**Git Lab System Design**

<https://www.youtube.com/watch?v=uCU8jdYzpac>

GitLab.com Infrastructure Overview and HA Walkthrough - June 18, 2018

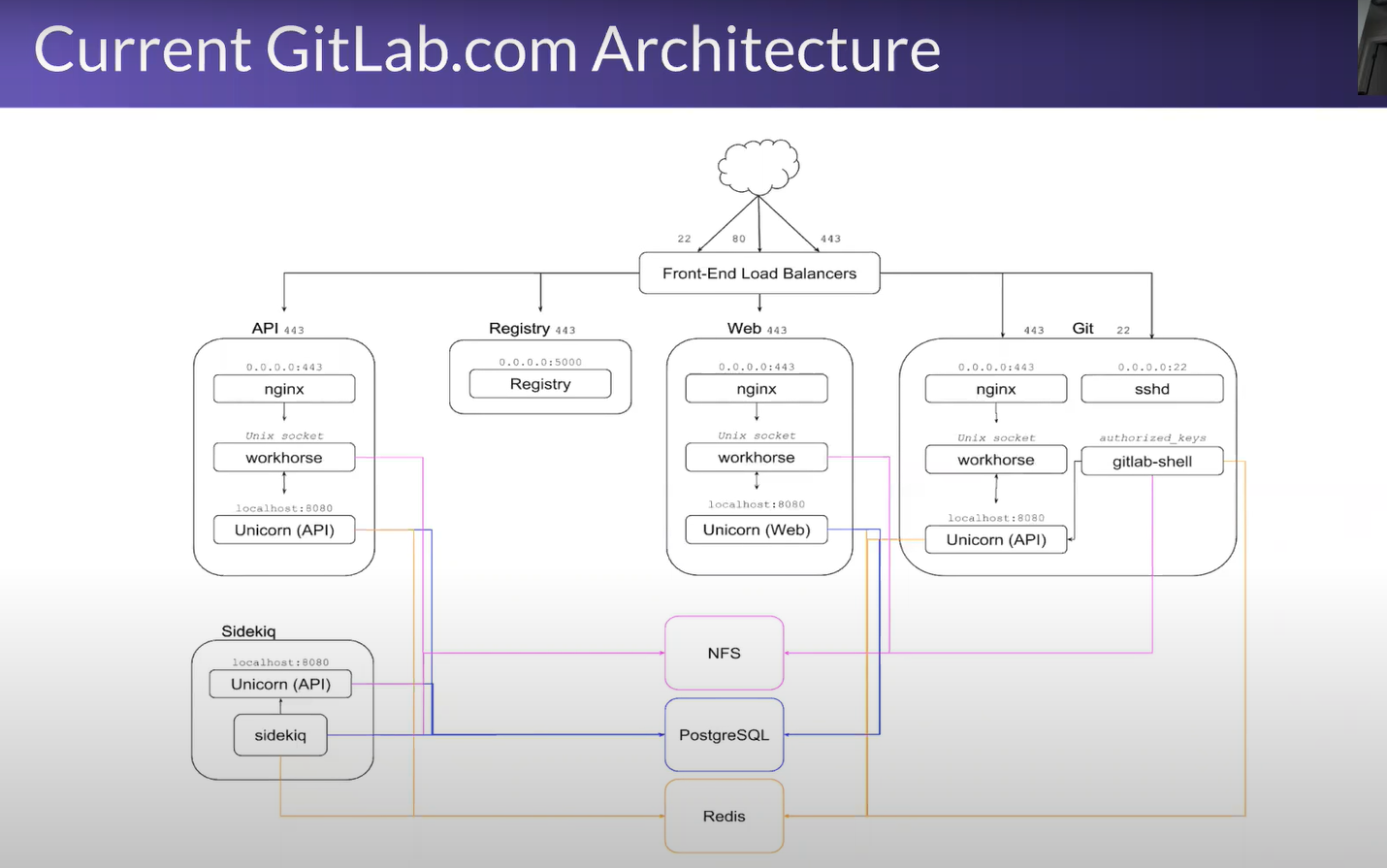
**GitLab Architecture:**

<https://about.gitlab.com/handbook/engineering/infrastructure/production/architecture/>

**GitLab GEO Tutorial:** By GitLab

<https://www.youtube.com/watch?v=h3Dwhv8JgPU>





Load balancer balances the load across the network.

