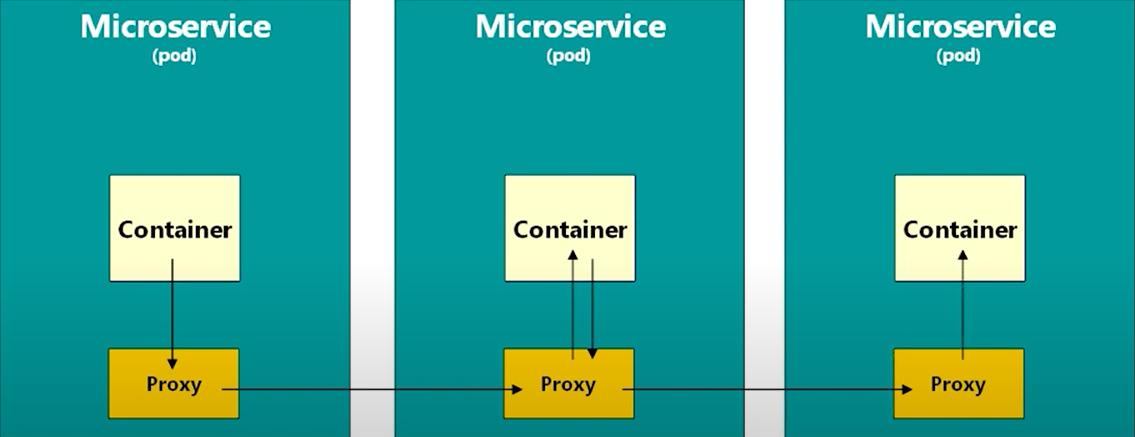
**Service mesh (Istio)**

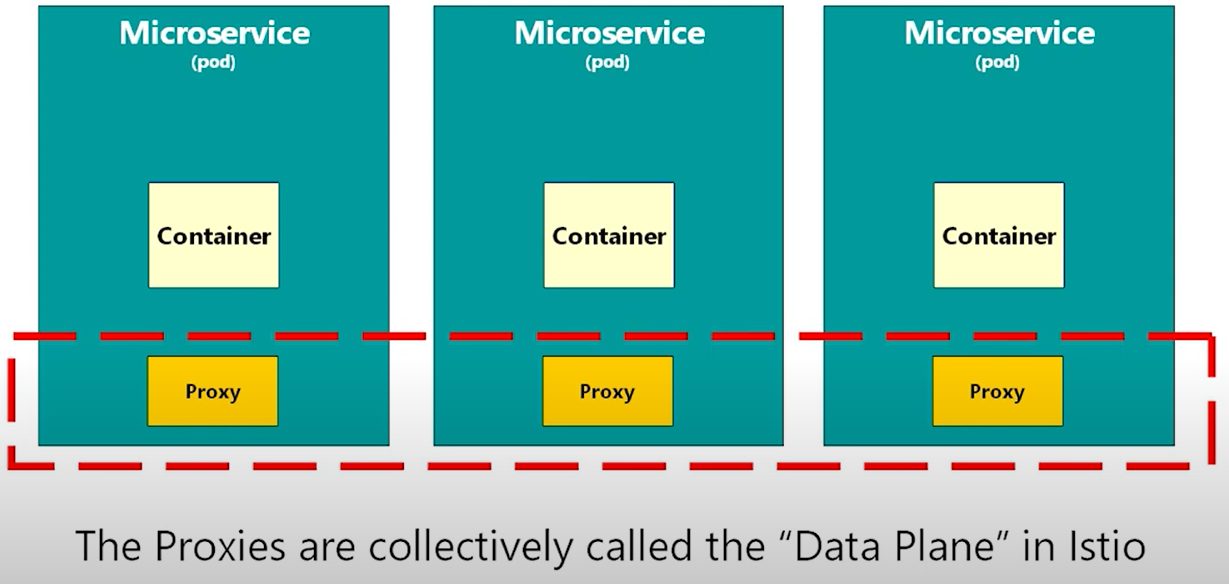
Istio is one of the service meshes. Service mesh is an extra layer of software, networking with each other, you deploy alongside your cluster (eg. K8s). This extra layer of software sits underneath of our componenets (pods) and analyzes all the traffic. With service mesh, we can analyze what is happenning and how our components (microservices) are connected with each other. With Kiali we can vizualize all of these (network connections, responses, etc).

