Git

What is Git?

Git is a distributed version control system that helps manage and track changes in the source code efficiently.

What is GitHub?

GitHub is a web-based platform that uses Git to store and collaborate on code.

Version Control System

A Version Control System (VCS) is a software tool that helps manage and track changes to files and maintain a complete history of those changes.
Listed below are the functions of a VCS

- Allows developers to work simultaneously.
- Does not allow overwriting each other's changes.
- Maintains a history of every version.

Following are the types of VCS

- Centralized version control system (CVCS).
- Distributed/Decentralized version control system (DVCS).

What is VSC?

- VCS helps to track and manage changes to source code.
- Experimentation of code changes
- Reduce storage
- Maintain the different version of the code
- Encryption when uploading or downloading the files or data.

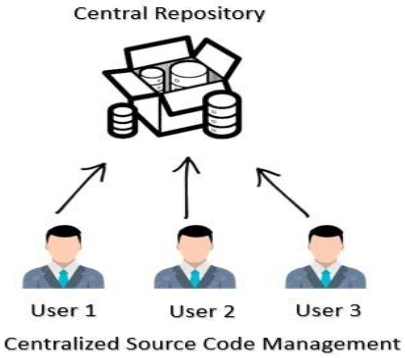
Type of VCS

1. Local Version control system
2. Centralized version control system
3. Distributed version control system

What is Centralised Version Control?

- It is used for managing the versions of the codes.
- Any developer can view/modify the code of another developer.
- If the central server fails, we can lose the data
- Continuous internet is required for connecting and storing the data (Slow Process)
- Code is not available locally

-> Operate on sync model, dependency on network.
-> Encryption is missing while pull and push the files to remote repository.
-> Local repository is not available.
-> If server goes down the repo will be gone. To overcome we need to take the back up.

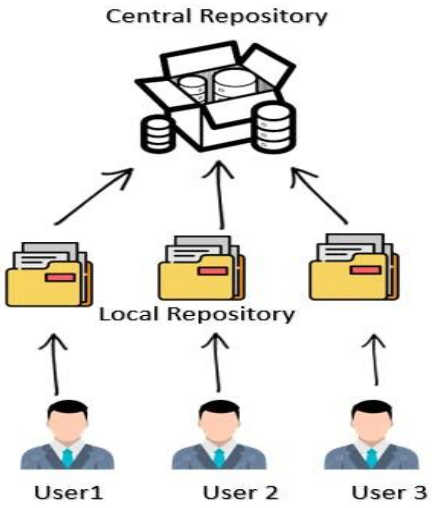


Distributed Version Control System:

- Git invented by Linus Torvalds in 2005
- GitHub is a web-based platform that uses Git to store and collaborate on code.
- You can access the code fast as the code is stored in a local repository as well
- If the remote repository goes down, every developer has their own code store at the local repository
- **Version control is done**

-> Encryption is available when data is traveling over the internet, pull and push. Data encrypted. SHA 256 (secure hash algorithm).

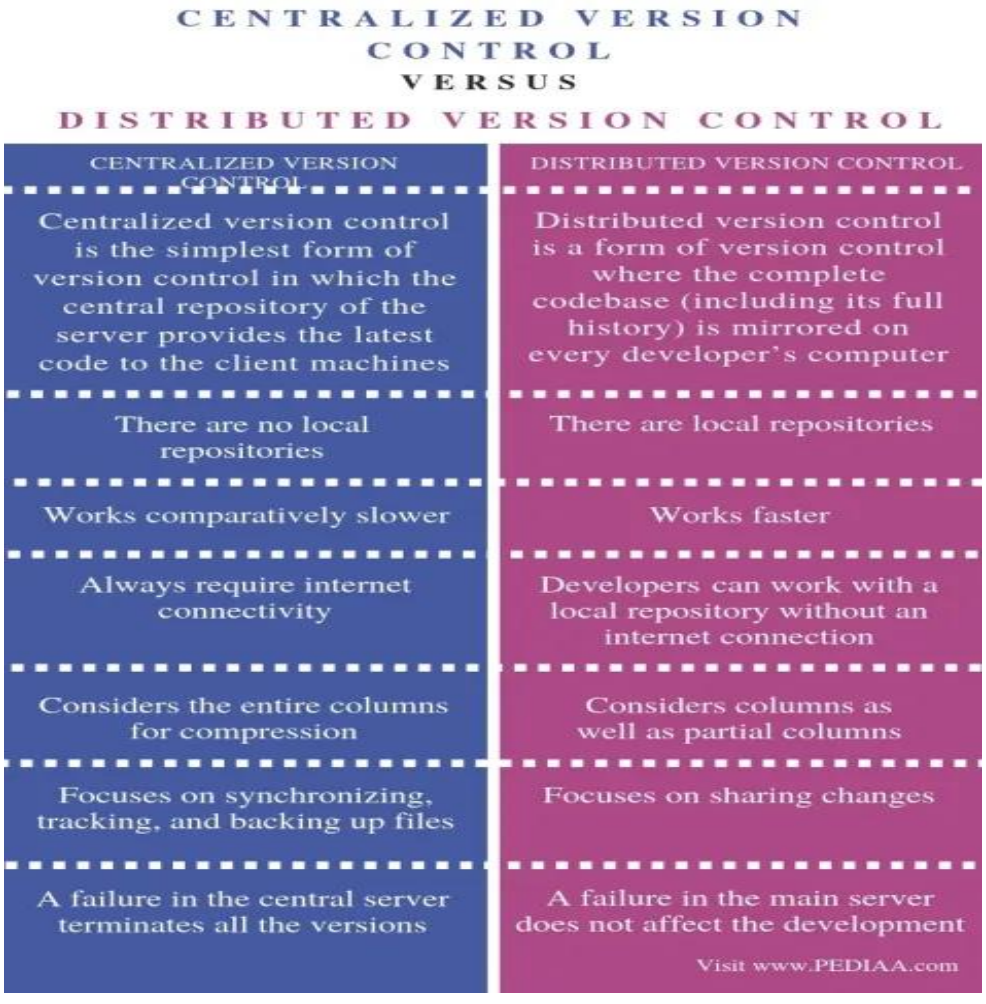
-> No dependency on the internet.



Difference between CVCS and DVCS?

Difference Between CVCS and DVCS

1. Every time user need to pull the data from the server and push back once done	1. It has the local repository and remote server repository so continuous connection is not required
2. Only online access available	2. Both online offline access available
3. Slower as command needs to communicate to the server	3. Fast as it deals with local repository
4. If Central repository is down , Deweloper can't work	4. No impact of central repository failure, as it use local for code



Types of VCS:

CVCS Tools

1. Team Foundation server (TFS) - ms office
2. Sub version - SVN

DVCS tools

1. Tortoise
2. Mercurial
3. Git

What is Git?

Git is a distributed version control system that helps manage and track changes in the source code efficiently.

https://learngitwithpra9.hashnode.dev/git-source-code-management-part-1

The Git config command

To register a username, run the below command:

```
$ git config --global user.name "Ram"
```

To register an email address for the given author, run the below command:

```
$ git config --global user.email "Jaisitarama@gmail.com"
```

To check configuration settings;
This command used to list all the settings that Git can find at that point.

```
$ git config --list
```

Git configuration levels

The git config command can accept arguments to specify the configuration level. The following configuration levels are available in the Git config.

- Local
- Global
- system

--local

It is the default level in **Git**. Git config will write to a local level if no configuration option is given. Local configuration values are stored in **.git/config** directory as a file.

Applies settings to the specific repository.

```
$ git config --local
```

--global

The global level configuration is **user-specific configuration.** User-specific means, it is applied to an individual operating system user. Global configuration values are stored in a user's home directory. **~ /.gitconfig** on UNIX systems and **C:\Users\\.gitconfig** on windows as a file format.

Applies settings to the user's environment. This is the most common level for personal settings.

```
$ git config --global
```

--system

The system-level configuration is applied across **an entire system**. **The entire system means all users on an operating system and all repositories.** The system-level configuration file stores in a gitconfig file off the system directory. $(prefix)/etc/gitconfig on UNIX systems and C:\ProgramData\Git\config on Windows.

The order of priority of the Git config is local, global, and system, respectively. It means when looking for a configuration value, Git will start at the local level and bubble up to the system level.

Applies settings to every user on the system.

```
$ git config --system
```

Git Terminology

Branch

In Git, a branch is a lightweight movable pointer to a specific commit. It allows developers to create a separate copy of the project (codebase) to work on new features or bug fixes without affecting the main project (codebase). It allows for parallel development and isolates changes from each other.

Checkout

Checkout is a command used to switch between different branches or to restore files in the working directory to a previous state.

- **Switching Branches**
  To switch to an existing branch named "feature-branch":
  ```
  $ git checkout feature-branch
  ```

- **Restoring a File to a Previous State**
  To restore a specific file, "example.txt," to the version in the last commit:
  ```
  $ git checkout -- file.txt
  ```

Cherry-Picking

The **`git cherry-pick`** command is used to apply a specific commit from one branch to another branch without merging the entire branch.

In case you made a mistake and committed a change into the wrong branch, but do not want to merge the whole branch. You can revert the commit and cherry-pick it on another branch.

```
$ git cherry-pick <commit-hash>
```

Clone

The **`git clone`** command is used to create a local copy of a remote Git repository. It downloads the entire repository, including all its files, commit history, and branches, to your local machine.

Fetch

`git fetch` only downloads the latest changes from the remote repository to local repository without merging the changes.
It downloads the latest commits, branches, and tags from a remote repository without automatically merging

HEAD

HEAD is the representation of the last commit in the current checkout branch.
"HEAD" refers to a special pointer that always points to the latest commit on the currently checked-out branch

Index

The Git index is a staging area between the working directory and repository.
The Git index is a temporary staging area that stores a snapshot of the working tree.

Master

Master is the default name for the main branch of a repository
Master branch is the default name for the first branch created in a new repository.

Merge

In Git, merge is a command used to combine changes from one branch into another.
It integrates the histories of the branches, allowing the integration of new features or fixes into the main codebase.

Origin

In Git, "origin" is the default name given to the remote repository from which local copy was initially cloned.

Pull/Pull Request

'git pull' is a combination of two commands: 'git fetch' and 'git merge'. It fetches the latest changes from the remote repository and automatically merges them with the local branch. It downloads the latest changes and merges them into the current branch, creating a merge commit if necessary.
Pull: The git pull command fetches and merges changes from a remote repository to a local repository.
Pull Request: A pull request is a proposal to merge changes from one branch into another within a repository.

Push

git push command is used to upload local repository content to a remote repository. Pushing is an act of transfer commits from your local repository to a remote repository.

Rebase

Git rebase is a command used to integrate changes from one branch onto another by moving or combining linear commits. It is an alternative to merging and allows for a cleaner commit history. With rebase, you can apply a series of commits from one branch onto another, resulting in a linear commit history.

You would use 'git rebase' when you want to:

- Incorporate changes from one branch onto another with a linear commit history.
- Squash multiple commits into a single commit for a cleaner history.
- Edit or reorder commits to improve readability or resolve conflicts.

Remote

In Git, a remote is a reference to a version of a repository hosted on a server, commonly used to collaborate with others. It allows sharing of code across different machines or teams.

Repository

A **repository (or repo)** is a central storage space where Git stores all the files and directories of a project, along with their complete history. It contains the entire version history of the project, including all the commits and branches.
It keeps track of all changes made to the codebase, allowing developers to collaborate, review, and manage different versions of the project efficiently.

Stash

The `git stash` command is used to temporarily save changes that are not ready to be committed yet. It is useful when you need to switch to a different branch or apply a hotfix but don't want to commit incomplete work. You can later retrieve the changes from the stash using **`git stash apply` or `git stash pop`.**

Tag

A **tag** is a reference to a specific commit that is used to mark a significant point in the project's history, it is used to mark important events like releases or versions.

Tags are often used to create a snapshot of the project at a particular moment.
There are two types of tags.

1. **Light-weighted tag**
2. **Annotated tag**

Upstream And Downstream

"Upstream" refers to the original repository or branch from which the code is cloned, essentially the source of the code.
"Downstream" refers local repository or branches that receive updates from the upstream.