**API’s:** We’ve series of hosts to cover API’s.

**Registry:** Registry are there for docker images.

**Web Servers:** For UI front-end.

**GIT:** Is there for GIT interactions, which may be based upon HTTPS OR SSH.

**NFS:** Is a centralized Storage System. NFS [Network File System] is also created in Sharded manner so any repo created, they get created in multiple NFS in round robin fashion. There are NFS sharding has been done and all of them get track it in Postgres. Any NFS which get around 60% space fill, we create a new NFS node and leave old NFS node for existing data growth and scale. We also take snapshot of all NFS shards to prevent single point of failure

**Sidekiq:** Is a worker process, that takes things out of queue, process them and bring them back in. So all Git operations, all notes, merge requests and CI Jobs, Sidekiq spun and run.

**Scaling Sidekiq:** The ability to take Sidekiq and split it out is one of the thing which people don’t really know they can do but it adds immense flexibility and scale for functionality. This is really handy if you have so.

Splitting Sidekiq enables you to have more processing and power to process more tasks. Splitting Sidekiq has been enabled in GitLab version 9.

In Sidekiq, basically there would be a Sidekiq cluster, which would have queue groups. So say for ex. you have multiple merge requests which takes a lot of time to process, we can spin up a Sidekiq node and enable Sidekiq cluster over there. Now this new Sidekiq basically can pull the things from Redis cluster and start processing them. We’ve different types of Queue in GitLab like **ASAP**, in such queue whatever the items are there it says it need to be process ASAP once we have a node is available. There is also a queue tagged as **best effort queue**, so it says whenever it can happen, just happen.

**Note:** There are multiple Sidekiq nodes are there which basically pulls up the data from queue and process them. And whenever a node picks the job id from queue to process it further, it acquire a lock on it and has a timeout as well so if there is nothing confirmation about job processing is returned then acquired lock would be removed and job will still persists in queue.

**Redis:** This is the work queue and Redis is where all of the real-time transaction queue stuff comes from. So anytime you click an object to be modifying your time that you submit, new text for an issue OR a note on something that get submitted into Redis through Sidekiq which is backend for Redis.

We divided Redis queue into 2 parts:

1. Cash queue: In this part of cache, the data we don’t write back to Disks so if we loose some data from this queue, it can be further cached from disk.
2. Faithful queue:

**Stateful System:**

* Data Storage/ System of record
* PostgreSQL
* PGBouncer
* Work Queue
  + Redis

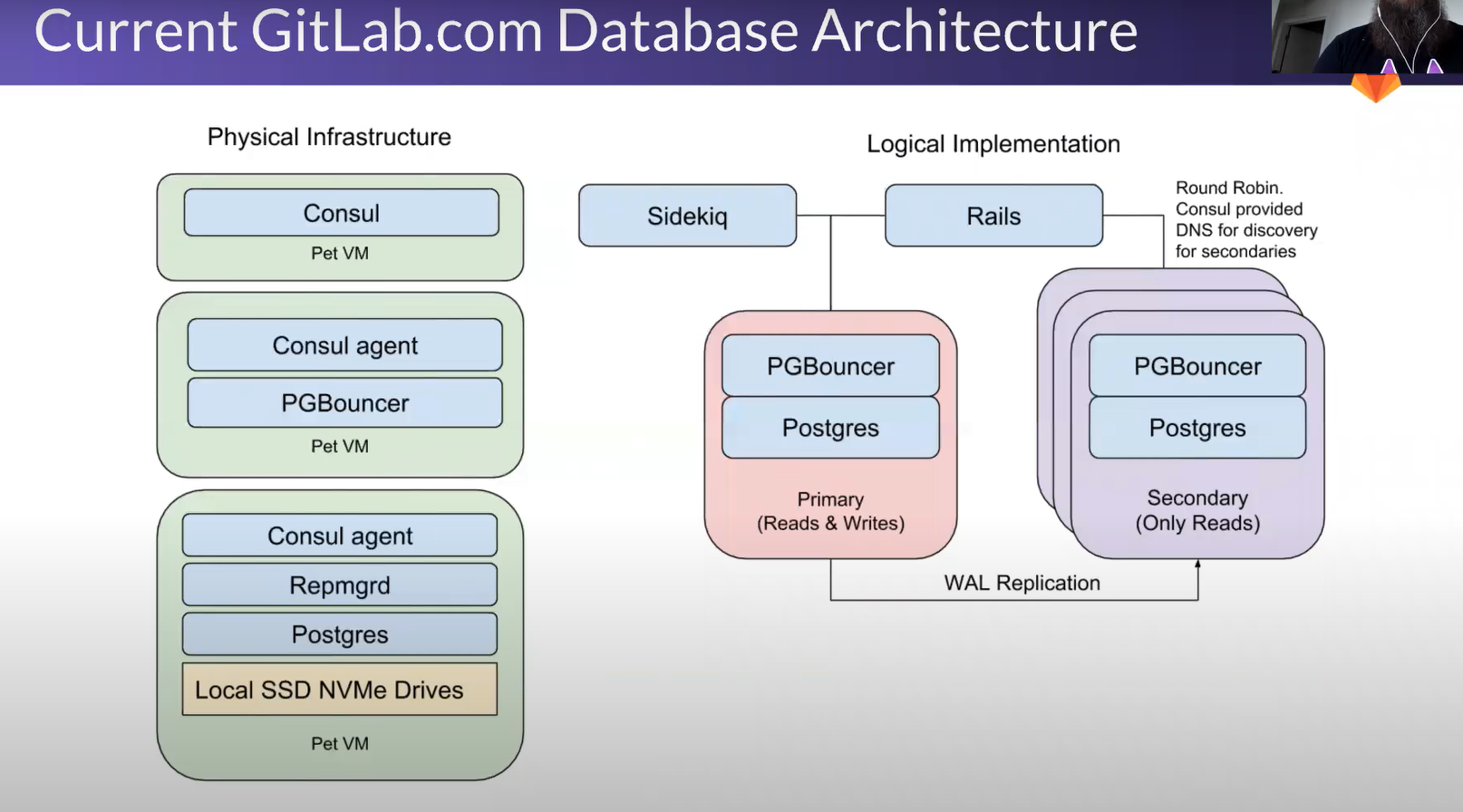
**Stateless Systems:**

* UI Display are stateless system like Web nodes, they don’t have any state to them.
* Processing like Sidekiq Nodes. If you’re running on single node, you can restart your node which would be non-destructive thing that you can do to stop and start your queue processing.

Sidekiq reads jobs from a Redis queue, using the First In First Out (FIFO) model, to process jobs. Job processing is asynchronous.

**GitLab Database Architecture:**

PGBouncer, Repmgr and Console



Above is a GitLab Database Architecture, where Postgres have been used in HA fashion.

There we’ve Postgres with PGBouncer Infront of it for connection pooling. Here Application talks with PGBouncer to access the database.

There **Repmgr** is used to add Postgres nodes on dynamic basis. When we’ve more than one Postgres node, then we make one node as a primary where all updates on issues and comments would happen. The other nodes, used as follower nodes which would follow the primary node, and would serve basically read related workload.