Without a service mesh we would have to look at the logs of our pods which is rather a tedious and difficult task. But with service mesh we know exactly where the error is happening.

Istio injects its own container which is called a proxy. Istio is created in isto-system namespace so if we want to get pods of istio then the command: kubectl get pods –n istio-system. The proxies are called data plane.







And istio-telemetry (that analyzes every request sent to a pod), istio-pilot, istio-tracking are together called control plane.



An Istio service mesh is logically split into a **data plane** and a **control plane**.

* The **data plane** is composed of a set of intelligent proxies ([Envoy](https://www.envoyproxy.io/)) deployed as sidecars. These proxies mediate and control all network communication between microservices. They also collect and report telemetry on all mesh traffic.
* The **control plane** manages and configures the proxies to route traffic.

The proxies are actually envoy proxies and istio abtracts these envoy proxies so that we don’t deal with them.

Istiod is a pod that makes up our control plane of the istio service mesh. Istiod is responsible for injecting sidecar proxies into our services. These proxies then form the data plane.

Istiod has the following components: Pilot, Citadel, Galley.

The **pilot** component is responsible for traffic management, and injecting and managing the lifecycle of the proxies.

The **citadel** component is basically the certificate or authority which helps achieve mutual TLS between services that are part of the mesh.

The **galley** component in istiod is a translator. It translates kubernetes yaml into a format that istio can understand.

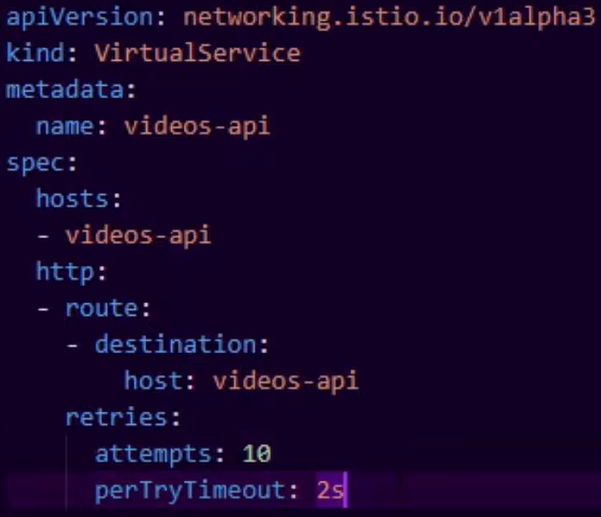
We can set the istio-injection-enabled label on a namespace to have the istio side car automatically injected into any pod that gets created in the labelled namespace. Running pods will have to be recreated for injection to occur. The command🡺

kubectl label napespaceName istio-injection-enabled=enabled

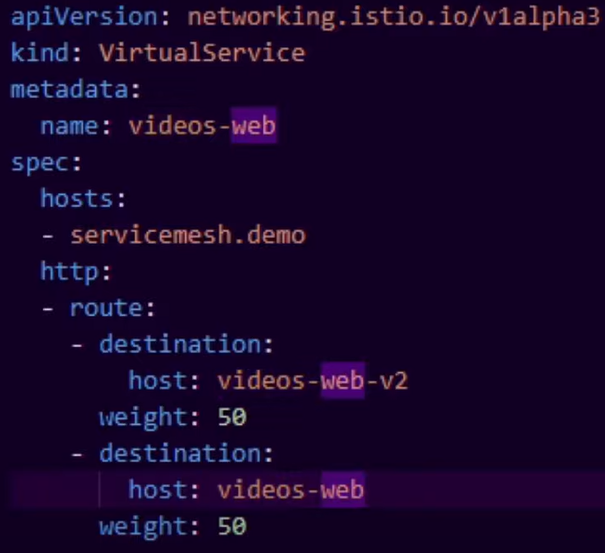
kubectl label default istio-injection-enabled=enabled

Now our entire default namespace is meshed.

Vitrual service is an istio configuration that allows us to affect traffic routing. We use a virtual service to define what we want istio to do with the traffic, for example, automatic retry, timeout settings and more. For instance, how many retries of the traffic can be applied to a specific service or where to route the traffic.



We can also set the virtual service such that 50% of the traffic goes to one service and 50% goes to another one.



Here the weight denotes the percentage of the traffic. So 50% goes to video-web-v2 and 50% goes to videos-web.