Git Revert

In Git, the revert command is used to create a new commit that undoes the changes made by a previous commit.

```
$ git revert <commit_hash>
```

Git Reset

It is used to unstage the changes from the commit. It will rollback to file into working directory from staging area or index area.

```
$ git reset HEAD <filename>
```

In Git, the reset command is used to undo changes by moving the current branch to a specific commit. It can modify the staging area and the working directory to match the specified commit, effectively rolling back changes.

- Soft: Moves the branch pointer to the specified commit but leaves the changes in the staging area.

  ```
  $ git reset --soft <commit_hash>
  ```

- Mixed: Moves the branch pointer to the specified commit and clears the staging area, but keeps the changes in the working directory.

  ```
  $ git reset –mixed <commit_hash>
  ```

- Hard: Moves the branch pointer to the specified commit and discards all changes in the staging area and working directory.

  ```
  $ git reset --hard <commit_hash>
  ```

Git Ignore

In Git, a **.gitignore** file is used to specify which files and directories should be ignored by Git. This helps keep the repository clean by excluding unnecessary files, such as temporary files, build artifacts, and sensitive information, from being tracked.

Git Diff

It is used to show the differences between two versions of a file, commit, or branch.

```
$ `git diff file.txt`
```

In Git, the diff command is used to show the differences between changes in the working directory, staging area, or between commits. It highlights what changes have been made to the code, making it easier to review modifications.

1. **Show differences between working directory and staging area:**

   ```
   $ git diff
   ```

2. **Show differences between staging area and the latest commit:**

   ```
   $ git diff --staged
   ```

3. **Show differences between two commits:**

   ```
   $ git diff <commit1_hash> <commit2_hash>
   ```

4. **Show differences between a file in two commits:**

   ```
   $ git diff <commit1_hash> <commit2_hash> -- <file_path>
   ```

Git Flow

Git Flow is a branching model for Git that uses multiple branches to manage code changes.

It provides a structured and efficient workflow for managing feature development, releases, and maintenance. It defines a set of roles and rules for branches to ensure a smooth collaboration process.

Key Branches in Git Flow

1. **master**: The production-ready branch containing the stable and deployable code. It contains the production-ready code.
2. **develop**: Serves as the integration branch where feature branches are merged before being released.
3. **feature**: Temporary branches used to develop new features. These are branched from develop.
4. **release**: Used to prepare for a new production release. Created from develop and merged into both master and develop after the release.
5. **hotfix**: Used to quickly fix issues in production. Created from master and merged back into both master and develop.

Git Squash

In Git, squash refers to the process of combining multiple commits into a single commit for a cleaner history.

- **Squash the last n commits:**

```
$ git rebase -i HEAD~<n>
```

- Replace <n> with the number of commits to squash.
- In the interactive rebase editor, change "pick" to "squash" (or "s") for the commits to combine.

```
pick <commit-hash> Commit message 1
squash <commit-hash> Commit message 2
squash <commit-hash> Commit message 3
```

- Save and exit the editor. Git will combine the selected commits into one and prompt you to provide a new commit message.

Git Rm

In Git, the rm command is used to remove files from both the working directory and the staging area.

```
$ git rm <file_name>
```

Git Fork

A Git fork is a personal copy of another user's repository
To resolve an issue for a bug that you found, you can:

- Fork the repository.
- Make the fix.
- Forward a pull request to the project owner.

12 Git Commands

There are many different ways to use Git. Git supports many command-line tools and graphical user interfaces. The Git command line is the only place where you can run all the Git commands.
The following set of commands will help you understand how to use Git via the command line.
Basic Git Commands
Here is a list of most essential Git commands that are used daily.
1. Git Config command
2. Git init command
3. Git clone command
4. Git add command
5. Git commit command
6. Git status command
7. Git push Command
8. Git pull command
9. Git Branch Command
10. Git Merge Command
11. Git log command
12. Git remote command

1) Git config command

In Git, the **git config** command is used to set configuration options for Git repositories. like your username, email address, default editor, and other settings. These settings can be applied at different levels: system, global, and local.

Syntax

- **Set user name:**
```
$ git config --global user.name "Your Name"
```

- **Set user email:**
```
$ git config --global user.email "your.email@example.com"
```

- **Set the default text editor:**
```
$ git config --global core.editor "your_editor"
```

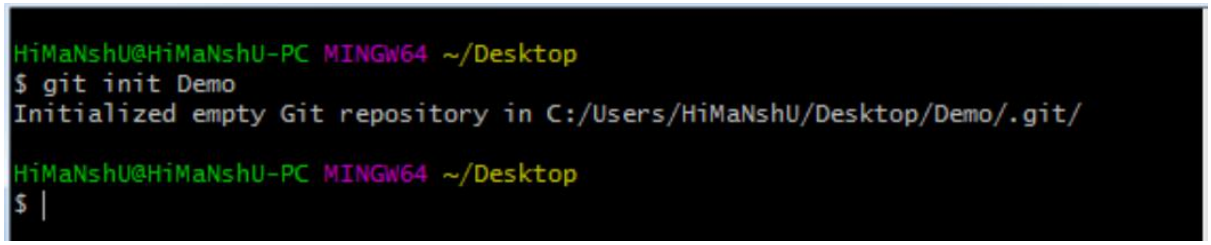
- **View current configuration settings:**
```
$ git config --list
```

2) Git Init command

This command is used to create a local repository. This init command will initialize an empty repository.

Syntax

```
$ git init Demo
```

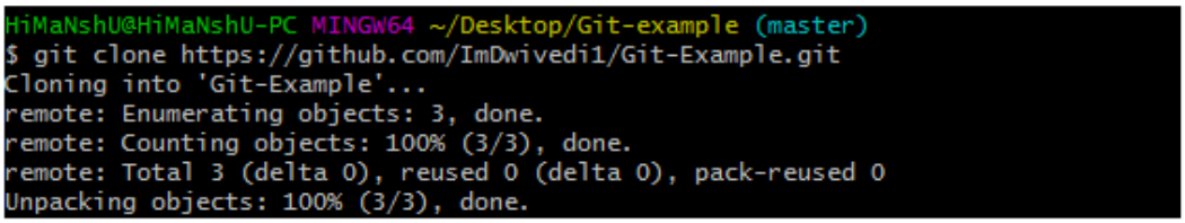


3) Git clone command

The git clone command is used to create a copy of an existing repository.

Syntax

```
$ git clone URL
```



4) Git add command

This command is used to add one or more files to staging (Index) area.

Syntax

To add one file

```
$ git add Filename
```

To add more than one file

```
$ git add*
```

5) Git commit command

Commit command is used in two scenarios. They are as follows.

Git commit -m

The commit command is used to save changes to the local repository.

Each commit represents a snapshot of the project at a specific point in time, allowing for a detailed history of changes and easy rollback if needed.

Syntax

```
$ git commit -m " Commit Message"
```

Git commit -a

This command commits any files added in the repository with git add and also commits any files you have changed since then.

Syntax

```
$ git commit -a
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (master)
$ git commit -a -m "Adding the key of c"
[master (root-commit) 758797a] Adding the key of c
 1 file changed, 2 insertions(+)
 create mode 100644 README.md
```

6) Git status command

The status command is used to display the state of the working directory and the staging area.

Syntax

```
$ git status
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (master)
$ git status
On branch master
Your branch is based on 'origin/master', but the upstream is gone.
  (use "git branch --unset-upstream" to fixup)

nothing to commit, working tree clean
```

7) Git push Command

The push command is used to upload local repository content to a remote repository.

Git push command can be used as follows.

Git push origin master

This command sends the changes made on the master branch, to your remote repository.

Syntax

```
$ git push [remote] master
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (mast
er)
$ git push origin master
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$ git push origin master
Everything up-to-date

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$
```

Git push -all

This command pushes all the branches to the server repository.

Syntax

```
$ git push --all
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$ git push --all
Everything up-to-date

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/gitexample2 (master)
$
```

Push tags:

Syntax

```
$ git push --tags
```

8) Git pull command

The pull command is used to fetch and merge changes from a remote repository into the current branch.

Syntax

```
$ git pull URL
```

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (master)
$ git pull https://github.com/ImDwivedi1/Git-Example
warning: no common commits
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/Git-Example
 * branch            HEAD       -> FETCH_HEAD
fatal: refusing to merge unrelated histories

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/Git-example (master)
$
```

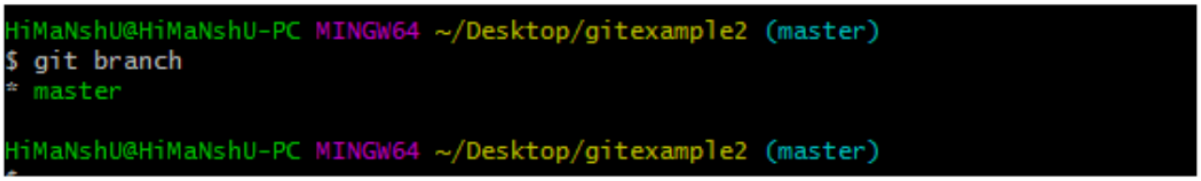
9) Git Branch Command

This command displays a list all the branches available in the repository.

Syntax

- List all branches:
  ```
  $ git branch
  ```
- Create a new branch:
  ```
  $ git branch <branch_name>
  ```
- Switch to a branch:
  ```
  $ git branch <branch_name>
  ```
- Create and switch to a new branch:
  ```
  $ git branch -b <branch_name>
  ```
- Rename a branch:
  ```
  $ git branch -m <old_branch_name> <new_branch_name>
  ```

- Delete a branch:
```
$ git branch -d <branch_name>
```
- Show the last commit on each branch:
```
$ git branch -v
```



10) Git Merge Command

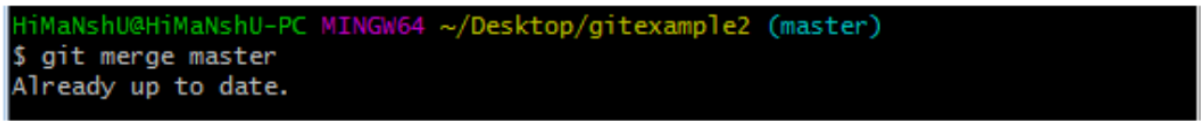
the merge command is used to combine changes from different branches into a single branch.
This command is commonly used to integrate feature branches, bug fixes, or other contributions into the main codebase.
Syntax

- Merge a branch into the current branch:
```
$ git merge <branch_name>
```

- Merge a branch without creating a merge commit (fast-forward merge):
  Performs a fast-forward merge if possible, updating the current branch to match the specified branch without creating a merge commit.
```
$ git merge --ff-only <branch_name>
```

- Squash and merge a branch:
  Combines all commits from the specified branch into a single commit in the current branch.
```
$ git merge --squash <branch_name>
```

- Abort a merge in progress:
  Cancels an ongoing merge operation and reverts the working directory to the state before the merge.
```
$ git merge --abort>
```



11) Git log Command

The log command is used to display the commit history of a repository.
It shows a list of commits along with information like commit hashes, authors, dates, and commit messages.

This command is used to check the commit history.
Syntax

- Log command:

```
$ git log
```

- Show a specified number of commits:

```
$ git log -n <number>
$ git log -n 5
```

- Show commits by a specific author:

```
$ git log --author="Author Name"
```

- Show commits containing a specific keyword:

```
$ git log --grep="keyword"
```

- Show a summary of each commit (shot format - one line per commit):

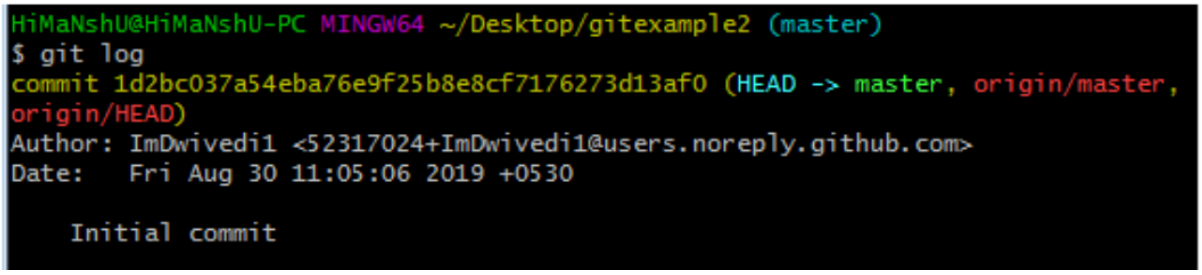
```
$ git log --oneline
```

- Show a graphical representation of the commit history:

```
$ git log --graph
```

- Show changes introduced by each commit:

```
$ git log -p
```



12) Git remote Command

The remote command is used to manage and interact with remote repositories.

Syntax

- List all remote repositories:

```
$ git remote -v
```

- Add a new remote repository:

```
$ git remote add <remote_name> <repository_url>
```

- Remove a remote repository:

```
$ git remote remove <remote_name>
```

- Rename a remote repository:

```
$ git remote rename <old_name> <new_name>
```

- Show details of a specific remote:

```
$ git remote show <remote_name>
```

- Update remote references:

```
$ git remote update
```

Git Flow / Git Branching Model

Git Stash
Git stash
Git stash save
Git stash list
Git stash apply
Git stash changes
Git stash pop
Git stash drop
Git stash clear
Git stash branch

Git Tags

Git Merge Conflict
$ git mergetool

https://media.licdn.com/dms/document/media/v2/D4E1FAQG-A8EirXkWMQ/feedshare-document-pdf-analyzed/B4EZQmafNuGYAc-/0/1735811358843?e=1737590400&v=beta&t=VKeR0i-B-kjaUxRHrvZYRf1wUuBSmYR_DtfGrXMh0EY

https://media.licdn.com/dms/document/media/v2/D561FAQGHhm0aWFvW6Q/feedshare-document-pdf-analyzed/B56ZQLw9WMGQAY-/0/1735364193676?e=1738195200&v=beta&t=oq4Dejgh_RhLu2JGc-KYsfcrcLuStAkw5dHmGbnzHWc

Docker

Docker is containerization technology tool.
Kubernetes is container orchestration tool.

Docker
Docker is mini-OS
Lightweight containers and fast responsible.

Docker has container runtime, which will allow you to run container or manage the life cycle of the container.

Container is ephemeral - short life or lived, containers can die or revive anytime.

Containerization:
It's all about deploy application with required dependencies is known as containerization.

Kubernetes:
Kubernetes is an orchestration platform for running and managing container for many (100's of) containers runtimes.

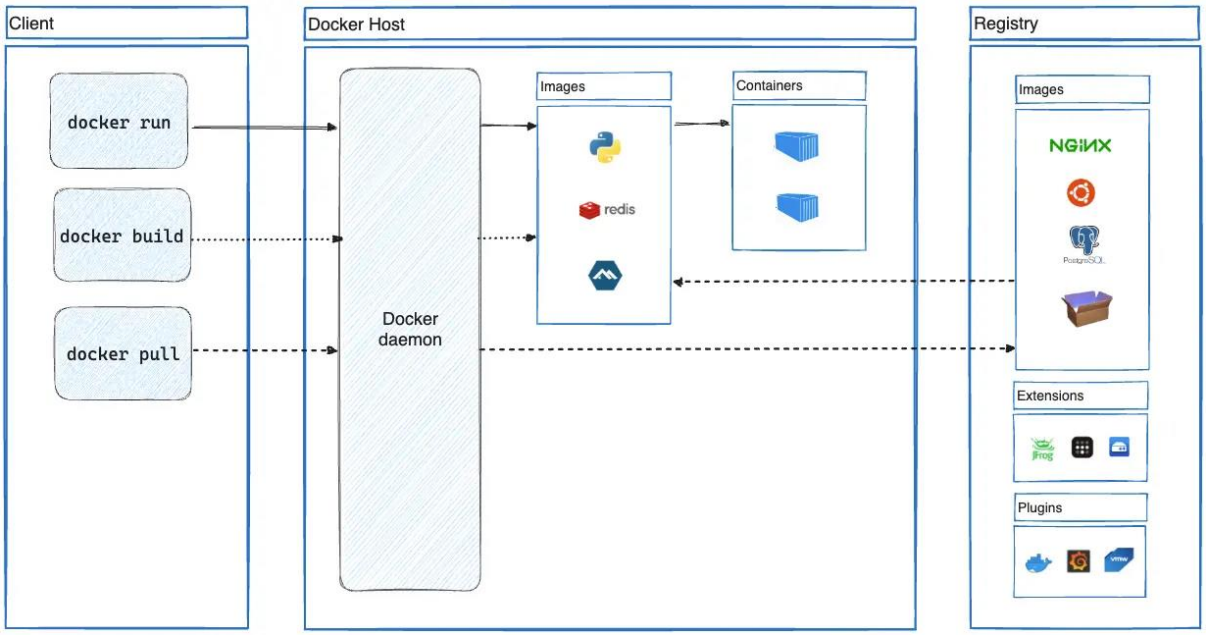
Virtualization:

-

Container Engine:
Software which helps to implement containerization on a machine or server is caller container Engine.

Tool:
Docker, Jail, Crio.

Docker Architecture:



Docker CLI/Client:

Docker client is a tool helps interact with Docker and manage containers, images, networks, and volumes directly from the command line. It is a powerful interface for Docker, providing commands to perform tasks such as building, running, and stopping containers, as well as managing Docker images and networking.

Some common Docker CLI commands include:

- **docker run**: Used to create and start a container from images.
- **docker build**: Builds a Docker image from a Dockerfile.
- **docker pull**: Downloads an image from a Docker registry like Docker Hub.
- **docker ps**: Lists all running containers.
- **docker images**: Lists all available Docker images in server.
- **docker push**: Uploads an image to a Docker registry.
- **docker stop**: Stops a running container.
- **docker rm**: Removes a stopped container.
- **docker exec:** Executes a command inside a running container.
- **docker start:** Starts a stopped container.
- **docker logs:** Displays logs from a container.
- **docker attach:** Attaches to a running container's main process (useful for debugging).

Docker Daemon (Docker service):

Docker daemon manages all the services by communicating with other daemons (**services**). It manages docker objects such as images, containers, networks, and volumes with the help of the API requests of Docker.

(The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.)

Docker Host
A **Docker Host** refers to a machine or environment where Docker is installed and runs.
Docker Host is responsible for managing Docker containers and images, as well as handling the resources (CPU, memory, disk, etc.) needed to run them.
In simple terms:

- **Docker Host** is the machine or environment where the **Docker Engine** is running.
- It provides the environment where containers are created, managed, and executed.

Docker Registry
All the docker images are stored in the docker registry.
Registry: It is a collection of all dependencies and docker container images

Docker pull: It helps to download the images or dependencies from registry on to the docker host.
Docker pull <image name:version>

Docker Images
An image contains instructions to create a docker container. It helps to create container from docker images. Image is collection of layers. (Hash tag or commit id)

A Docker image includes the application code, libraries, dependencies, tools, and other files required to run an application.

Docker Container
Containers are created from docker images. With the help of Docker API or CLI, we can start, stop, delete, or move a container.

Docker container is a runtime instance of an image. It will allow to package applications with all parts needed such as libraries and other.

Docker Services:

➔ After docker install, we must follow 3 steps (or) activities.
1. Start the docker services.
   Ex: **service docker start**
2. Enable docker services at boot time.
   chkconfig docker on
3. Add the user account to the docker root group.
   Ex: **usermod -a -G dockerroot <username>**

➔ After installing docker, it will create group called dockerroot. We must add user account to the docker root group.

Docker commands:

How to check docker client and engine version?
docker version

How to check docker server configuration?
docker info

How to find the container image?
Docker images

How to check the container information?
Docker ps -a
Or
Docker ps -> it will shows only live or running containers

How to create images?
Docker build –tag <image Name> <file path to docker file>

How to create a container?
Docker run -it <image name or Id> <container shell ex: sh, bash>
Note : -it is used for create and log into the container

How to close a container and come out?
Type exit in container and click on enter.

How to come out from container without closing session?
Ctrl + pq

How to stop container?
Docker container stop <container id or name>

How to start container?
Docker container start <container id or name>

How to log into the container or docker exec command?
Docker exec -it <container name or id > <container shell>

Note: the container should be running and up to log into the container
If container is stopped or not running we have to start and log in.

How to start and log into a container?
Docker start -ai <container name or id >
Note: Here we can't change the shell, default shell is sh

Create or update container names?
Docker rename <old container name > <new container name>

How to name a container while creating?
Docker run -name <container name> -it <image id> <container shell>

How to delete containers?
Docker rm <container id or name>
Note: always delete a container when status is showing as exited.

docker rm <container id or name> --force (to remove container forcefully)

How to delete multiple containers?
Docker rm < container name1> name2 name3 etc…

How to delete docker images?
Docker rmi <image name or id >
Note: it will only delete images not the container running from those images.

Image tagging:
Two types of image tags are available.

1. **Local tag**
   Local tag helps to create an alias name use case on docker server or host. We can't upload local tag images to docker registry
   Docker image tag <source image name or id > <new image name >

2. **Remote tag**
   We can generate image alias both for use case or keeping images on docker host and upload to the docker registry.
   Docker tag <source image name or id > <docker hub id>/<new image name>:<image version>

How to login to docker hub from cli mode?
Docker login

How to push images to docker registry or hub?
Docker push <image name>
Note: **only use image names to push images to docker hub**

How to logout from docker hub?
Docker logout

Docker image inspect:
Inspect command shows the properties of the image.

How to get the image meta data info or properties?
Docker image inspect <image id or name>

How to get the container meta data info or properties?
Docker container inspect <container name or id>

Docker image history:
This command helps to check the layers of the image or code of the image.
Docker image history

Docker layer:

Docker build follows the process of interpretation it will check the code line by line.
If we have 4 lines in a code
First line code is correct it will create hashtag or commit id. If the hashtag crated this will generate a layer.

Image is a collection of layers
In dockerfile output layers are arranged in descending order
The last line of the layer in an image automatically considered as image id.
Each line of code in dockerfile we have to call as instruction.
If instruction executed correctly docker will create layer, all these instructions are written in a file is called dockerfile. For docker file no extension.

Docker build is process of converting instructions into layers if all layers are successfully executed then it will generate a docker image else if one of the instructions fails then layer will not generate. If one of the layers fails then docker image fails to generate.

Dockerfile:

- Dockerfile helps to create images,
- File contains instructions.
- These instructions more look like Linux commands.

Work flow of docker image process:

1. Crete project folder
2. Cd into folder, create dockerfile. Name is Dockerfile.
3. Open the dockerfile, write the code and save it.
   Eg: FROM python:3.12
   WORKDIR /usr/local/app
   # Install the application dependencies
   COPY requirements.txt ./
   RUN pip install --no-cache-dir -r requirements.txt
   # Copy in the source code
   COPY src ./src
   EXPOSE 5000
   # Setup an app user so the container doesn't run as the root user
   RUN useradd app
   USER app
   CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
4. Excute it (Image build process)
   Commad:
   Docker image build –tag <image_name > <docker file path>
5. Result: docker ps
6. Then push to docker hub.

Docker instructions:

- **FROM <image>** - this specifies the base image that the build will extend.
- **WORKDIR <path>** - this instruction specifies the "working directory" or the path in the image where files will be copied and commands will be executed.

- **COPY <host-path> <image-path>** - this instruction tells the builder to copy files from the host and put them into the container image.
- **RUN <command>** - this instruction tells the builder to run the specified command. It executed at time of image build process.
- **ENV <name> <value>** - this instruction sets an environment variable that a running container will use.
- **EXPOSE <port-number> -** this instruction sets configuration on the image that indicates a port the image would like to expose.
- **USER <user-or-uid>** - this instruction sets the default user for all subsequent instructions.
- **CMD ["<command>", "<arg1>"]** - this instruction sets the default command a container using this image will run. It executed at time of container creation.
- **ENTRYPOINT [command param1 param2]:** specifies the main command to execute when a container starts.

For more information about docker installations:
https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/

Docker Networking

Docker Networking refers to how Docker containers communicate with each other and the outside world. Docker uses a network model that allows containers to connect to one another and to external systems while ensuring isolation, security, and scalability.

Key Concepts in Docker Networking:

1. **Container Networking**: Each container gets its own network stack (IP address, ports, etc.) when it is created, and it can communicate with other containers and external services based on its network configuration.
2. **Network Drivers**: Docker uses different network drivers to control how containers communicate. Each driver defines a networking mode (e.g., bridge, host, overlay, etc.) and manages the behaviour of containers in that network.

   The Docker Network command is the main command that would allow you to create, manage, and configure your Docker Network. Let's see what the sub-commands can be used with the Docker Network command. to know more about Creating a Network in Docker and Connecting a Container to That Network.

3. **IP Addresses and Ports**: Each container gets its own IP address on the network. Containers can expose ports, allowing external systems to communicate with them (e.g., HTTP on port 80).
4. **Network Isolation**: Containers can be isolated in different networks for better security and control. By default, containers that are part of the same network can communicate with each other, while containers on different networks are isolated.

Docker Network Drivers:
Docker supports several types of network drivers, each designed for different use cases:
1. Bridge Network (default)

- **Bridge networks** create a software-based bridge between host and the container. Containers connected to the network can communicate with each other, but they're isolated from those outside the network.

- **Description**: This is the default network driver when you create a container. Containers on a bridge network are connected to a virtual bridge (a software-based network bridge) on the host system. Containers can communicate with each other through this bridge and can be accessed externally by exposing ports.

- **bridge:** If you build a container without specifying the kind of driver, the container will only be created in the bridge network, which is the default network.

Run a container on the bridge network (default)
$ docker run -d --name my_container --network bridge my_image

2. Host Network

- **Description**: When a container is run on the host network, it shares the host's network stack directly. The container doesn't get its own IP address but uses the host's IP and network interfaces. It can access all ports and services available on the host machine.

- **host:** Containers will not have any IP address they will be directly created in the system network which will remove isolation between the docker host and containers.

Run a container on the host network
$ docker run -d --name my_container --network host my_image

3. Overlay Network

- **Description**: Overlay networks allow containers to communicate across multiple Docker hosts. This network driver is commonly used when deploying applications in a **Swarm** or **Kubernetes** cluster. It abstracts the underlying network and connects containers across different hosts over a virtual network.
- **Use Case**: Ideal for multi-host communication in a Docker Swarm or Kubernetes cluster.

- **overlay:** overlay network will enable the connection between multiple Docker demons and make different Docker swarm services communicate with each other

Create an overlay network
$ docker network create --driver overlay my_overlay_network

Run containers on the overlay network (example in Swarm mode)
$ docker service create --name my_service --network my_overlay_network my_image

4. None Network

- **Description**: This driver disables networking for the container entirely. The container will not have any network interfaces or be able to communicate with other containers or the external network.

- **Use Case**: Used when you want to isolate a container from any networking (e.g., for security reasons or when running an application that doesn't require networking).

- **none:** IP addresses won't be assigned to containers. These containments are not accessible to us from the outside or from any other container.

Run a container with no network access
$ docker run -d --name my_container --network none my_image

5. Macvlan Network

- macvlan is another advanced option that allows containers to appear as physical devices on your network. It works by assigning each container in the network a unique MAC address.

- **Description**: The Macvlan driver allows containers to appear as individual devices on the physical network. Each container gets its own MAC address and IP address, making it behave as if it is a physical device connected to the network.
- **Use Case**: Useful when you need containers to directly interact with the physical network or when legacy applications require containers to have a unique MAC address and IP.

- **macvlan:** macvlan driver makes it possible to assign MAC addresses to a container.

Create a Macvlan network
$ docker network create -d macvlan --subnet=192.168.1.0/24 --gateway=192.168.1.1 my_macvlan_network

Run a container on a Macvlan network
$ docker run -d --name my_container --network my_macvlan_network my_image

6. ipvlan Network:

- IPvLAN is an advanced driver that offers precise control over the IPv4 and IPv6 addresses assigned to your containers, as well as layer 2 and 3 VLAN tagging and routing.
- Users have complete control over both IPv4 and IPv6 addressing by using the IPvlan driver.

Common Docker Networking Commands:

- List Networks:
  $ docker network ls

- Create a Network:
  $ docker network create --driver bridge my_bridge_network

- Inspect a Network (get detailed information about a network):
  $ docker network inspect my_bridge_network

- Connect a Container to a Network:
  $ docker network connect my_bridge_network my_container

- Disconnect a Container from a Network:
  $ docker network disconnect my_bridge_network my_container

- View Container's Network Information:
  $ docker inspect my_container

Exposing Ports for External Access:
In Docker, containers typically run in isolated networks and cannot be accessed externally unless specific ports are exposed. You can map container ports to host ports using the -p or --publish option.

Run a container and expose its port (e.g., 80 in container to port 8080 on the host)
$ docker run -d -p 8080:80 my_image
This allows you to access the container's service (running on port 80 inside the container) via the host machine's port 8080.

sudo docker network

How to create a docker network.
$ sudo docker network create --driver <driver-name> <bridge-name>

Using the "Connect" command, you can connect a running Docker Container to an existing Network.
sudo docker network connect <network-name> <container-name or id>

Using the Network Inspect command, you can find out the details of a Docker Network.
You can also find the list of Containers that are connected to the Network.
sudo docker network inspect <network-name>

The disconnect command can be used to remove a Container from the Network.
sudo docker network disconnect <network-name> <container-name>

You can remove a Docker Network using the rm command.
Note that if you want to remove a network, you need to make sure that no container is currently referencing the network.
sudo docker network rm <network-name>

To remove all the unused Docker Networks, you can use the prune command.
sudo docker network prune

Docker Networking – Basics, Network Types & Examples
https://spacelift.io/blog/docker-networking

Docker Volume and Bind Mounts:

In Docker, **volumes** and **bind mounts** are both used to persist data and share data between the host system and containers. **Docker volumes and bind mounts** are two ways to manage persistent data in Docker containers.
Here's an explanation of each:

1. Docker Volumes
A **Docker volume** is a specialized storage mechanism managed by Docker, designed for storing data that should persist even after containers are stopped, removed, or recreated. Volumes are stored in a specific location on the host filesystem, but Docker manages the underlying filesystem for the volume.

Key Features of Volumes:

- **Managed by Docker:** Volumes are stored outside the container filesystem, managed, and maintained by Docker.
- **Persistence**: Data stored in volumes persists even if a container is deleted and restarted. Volumes can be reused across multiple containers.
- **Isolation**: Volumes are independent of the host filesystem, making them more portable and secure. Docker handles the storage location.
- **Sharing Data**: Volumes can be shared and reused by multiple containers. For example, multiple containers can mount the same volume, enabling data sharing between them.
- **Backup and Restore**: Volumes are easier to back up and restore compared to bind mounts because Docker provides built-in tools.

Example:
Create a volume
$ docker volume create my_volume

Run a container with a volume mounted
$ docker run -d -v myvolume:/data myimage
$ docker run -d -v my_volume:/data my_container

2. Bind Mounts
Bind mounts allow you to mount a directory or file from the host system directly into a container. This means that changes made inside the container will be reflected on the host, and vice versa.

Bind mounts allow you to directly access and modify files and directories on the host machine from within the container.

Key Features of Bind Mounts:

- **Direct Host Access**: Bind mounts provide direct access to the host filesystem. You specify an exact path on the host (e.g., /path/on/host), and Docker mounts it to the container.
- **More Flexibility**: You can mount specific files or directories from the host into a container, allowing for custom configurations, shared logs, or easy development workflows.
- **Less Isolation**: Since bind mounts expose the host filesystem directly, they are less isolated than volumes, which may present security risks.

- **Performance**: Bind mounts can be faster for certain use cases, such as when you're working with large amounts of data and need direct access to the host filesystem.

Use Case:
Bind mounts are often used when you want a container to access files or directories that are located on the host system (e.g., source code during development or configuration files).

Example:
Run a container with a bind mount (host directory to container directory)
$ docker run -d -v /path/on/host:/data myimage
$ docker run -d -v /host/path:/container/path my_container

Comparison Between Docker Volumes and Bind Mounts

Feature	Docker Volumes	Bind Mounts
Storage Location	Managed by Docker (location depends on Docker setup)	Defined by the user on the host filesystem
Persistence	Data persists after container removal	Data persists unless the file or directory is deleted from the host
Backup and Restore	Easier, as volumes are managed by Docker	Backup and restore must be handled manually
Portability	Portable across hosts (if shared between containers)	Tied to the host filesystem
Security	More isolated from the host filesystem	Direct access to the host filesystem
Performance	May have slight overhead due to Docker management	Faster in some cases since it's a direct mount
Use Cases	Ideal for persistent storage (databases, logs, etc.)	Ideal for shared development files or custom configurations

---

When to Use Volumes vs Bind Mounts:

- **Use Volumes** when:
  - You need persistent, portable storage that is independent of the host filesystem.
  - You are running databases or applications that need reliable, managed storage.
  - You want Docker to handle storage management for backups, snapshots, and migration.
- **Docker Volumes** are typically preferred for production environments due to their portability, easier management, and Docker's built-in support for persistence and backup.

- **Use Bind Mounts** when:
  - You need to share files or directories directly between the host and the container (e.g., for development, configuration files, or logs).
  - You need more control over the location of the data on the host.
  - You want to mount specific files (e.g., a configuration file) into a container.

- **Bind Mounts** provide more flexibility and are useful for development or situations where you need direct access to the host filesystem.

Docker Volumes – Guide with Examples

https://spacelift.io/blog/docker-volumes

Docker Compose

Docker Compose is a tool that makes it easier to create and run multi-container applications. It automates the process of managing several Docker containers simultaneously, such as a website frontend, API, and database service.

Dockerfile