**Console:** Is a service discovery tool, used to know and track when database node comes online and when database node goes offline. It can also scale and use for Web nodes, Redis etc.

Console is installed with PG database. It keeps track of Postgres nodes health and if any of the node become unhealthy then it takes out that node.

The RDBMS [Postgres] we’re choosing here because whatever the data were pushing-in and pull-from GitLab, it requires to be consistent.

**Postgres DB usage:** Here Postgres captures info like user id, comment on issue, what’s the latest hash ref is to be displayed of the Git repo that you’re looking at.

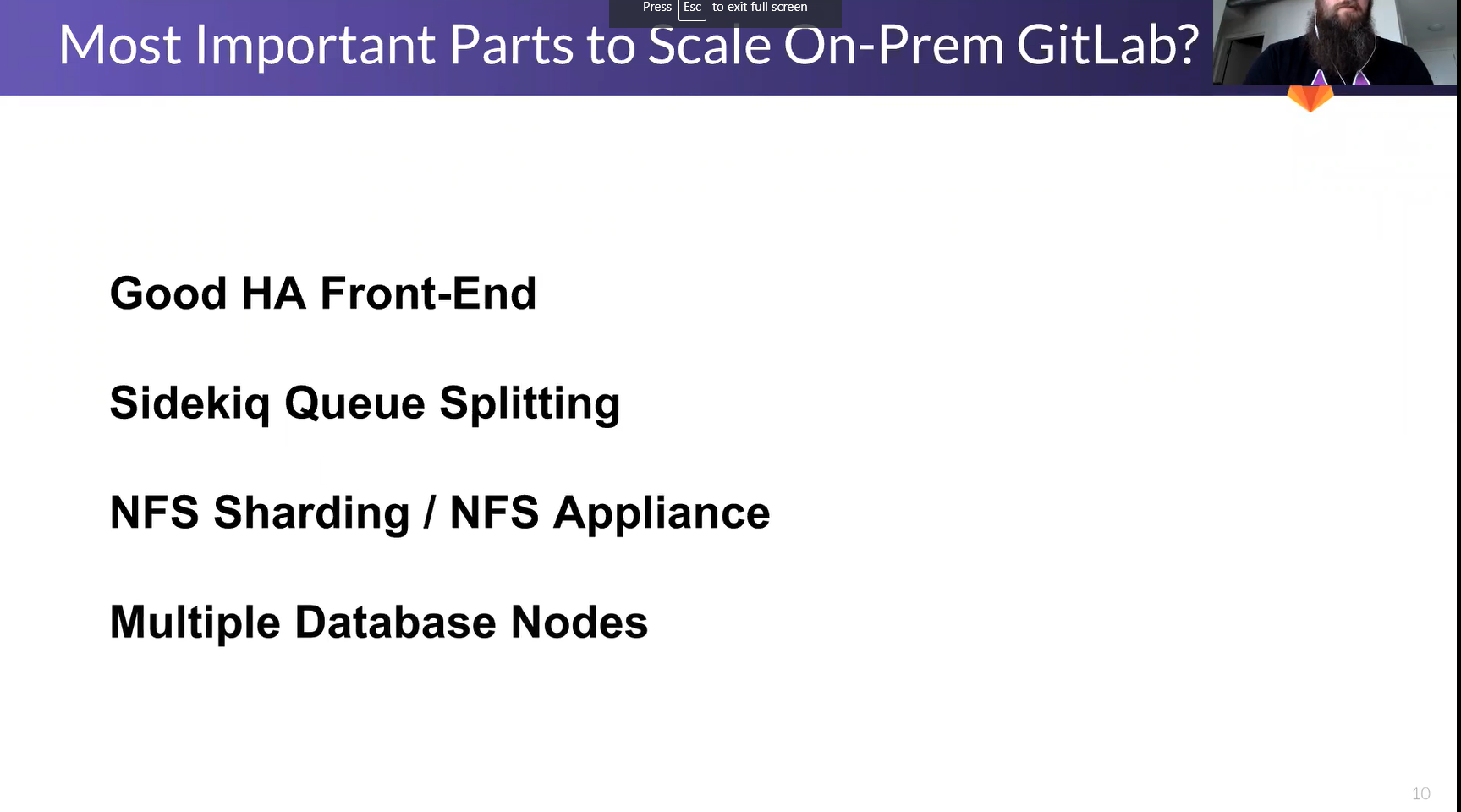
The data we’re pushing into Postgres DB are stateful.