```
# syntax=docker/dockerfile:1
FROM python:3.10-alpine
WORKDIR /code
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run", "--debug"]
```

compose.yaml

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

docker compose up
docker compose down
docker compose stop

Docker Compose – What is It, Example & Tutorial
https://spacelift.io/blog/docker-compose

Docker Security:

Docker security is essential for ensuring the safe operation of containers in both development and production environments. By applying these security practices and tools, you can minimize risks, protect the host system, and prevent attacks. Security should be an ongoing process, with regular updates, scans, and reviews to ensure Docker environments remain secure.

1. Image Security
Use Official Images: Official images on Docker Hub are often vetted for security issues. You should prefer official or trusted images over random, unverified ones.
Scan for Vulnerabilities: Docker images may contain vulnerabilities. You can use tools like Docker's docker scan (integrated with Snyk) to scan images for known vulnerabilities.
Minimize Base Image: Use minimal base images (e.g., alpine or scratch) to reduce the attack surface, avoiding unnecessary software in the image.

2. Container Isolation
Namespace Isolation: Docker uses namespaces to isolate containers from each other and from the host system. This helps ensure that one container cannot affect the others or the host system directly.
Control Groups (cgroups): Docker uses cgroups to control resources allocated to containers (like CPU, memory, and disk). This prevents one container from consuming too many resources and impacting others.
Seccomp Profiles:
AppArmor/SELinux:

3. User and Permissions Management
Least Privilege Principle: Containers should run with the least privileges required. This means avoiding running containers as the root user when unnecessary.
File Permissions: Properly manage file permissions in your container and on the host system to prevent unauthorized access.
User Namespaces:

4. Docker Daemon Security
Docker Daemon Socket: By default, the Docker daemon listens on a Unix socket (/var/run/docker.sock), which is accessible to users in the docker group. Restrict access to this socket to prevent unauthorized control over Docker.
Remote API Security: If you're using the Docker Remote API, ensure it is securely configured using TLS/SSL to encrypt communication. Access should be controlled with appropriate authentication mechanisms.
Limit Docker Daemon Privileges: Ensure that the Docker Daemon itself is running with the least privileges required, and limit its exposure to unnecessary services.

5. Networking Security
Network Segmentation: Use Docker's built-in networking capabilities (e.g., bridge, overlay) to isolate containers and limit network communication between them, based on their needs.
Secure Connections: Containers communicating over the network should use encryption (e.g., HTTPS, TLS) to prevent data from being intercepted.
Firewalls: Control network access to Docker containers using firewalls, either on the Docker host itself or within the container's network settings.

6. Logging and Monitoring
Log Container Activity:
Monitor Resources:
Audit Docker Commands:

7. Container Runtime Security
Runtime Security Tools:
Container Scanning at Runtime:

10. Vulnerability Patching
Regularly Update Images: Regularly pull the latest versions of images from official repositories to ensure you get security patches and updates.
Update Docker and Host Systems: Make sure that both Docker and the host system are kept up-to-date with the latest security patches and updates.

11. Security Tools for Docker
Trivy: A simple and comprehensive vulnerability scanner for containers, which checks images for vulnerabilities in OS packages and application dependencies.

12. Orchestration Security
When using orchestration systems like Docker Swarm or Kubernetes, consider the security of the cluster, such as secure communication (TLS), role-based access control (RBAC), and secrets management.

Image layers
what is the Base layers in docker file?
Latest version: 1.27.0-alpine
If the layers are more, then it can be easy to hack
Docker exec -it
we need to create new user to login into container.
It should username@container name.

trivy - to scan the images - it will teach in pipeline projects.

Docker ignore --
We should not use add command, this comes under security.

Add command can also download the files from url, because of this there might causes of hack.

Environmental variables, username and password.
Don't hard code the passwords and username.
User docker secretes.

Interview questions
1. which image have you used, base image or latest image?
2. which user have you user, either root user or different user?
3. don't provide the privileges for some users.
4. docker instructions commands
5. what is the Base layers in docker file?

Docker Troubleshooting:
Top
Stats
Logs
Logs -f
exce -it
inspect volume, network,
History

Docker compose and docker multi stage build.

Kubernetes

Kubernetes:

- Kubernetes is an open-source Container orchestration platform that automates the deployment, management, scaling, and networking of containers across the cluster.
- It is focused on managing the life cycle of containers. It schedules, run, manage isolated containers which are running on virtual/physical/cloud machine
- Kubernetes is a cluster. Cluster is a group of nodes, that run containerized applications.

Docker is container platform
Kubernetes is a container orchestration platform
Orchestration tool = Container Management tool

Docker
Docker has container runtime, it will allow to run container or manage the life cycle of the container.
Container is ephemeral - short life or lived, containers can die or revive anytime.

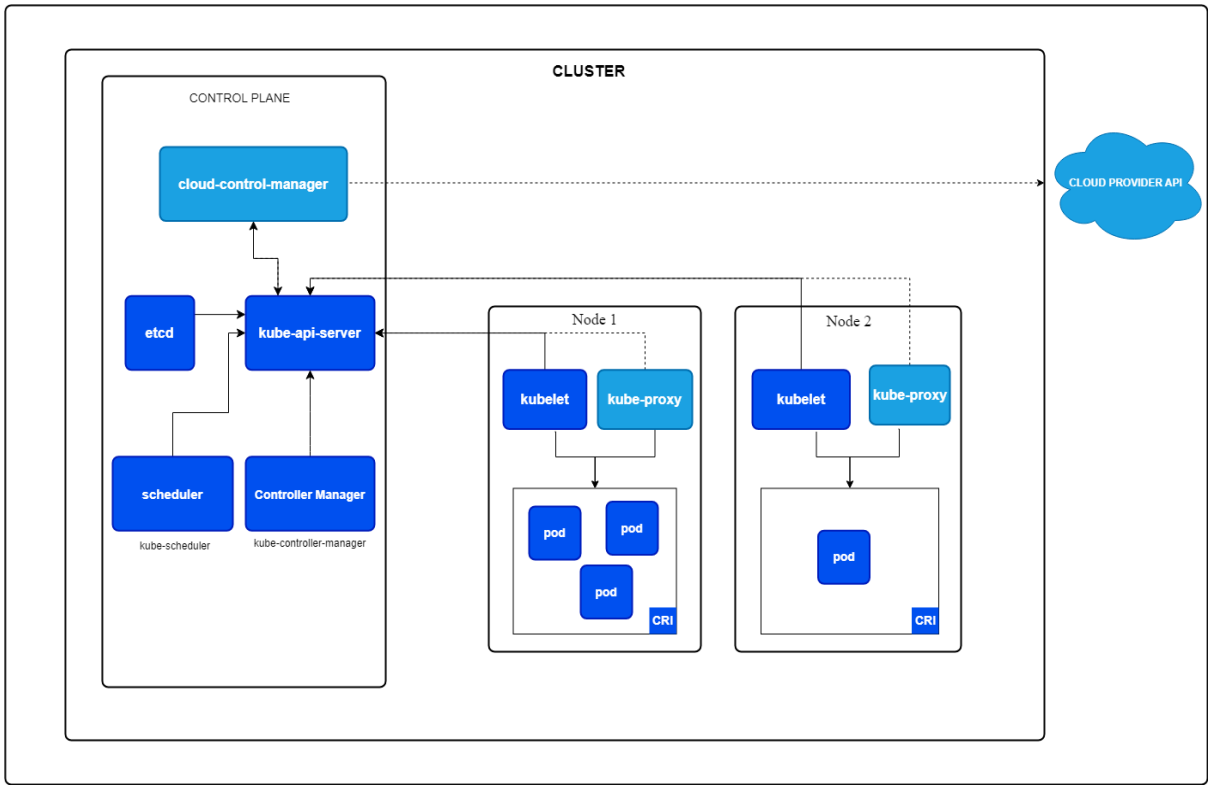
- Problem with scaling up the container:
- can't communicate with each other.
- Auto scaling and auto healing
- load balancing
- container had to be managed carefully.
- Single host nature of docker container
- **Enterprise level support**.
    - Auto scaling
    - Auto healing
    - Load balancer support
    - Firewall support
    - Api support gateways
    - White listing
    - Black listing

Advantages for k8s:

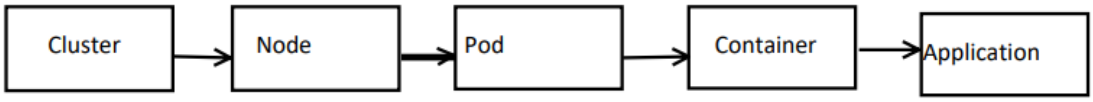
- **Container Orchestration:** It automates the management of containers running across a cluster of machines. Kubernetes ensures that containers are deployed, scaled, and maintained consistently.
- **Auto Scaling**: Kubernetes can automatically scale applications up or down based on resource utilization (like CPU or memory) or other custom metrics.
- **Auto Healing:** Kubernetes automatically replaces failed containers, restarts them, or reschedules them as necessary, ensuring the application remains available.
- **Load Balancing:** It automatically distributes incoming traffic to containers to ensure that no single container is overwhelmed, improving the overall performance and availability.
- **Service Discovery and Networking:** Kubernetes helps containers communicate with each other by providing a DNS-based service discovery system.

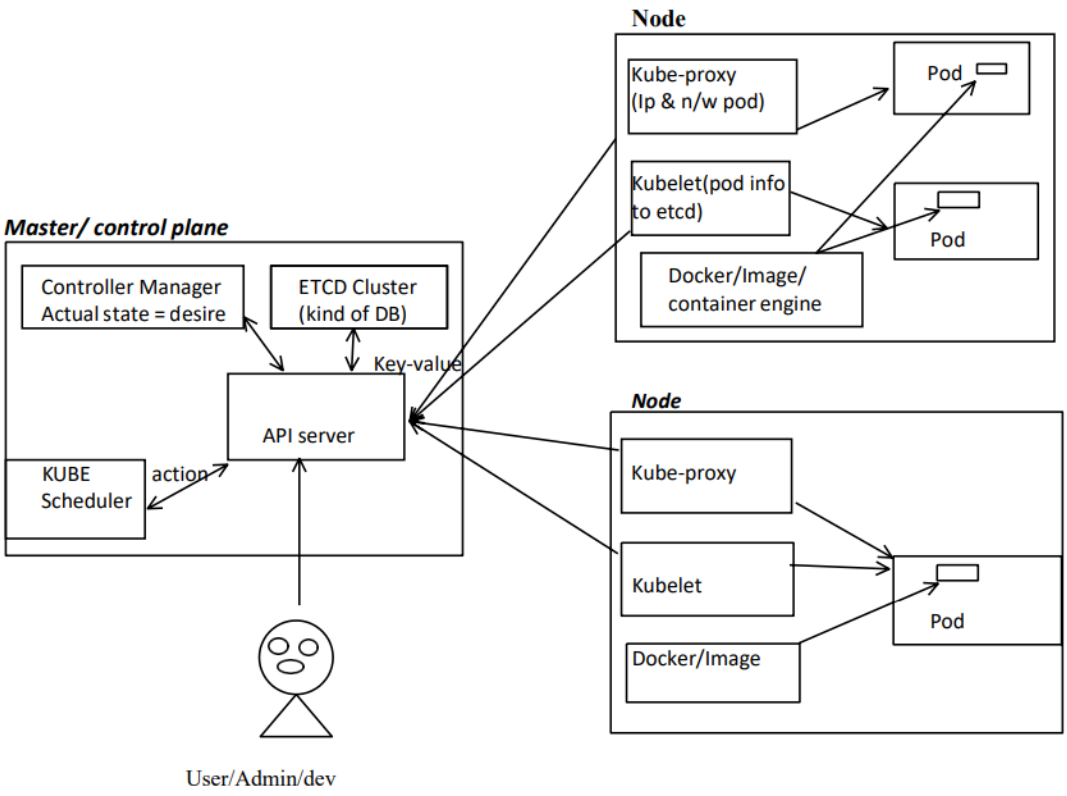
- **Platform independent (cloud/virtual/physical)**
- **Fault tolerance (node/pod failure)**
- **Rolling application upgrades and downgrades with zero downtime (Rolling back)**
- **Health Monitoring of pod**
- **Updates/new release/deployment**
- **Secrete/config (master node)**

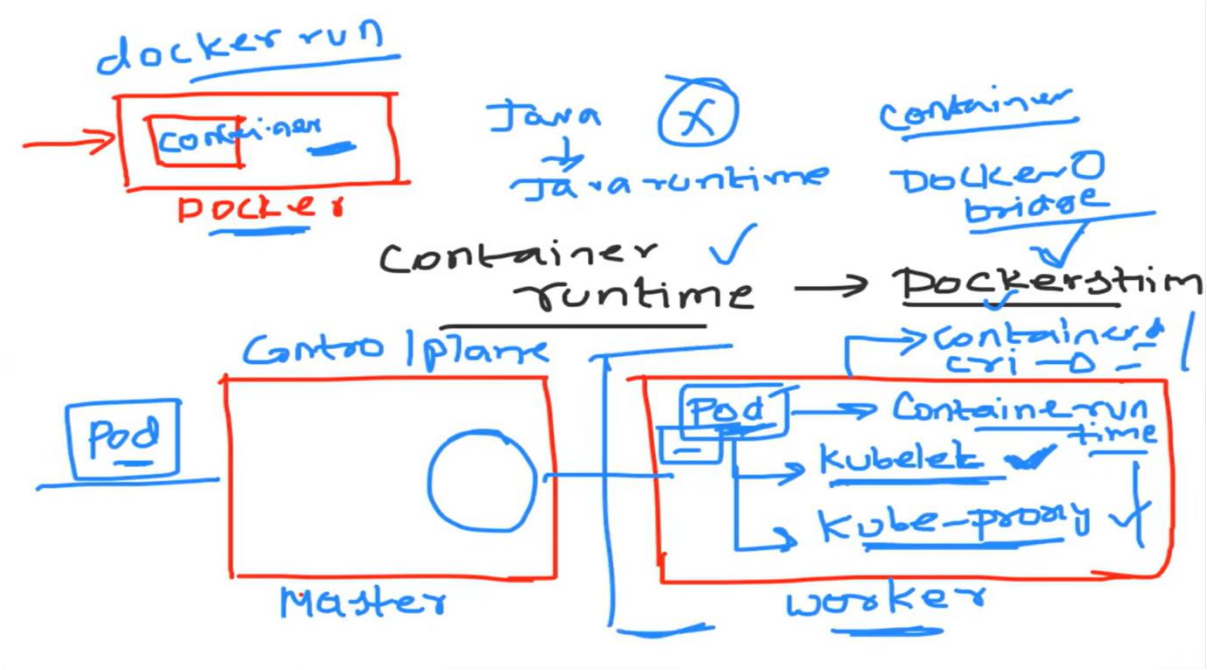
Kubernetes architecture:



Request flow High Level Diagram







Kubernetes architecture allows Kubernetes to provide a scalable, reliable, and flexible platform for managing containerized applications

Kubernetes Workflow:

1. **User interacts with the API server** (using kubectl or other tools).
2. **API server** processes the request and stores the data in etcd.
3. **Scheduler** assigns Pods to appropriate nodes based on resources and constraints.
4. **Kubelet** ensures that containers are running on the nodes.
5. **Kube Proxy** manages network traffic between services and Pods.

Role Of Master:

Kubernetes cluster runs on VM / BareMetal / cloud or mix.
1. K8s having one or more master and one or more workers.
2. The master is now going to run set of k8s process. These process with ensure smooth functioning of master these processes are called control plane
3. Can be multi master for high availability
4. Master runs control plane to run cluster smoothly.

Components Of master plane:

1. API Server
2. ETCD (not part of k8s but without this k8s won't work so consider this also a part of k8s)
3. Kube Scheduler
4. Kube Controller Manager
5. Cloud Controller Manager

1. API Server:

- The API server is a component of the Kubernetes control plane that exposes the Kubernetes API and it is responsible for **handling RESTful API requests**. The API server is the **front end** for the Kubernetes control plane.
- The API server authenticates and authorizes the request, and then sends the request to the relevant K8s components
- It validates and configures API objects (e.g., pods, services, deployments) via the kubectl command-line tool or client libraries.

2. ETC:

- It will **store the entire cluster information as a objects or key value pair.**
- It stores all Kubernetes objects (e.g., Pods, Deployments, Services) and ensures data consistency across the entire cluster.
- A key-value store that stores the configuration data of the cluster, including configurations, secrets, and state of the cluster information.

Features:

a) **Fully replicated:** entire state is available on every node of cluster
b) **Secure:** implements TLS with optional client-certificate authentication
c) **Fast:** benchmark at 10,000 writes per second

3. Kube Scheduler:

- The **Kube Scheduler** is responsible for scheduling the Kubernetes resources or pods on worker nodes.

- It will assign newly created Pods to nodes based on resource availability and policies **or [**based on resource requirements and other constraints (e.g., availability of resources, node affinity, taints/tolerations).**]**

4. Kube Controller Manager:

- The controller manager is responsible for ensuring that **the desired state of the cluster same as the actual state.**
- This component **runs controllers** that manage the state of the cluster. It **watches** the state of the cluster through the **API server** and makes changes to reach the desired state. Examples of controllers include the **Replication Controller** and **Deployment Controller**.
- **Controller** will ensure that the actual state in the yaml manifest is same as desired state.

  Controller Manager types:
  - Service controller
  - Pod controller
  - Ingress controller
  - Deployment controller
  - Replicaset controller
  - DaemonSet controller
  - Job Controller (Kubernetes Jobs)
  - CronJob Controller
  - endpoints controller
  - namespace controller
  - service accounts controller.
  - Node controller

- It manages all the controllers.
- You can extend Kubernetes with **custom controllers** associated with a custom resource definition.

Controller Components:

a) **Node Controller:** for checking of nodes that has detect in cloud after it's stop responding
b) **Route Controller:** Responsible to setting up n/w, route
c) **Service Controller:** Responsible for load balancing
d) **Volume Controller:** Managing Volumes

5. Cloud Controller Manager:

- This component integrates Kubernetes with cloud provider APIs to manage cloud specific resources (e.g., load balancers, storage, etc.)
- The cloud controller manager acts as a bridge between Cloud Platform APIs and the Kubernetes cluster.

Components Of Worker Plane:

1. Kube Proxy
2. Kubelet

3. Pods
4. Container Engine

1. Kube Proxy:

- The **Kube Proxy** is responsible for maintaining network rules for pod communication.
- **Kube Proxy** implements networking rules defined by the API server and ensures that network requests (traffic) are routed to the correct pods across different nodes.
- It manages load balancing and routing the network traffic between services in the cluster.
- It provides networking, IP address and load balancing capabilities to pods.
- This component will distribute to pods
- It runs on each node

2. Kubelet:

- It helps to interact with different Kubernetes services.
- This is responsible for creation of pods and ensuring that pods are in running state.
- **Kubelet** is an **agent** that runs on each worker node and ensures that **containers are running and healthy in a pod** as expected by communicating with the API server to receive instructions and report back the current state of pods.
- Kubelet will check the pods health status.
- The health of pods is checked using **liveness probes** and **readiness probes**. These probes are configured in the pod's definition and helps to ensure the application inside the pod is running properly and can handle requests.

3. Pods:

- Pods are the Kubernetes objects.
- Pods are the smallest deployable units in Kubernetes that can contain one or more containers, which share the same network and storage resources.
- Pod having it’s IP address but container don't have
- Auto scaling and auto healing by default not provided by pod. For this high level k8s object required

4. Container Engine or Runtime:

- The container runtime is responsible for running and managing containers on the worked node. Kubernetes supports several container runtimes, including Docker, containerd, CRI-O and docker shim (Docker runtime)
- Kubernetes interacts with the container runtime via the **Container Runtime Interface (CRI)**.

Kubernetes Cluster Addon Components

- Apart from the core components, the Kubernetes cluster needs addon components to be fully operational. Choosing an addon depends on the project requirements and use cases.

Following are some of the popular addon components that you might need on a cluster.

1. **CNI Plugin (Container Network Interface):** Manages networking for containers, ensuring seamless communication between pods.
2. **CoreDNS (For DNS server):** Acts as Kubernetes built-in DNS server, enabling **DNS-based service discovery** for pods and services.
3. **Metrics Server (For Resource Metrics):** This addon helps to collect performance data and resource usage of Nodes and pods in the cluster.
4. **Web UI (Kubernetes Dashboard):** This addon enables the Kubernetes dashboard to manage the object via web UI.

1. CNI Plugin

What is CNI?

- Container Networking Interface (**CNI**) is a **plugin-based architecture** with vendor-neutral specifications that provide networking capabilities for containers.
- It standardizes container networking across different **container orchestration platforms** such
  - Kubernetes
  - Mesos
  - CloudFoundry
  - Podman
  - Docker
- It was originally designed by the Cloud Native Computing Foundation (CNCF) and is **not specific to Kubernetes**.

🌐 Why Use CNI?

Container networking requirements vary across organizations—some may need advanced **isolation**, **security**, **encryption**, or **scalability**. As container technology evolved, many networking vendors built **CNI-based solutions** to address these varied needs. These solutions are known as **CNI Plugins**.

How CNI Works with Kubernetes

1. **Pod CIDR Assignment**
   - The **Kube-controller-manager** assigns a **pod CIDR** (a unique range of IP addresses) to each node.
2. **Pod Launch via Kubelet**
   - **Kubelet** interacts with the **container runtime** to launch scheduled pods.
   - The **CRI plugin** (part of the container runtime) communicates with the **CNI plugin** to configure pod networking.
3. **Pod-to-Pod Networking**
   - The **CNI Plugin** ensures networking between **pods across different nodes** using an overlay network.

🔐 Key Features of CNI Plugins

1. **Pod Networking**
   - Handles IP address allocation, routing, and connectivity.
2. **Network Security & Isolation**
   - Uses **Network Policies** to control traffic flow between pods and namespaces.

Popular CNI Plugins

- **Calico** – Provides high-performance **networking and security** using BGP.
- **Flannel** – A simple overlay network solution for Kubernetes clusters.
- **Weave Net** – Offers automatic encryption and **peer-to-peer networking**.
- **Cilium** – Uses **eBPF** for efficient network security and observability.
- **Amazon VPC CNI** – Optimized for **AWS Kubernetes clusters**.
- **Azure CNI** – Integrates with **Azure Virtual Networks** for Kubernetes networking.

Kubernetes Networking Based on Hosting Platforms

- **Single Cloud Deployment** → Use kubectl.
- **On-Premise Deployment** → Use kubeadm.
- **Hybrid/Federated Kubernetes** → Use kubefed.

The difference between an object and a resource in Kubernetes.

Object

Anything a **user creates and persists in Kubernetes is an object**. For example, a namespace, pod, Deployment configmap, Secret, etc.

Resource

In Kubernetes, everything is accessed through APIs.
a **resource** is a **specific API URL** used to access an object, and they can be accessed through HTTP verbs such as GET, POST, and DELETE.

https://devopscube.com/kubernetes-objects-resources/

Kubernetes Objects

- **Pod**
- **Services**
- **Deployments**
- **Volume**
- **Namespace**
- **ReplicaSets**
- **Replica Controller**
- **Secrete**
- **Config Maps**
- **StatefulSets**
- **Jobs**
- **Daemon Sets**
- **Label**
- **Ingress**
- **Network**

1) Pod: A thin wrapper around one or more containers.
2) Service: Maps a fixed IP address to a logical group of pods
3) Volume: a directory with data that is accessible across multiple containers in a Pod
4) Namespace: a way to organize clusters into virtual sub-clusters
5) ReplicaSets: Ensure high availability. Ensures a defined number of pods are always running.
6) Replica Controller: Ensures a defined number of pods are always running
7) Secrets: an object that contains a small amount of sensitive data such as a password, a tok1en, or a key
8) Config Maps: Manage app config. an API object that lets you store configuration for other objects to use
9) Deployments: Deployment-manage app updates.
10) StatefulSets: Manage stateful app the workload API object used to manage stateful applications.
11) Jobs: Run & scheduled jobs. Ensures a pod properly runs to completion and stop after process complete it's execution
12) Daemon Sets: Running on each node. Implements a single instance of a pod on all (or filtered subset of) worker node(s).
13) Label: Key/Value pairs used for association and filtering

1) Pod:

- Pods are the Kubernetes objects
- The smallest deployable units in Kubernetes that can contain one or more containers, which **share the same network and storage (volumes) resources.**
- Pods are often ephemeral.
- Each pod is assigned a unique IP address within the cluster.
- **K8s uses YAML file to describe the desired state** of the containers in a pod. This is also called a **Pod Spec**. These objects are passed to the kubelet through the API server.

Imperative vs Declarative commands

- Kubernetes API defines a lot of objects/resources, such as namespaces, pods, deployments, services, secrets, config maps etc.
- There are two basic ways to deploy objects in Kubernetes: **Imperatively and Declaratively**

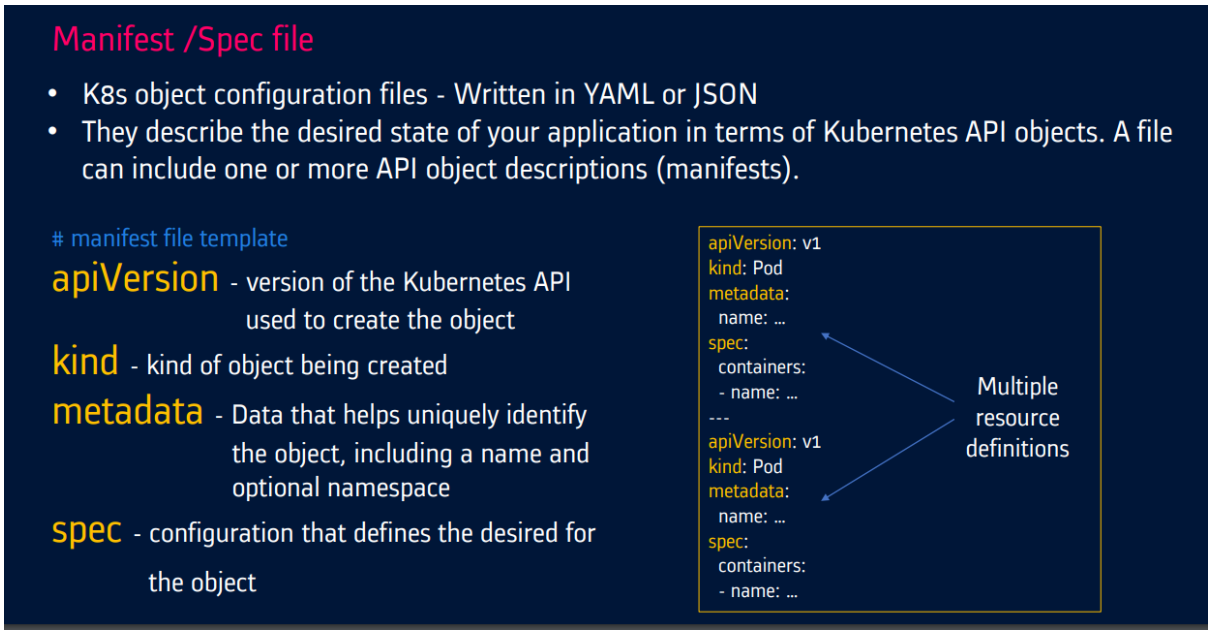
Imperatively Management:

- Involves using any of the verb-based commands like kubectl run, kubectl create, kubectl expose, kubectl delete, kubectl scale and kubectl edit
- Suitable for testing and interactive experimentation

Declaratively Management:

- Objects are written in **YAML files** and deployed using **kubectl create or kubectl apply**
- Best suited for production environments

Manifest /Spec file



Annotations

- **Annotations** are key-value pairs that provide metadata about objects. Unlike labels, which are used to select and organize resources.
- **Annotations** are used to **store non-identifying information** that can be used by tools and libraries.

Eg:
Pod example with annotations, multiple containers, and environment variables