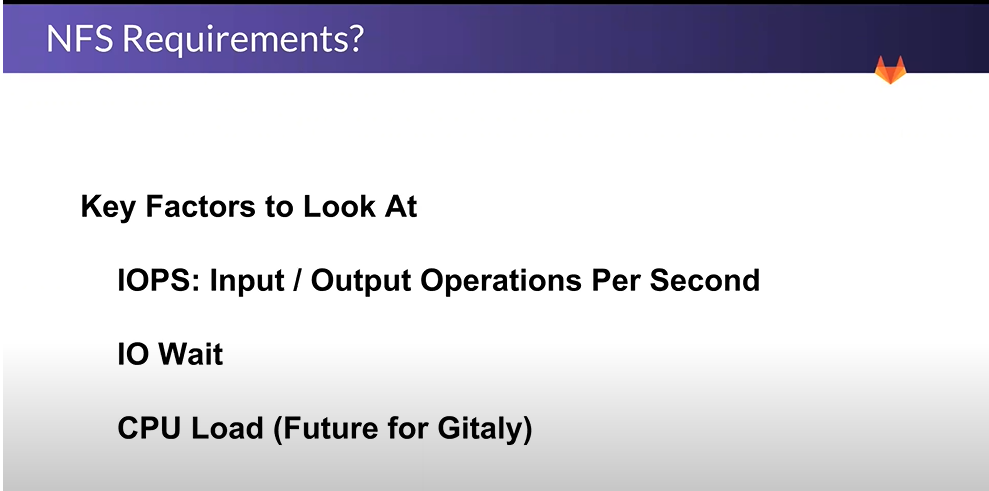
**DB point-in-time recovery:** So the data which Postgres is writing to wall segments, there is an open-source package called as **wally** which basically stream that data and directly push into S3 and during off business hour full backup are taken so this way we can do point in time restore.

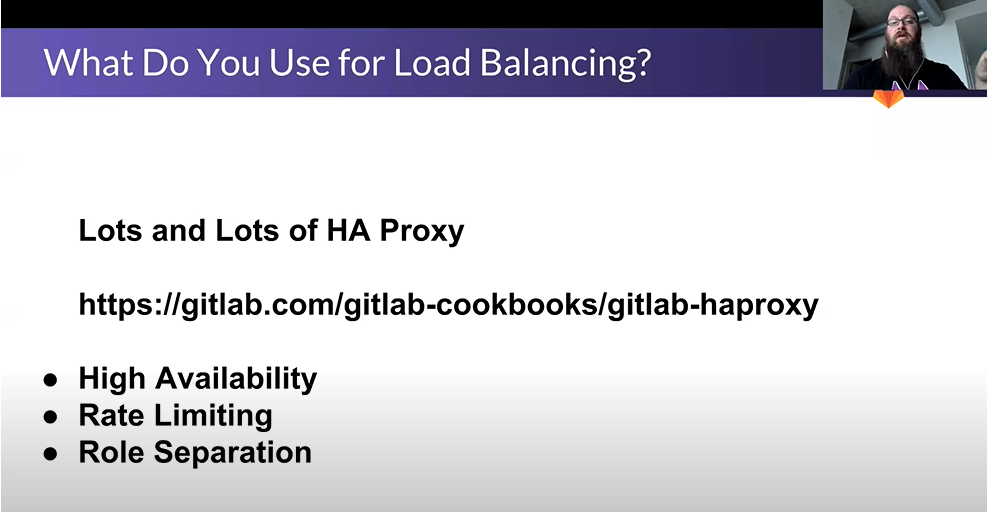
**High Availability:** You can set the HA for below components:

1. Database
2. Redis
3. Web front end
4. Sidekiq

**Scaling:**







**Load Balancer:**

HA proxy can used as Load balancer to perform above 3 tasks:

1. High Availability
2. Rate Limiting
3. Role Separation

HA proxy looks at the URL that you try to access and we split that based upon some base URL path, if the URL deal with series of git transactions those are handed off to a backend that has just **GIT** nodes that all they do is process git transactions whether they are SSH OR HTTPS.

And this is how WEB UI functionality also get splits.

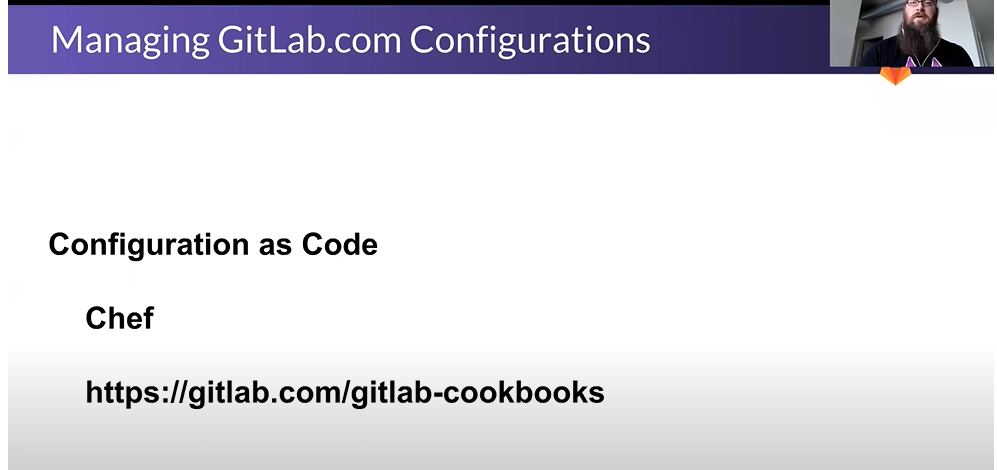
If you’re accessing a URL which is API centric, that goes to different cluster of backend which handles just API traffic. This API nodes are solely for CI traffic

And top of on that we apply Rate Limiting which basically controls how many sessions you can have per second. This basically prevent to make over sessions and creating any overload. So it keep on looking how many requests are coming from a specific IP and controls it.

**Monitoring:**

Monitoring can be set up-on NFS [for its IOPS, and IO wait], SideKiq, REDIS, HA Proxy and Postgres.





You can log all logs at Sys log, stream them out to a Cluster and look at the logs there.

**Database Modelling:**

User\_table:

User\_id, user\_name, user-email

Local GIT Table:

Local\_file\_id, local\_file\_name, file\_lines\_changed\_info, source\_branch\_id, source\_branch\_name, local\_repo\_id, local\_repo\_name, status, date and time, commit\_id, commit\_msg, user\_id[FK]

status: untracked, staged, commit, modified

Remote GIT Table:

remote\_file\_id, remote\_file\_name, merge\_id, merge\_msg, merge\_status, remote\_branch\_id, remote\_branch\_name, approver\_id, approver\_name, date and time , user\_id[FK], commit\_id[FK]

merge\_status: created, merged, merge\_conflict

Clone Table: protocol, clone\_id, source\_branch\_id, remote\_branch\_id [FK], remote\_branch\_name [FK]

Protocol: HTTPS OR SSH

**Git functions:**

Git add

Git commit -m

Git push

Git checkout

Git push -> This function will push the code into GitLab Server.

Git pull

Git reset -> to un-stage a change