```
apiVersion: v1
kind: Pod
metadata:
  name: Pod examples annotations, multiple containers, environment variables
# Basic Pod configuration with Pod name
  annotations:
    description: "This is an example Pod with annotations."
    config.checksum: "123456789"
    prometheus.io/scrape: "true"
    prometheus.io/port: "8080"
spec:
  containers:
  - name: main-container        # First container in the Pod
    image: nginx
    env:
    - name: MAIN_ENV_VAR
```

```
      value: "MainContainerValue"
  - name: sidecar-container     # Second container in the Pod
    image: busybox
    command: ["sh", "-c", "echo Hello from the sidecar container; sleep 3600"]
    env:                        # Pod Example with Environment Variable
    - name: SIDECAR_ENV_VAR
      value: "SidecarContainerValue"
```

Pod Lifecycle

1.Pending: The Pod is created but not yet running. This happens while Kubernetes schedules the Pod to a node.
2.Running: The Pod is successfully scheduled and all containers are running or in the process of starting.
3.Succeeded: All containers in the Pod have terminated successfully (exit code 0).
4.Failed: At least one container in the Pod has terminated with a non-zero exit code.
5.Unknown: The state of the Pod cannot be determined.

Pod Health check:

The health of pods is checked using **liveness probes** and **readiness probes**. These probes are configured in the pod's definition and help ensure the application inside the pod is running properly and can handle requests.

Liveness Probe

A liveness probe is used to **check if the application inside the pod is still running**. If the liveness probe fails, Kubernetes will restart the pod to try to recover the application.
Example of a liveness probe configuration:

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 3
  periodSeconds: 3
```

Readiness Probe

A readiness probe is used to **check if the application inside the pod is ready to handle requests**. If the readiness probe fails, the pod will be removed from the service's load balancer until it becomes ready again.
Example of a readiness probe configuration:

```
```

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 3
  periodSeconds: 3
```

Types of Probes

Kubernetes supports three types of probes:

1. **HTTP Probes**: Perform an HTTP GET request against a specified endpoint.
2. **Command Probes**: Execute a command inside the container and check the exit code.
3. **TCP Probes**: Perform a TCP check against a specified port.

Example Configuration

Here is an example of a complete pod configuration with liveness and readiness probes:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: my-application:latest
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 3
      periodSeconds: 3
    readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 3
      periodSeconds: 3
```

These probes help ensure that your application is running smoothly and can recover from failures automatically.

Pod contains single container or multiple containers.

Multiple containers

1. Init container.
2. Side-car container

By utilizing **init and sidecar containers**, we can create more robust and flexible deployments that meet the specific needs of your applications.

Init container.

- **Init containers** are special containers that **runs before the main application containers** in a pod.
- These containers are used to perform setup tasks (such as configuration, initialization, or data preparation) that need to be completed before the main application begins running.
- **Containers will delete, post complete the task** and only the main container will remain the same.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-init-container
spec:
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'echo Initializing; sleep 10']
  containers:
  - name: my-app-container
    image: nginx
```

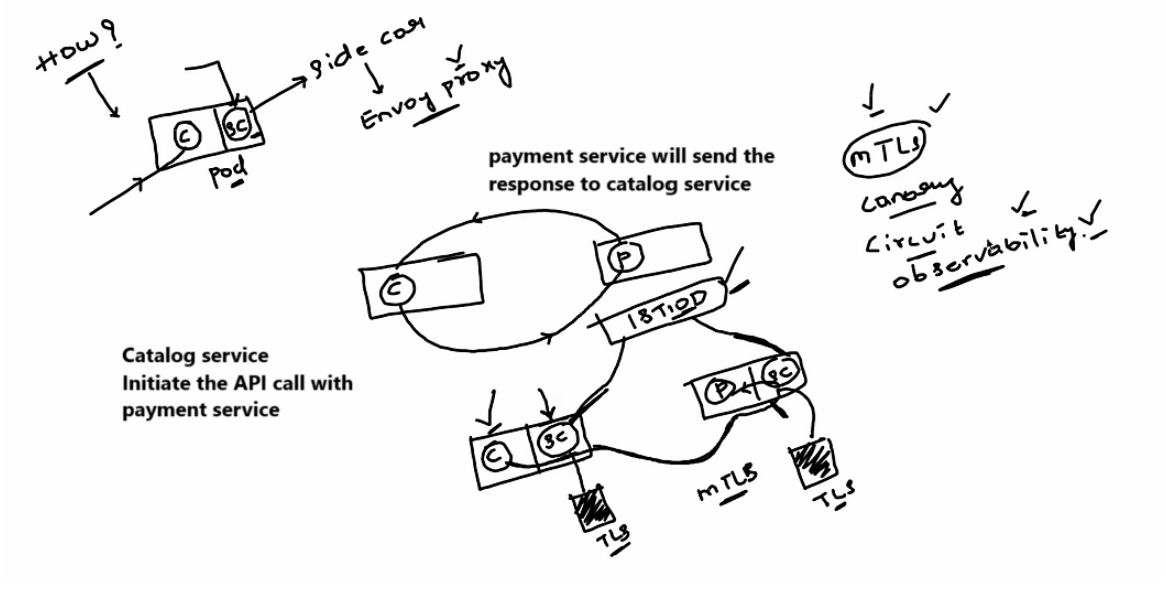
Use cases

- **Initializing shared volumes**
- Data Initialization
- Changing the file system before starting the app container
- Performing **automation backup** before starting an application.
- **Waiting for Dependencies to Be Ready**
  Scenario: A web application that depends on a database being initialized first.
- **Initializing Configurations or Secrets**: Pulling secrets from a vault and storing them in the application configuration file
  Scenario: A containerized application that needs to fetch secrets or configuration files from a centralized service (e.g., AWS SSM or HashiCorp Vault).
- **Preparing Data for the Main Application:**
  Scenario: A container that runs a web application that needs to download an initial set of data or files (like JSON or CSV files) from an external server or cloud storage.

Side-car container

- Sidecar containers are **secondary containers** within the same Pod that **runs alongside the main application container.**

- Sidecar containers extend the functionality of the main container by providing additional services, such as logging, monitoring, proxying, security, or data synchronization.
- Side car container contains envoy proxy application. It is proxy server it will handles the traffic management of pods.




```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-sidecar
spec:
  containers:
  - name: main-app
    image: nginx
  - name: sidecar-logging
    image: busybox
    command: ['sh', '-c', 'tail -f /var/log/nginx/access.log']
    volumeMounts:
        - name: log-volume
          mountPath: /logs
  volumes:
    - name: log-volume
      emptyDir: {}
```

Use Cases:

- **Logging (Fluentd, Elasticsearch)**: A sidecar container can be responsible for gathering logs from the main container and sending them to a logging system or processing them.
- **Monitoring (Metrics and Health Checks)**: A sidecar might collect metrics or health information from the main container for external monitoring purposes.
- **Proxying (Service Proxy or Service Mesh)**: A sidecar container can act as a proxy or a service mesh component (e.g., Istio sidecar) to manage inbound/outbound traffic for the main application container.
- **Security and Identity Management (Secret Injection and Identity Management)**: Sidecars can provide security features like injecting secrets into the environment of the main container.
- **Networking/Proxying (API Gateway/Proxying Requests):** Sidecar containers can serve as proxies or API gateways to route requests between services, for example, caching, rate limiting, or transforming requests.

When to Use Sidecar Containers

- **Sidecar Containers:** Use sidecar containers to add additional functionalities to your application, such as logging, monitoring, or communication proxies, without modifying the main application.

Feature	Init Containers	Sidecar Containers
Execution Time	Runs before the main container starts.	Runs alongside the main container during its lifecycle.
Primary Purpose	Initialization or setup tasks.	Auxiliary tasks that extend functionality (e.g., logging, proxying).
Example Use Cases	Waiting for dependencies, data migration.	Collecting logs, monitoring, proxying traffic.

Type of pods:

1. **Single-container Pods:**
   This is the simplest form of a Pod, containing only one container.
2. **Multi-container Pods:**
   These Pods contain **multiple containers** that need to work together.
3. **Static Pods:**
   Static pods are managed directly by the kubelet on a specific node, rather than through the API server. They are defined in the kubelet's configuration. **(kube-apiserver, etcd, kube-controller-manager, kube-scheduler)**
4. **Ephemeral Pods:**
   Ephemeral containers are used for troubleshooting running pods. They can be added to a pod without restarting it and are removed once the troubleshooting is complete.
5. **Guaranteed Pods:**
   These Pods are designed with **resource guarantees** (CPU and memory) in mind. Kubernetes will ensure that the requested resources are available.
6. **Pod Templates:**
   While not exactly a separate type of Pod, Pod templates are used to define a blueprint for creating Pods.

2) Service

Service:

- **Services:** Services in Kubernetes enables communication between Pods and external clients. They expose applications running inside Pods as **network services**.
- **Load Balancing and Health Monitoring:** A Service is a **round-robin load balancer** for Pods that match its labels or selector, and constantly monitors the Pods. If any Pod becomes unhealthy, the Service reroutes traffic to healthy Pods.
- **Stable Network Endpoint**: Services object provides a **stable network endpoint** to access a **set (group) of Pods**, along with **service discovery and load balancing**.
- Kubernetes **automatically distributes traffic between Pods** and provides **DNS resolution** for Service hostnames.
- Services provide **stable networking, load balancing, and service discovery**, ensuring **consistent communication within the cluster**, even when Pods are dynamically created, scaled, or replaced.
- Service objects is a logical bridge between pods and end user, which provide virtual IP
- **Scalability and Flexibility:** These features are designed to be **scalable, reliable**, and **flexible**, making it easier for developers to manage and orchestrate complex microservices architectures.
- **External Access:** Additional features like **Ingress** and **DNS-based service discovery** provide flexible external access to services and simplify service discovery within the cluster.
- **Networking plugins and configuration** options enable pod-to-pod communication and network isolation.
- Kubernetes uses a combination of **DNS-based service discovery** and **a built-in load balancer** called **kube-proxy**. Each Service is assigned a **unique DNS name**, which resolves to the cluster IP. The kube-proxy component load balances traffic across the Pods associated with the Service, distributing requests evenly.

Why Do We Need Services in Kubernetes?

If service is not available, due some issues, pod has deleted and created again because of auto healing. Pod IP address will change automatically, if IP address changes, we will not able to access the application with old IP address. To overcome this problem, we will create a service yaml file, it will interact with pods using labels and selectors. If any requested came, service file with communication with pods using the labels and selectors.

a) **Pods are Ephemeral:** Pods are temporary and can be destroyed or recreated for reasons like scaling, updates, or failures. Each new Pod gets a different IP address, making direct communication with Pods unreliable.

b) **Stable Communication:** Services provide a consistent way to access the Pods, regardless of changes in the underlying Pods IPs.

c) **Load Balancing:** Services distribute network traffic across multiple Pods.
d) **Service Discovery:** Simplifies communication between Pods by using stable IP addresses and DNS names.

Types of Kubernetes Services:

1. ClusterIP (default):

- Exposes the Service on an internal IP address within the cluster.
- It allows other Pods inside the cluster to communicate with it, but it's not accessible outside the cluster.
- **Use case**: Internal communication between services within the cluster.

```yaml
kind: Deployment
apiVersion: apps/v1
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector: # tells the controller which pods to watch/belong to
    matchLabels:
      name: httpddeployment
  template:
    metadata:
      name: testpod1
      labels:
        name: httpddeployment
    spec:
      containers:
        - name: c00
          image: httpd
          ports:
            - containerPort: 80
---
kind: Deployment
apiVersion: apps/v1
metadata:
  name: ubuntudeployment
spec:
  replicas: 1
  selector: # tells the controller which pods to watch/belong to
    matchLabels:
      name: ubuntudeployment
  template:
    metadata:
      name: testpod2
      labels:
        name: ubuntudeployment
    spec:
```

```
      containers:
        - name: c01
          image: ubuntu
          command: ["/bin/bash", "-c", "while true; do echo Hello-sagar; sleep
5 ; done"]
---
kind: Service # Defines to create Service type Object
apiVersion: v1
metadata:
  name: demoservice
spec:
  ports:
    - port: 80 # Containers port exposed
      targetPort: 80 # Pods port
  selector:
    name: httpddeployment # Apply this service to any pods which has the
specific label
  type: ClusterIP # Specifies the service type i.e ClusterIP or NodePort
```

2. **NodePort**:
    - Exposes the Service on each Node's IP address at a static port.
    - Pods can accessible from outside the cluster using **<NodeIP>:<NodePort>.**
    - Expose the service on the same port of each selected node in the cluster using NAT.
    - It allows external traffic to access the service by connecting to **<NodeIP>:<NodePort>.**
    - **Use case**: When you need external access to your service (but not via LoadBalancer) and you can target specific Node IPs.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector: # tells the controller which pods to watch/belong to
    matchLabels:
      name: httpddeployment
  template:
    metadata:
      name: testpod1
      labels:
        name: httpddeployment
    spec:
      containers:
        - name: c00
```

```
          image: httpd
          ports:
            - containerPort: 80
---
kind: Service # Defines to create Service type Object
apiVersion: v1
metadata:
  name: demoservice
spec:
  ports:
    - port: 80 # Containers port exposed
      targetPort: 80 # Pods port
  selector:
    name: httpddeployment # Apply this service to any pods which have this
specific label
  type: NodePort # Specifies the service type i.e ClusterIP or NodePort
```

3. **LoadBalancer**:
    - A service that distributes network traffic across multiple Pods.
    - Exposes the Service externally using an external (cloud provider's) load balancer.
    - This type of Service is accessible from outside the cluster, mainly via a public IP address.
    - **Use case**: To expose a service to the internet with automatic load balancing across Pods.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector: # tells the controller which pods to watch/belong to
    matchLabels:
      name: httpddeployment
  template:
    metadata:
      name: testpod1
      labels:
        name: httpddeployment
    spec:
      containers:
        - name: c00
          image: httpd
          ports:
            - containerPort: 80
```

```yaml
---
kind: Service # Defines to create Service type Object
apiVersion: v1
metadata:
  name: demoservice
spec:
  ports:
    - port: 80 # Containers port exposed
      targetPort: 80 # Pods port
  selector:
    name: httpddeployment
  type: LoadBalancer
```

4. **ExternalName**:
    - This Service type maps a service to an external DNS name (such as example.com), without exposing any Pods.
    - **Use case**: When you want Kubernetes to forward traffic to an external service by DNS name instead of managing Pods directly.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: example-externalname-service
spec:
  type: ExternalName
  externalName: my.database.example.com
```

- **selector**: This defines which Pods the Service will route traffic to (those with label app=exmaple-app).
- **port**: The port on the Service (80 in this case).
- **targetPort**: The port on the Pods that the Service will forward traffic to (8080 in this case).

Feature	ClusterIP	NodePort	LoadBalancer
Exposition	Exposes the Service on an internal IP in the cluster.	Exposing services to external clients	Exposing services to external clients
Cluster	This type makes the Service only reachable from within the cluster	A NodePort service, each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service.	A LoadBalancer service accessible through a dedicated load balancer, provisioned from the cloud infrastructure Kubernetes is running on
Accessibility	It is **default** service and Internal clients send requests to a stable internal IP address.	The service is accessible at the internal cluster IP-port, and also through a dedicated port on all nodes.	Clients connect to the service through the load balancer's IP.
Yaml Config	`type: ClusterIP`	`type: NodePort`	`type: LoadBalancer`
Port Range	Any public ip form Cluster	30000 - 32767	Any public ip form Cluster

How Services work internally:

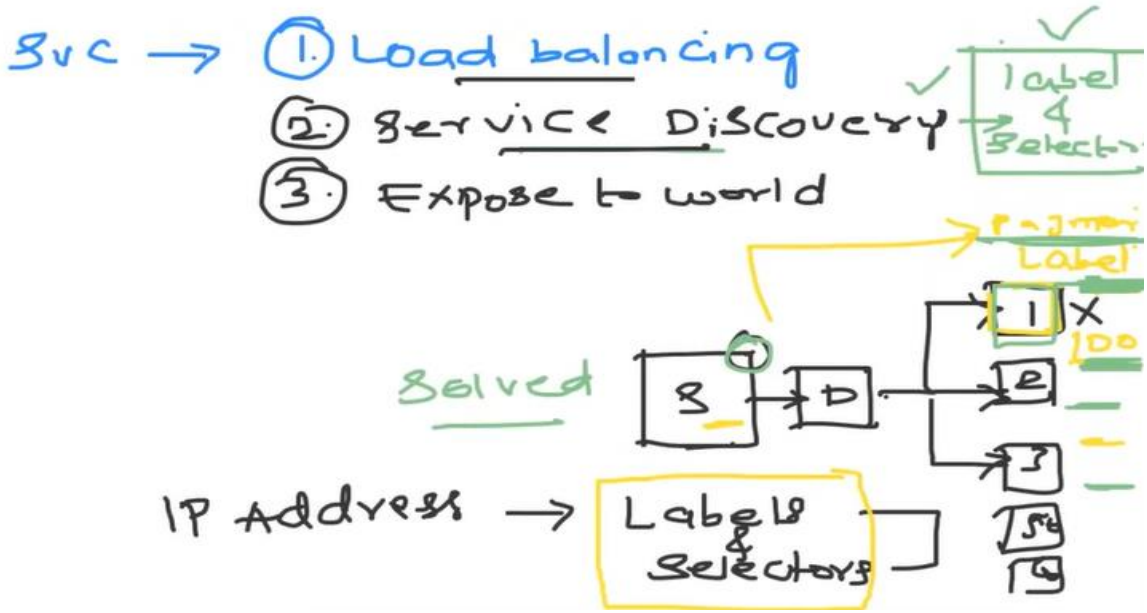
1. **Service Label Selector**: A Service identify the set of Pods using label selectors.
2. **End Points Object**: Kubernetes creates an Endpoints object, which keeps track of the IPs of the Pods selected by the label selector and Kubernetes automatically adjusts the endpoints when Pods are added or removed.
3. **Service Proxying (Routing)**: Kubernetes uses **kube-proxy** to route traffic to the Service and forward it to the appropriate Pod. **kube-proxy** uses methods like **iptables or IPVS for load balancing.**

Key Features of Kubernetes Services:

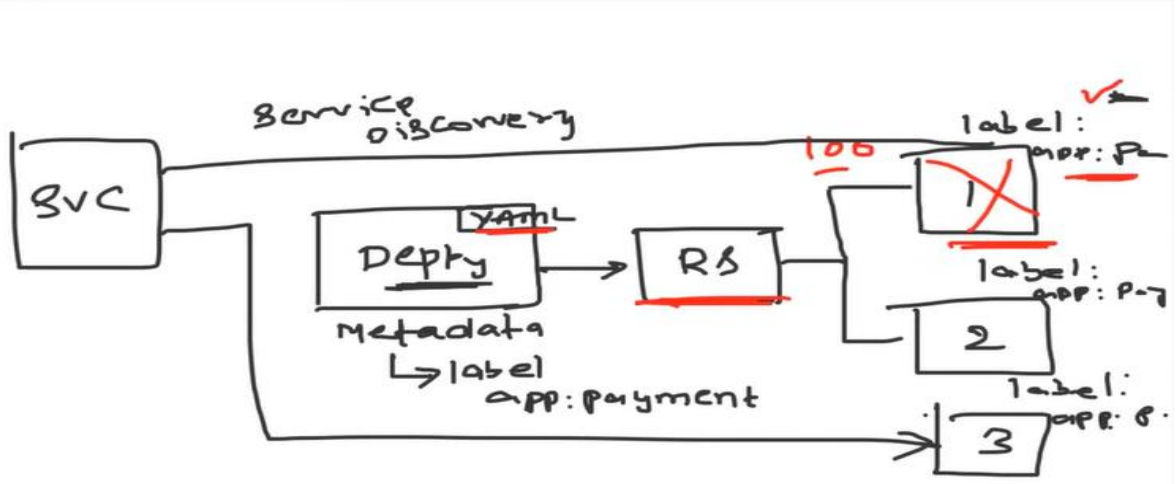
- **Stable Endpoint**: Services provide stable IP addresses and DNS names to access Pods.
- **Discovery and Load Balancing:** Kubernetes Services allow clients to discover and communicate with the right Pods using DNS-based service discovery and label selectors, while automatically distributing traffic across Pods through load-balancing mechanisms.
- **Selector and Labels:** A Service uses selectors to identify which Pods it should target. Pods with labels that match the selector will automatically become part of the Service.

Service responsibilities.

- Load balancing
- Service discovery mechanism (using labels and selectors)
- Expose to the external world.

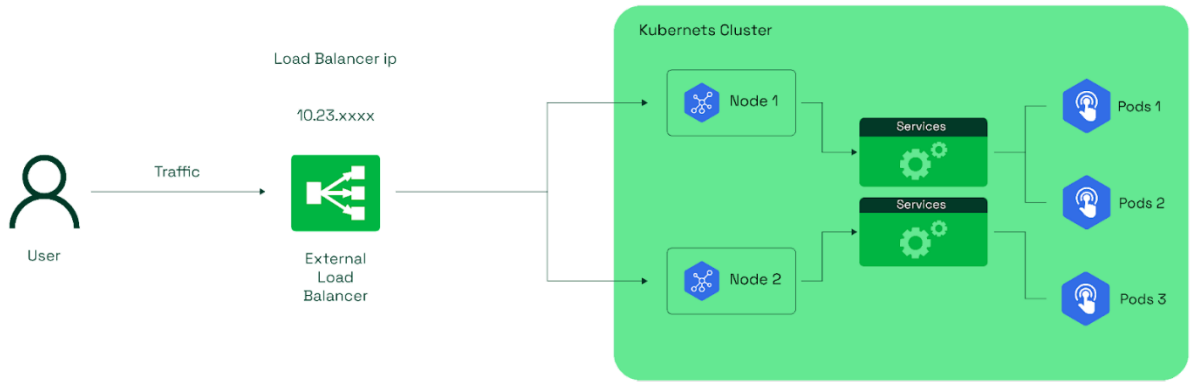


Service discovery



Load Balancer:

- A **LoadBalancer** service **provisions an external load balancer** to **expose applications** to **external traffic** and **distribute incoming network traffic** across multiple pods or nodes within the cluster, ensuring **reliability and scalability**.
- The Load Balancer will **forward the traffic** to the available nodes in the cluster on **the nodePort** assigned to the service
- Kubernetes offers **three types** of **load balancing algorithms** for Services, which distribute traffic based on **round-robin, least connections, or IP hash.**
- Load balancing is an essential part of Kubernetes networking, providing efficient and reliable traffic distribution across a cluster.



Eg:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: httpddeployment
spec:
  replicas: 1
  selector: # tells the controller which pods to watch/belong to
    matchLabels:
      name: httpddeployment
  template:
    metadata:
      name: testpod1
      labels:
        name: httpddeployment
    spec:
      containers:
        - name: c00
          image: httpd
          ports:
            - containerPort: 80
---
kind: Service # Defines to create Service type Object
apiVersion: v1
metadata:
  name: demoservice
```

```yaml
spec:
  ports:
    - port: 80 # Containers port exposed
      targetPort: 80 # Pods port
  selector:
    name: httpddeployment
  type: LoadBalancer
```

LoadBalancer Use-cases in K8s

- Expose your applications to the public internet.
- Traffic distribution across Nodes and Pods

How Load Balancer works in K8s?

- A load balancer service is created, which identifies the cloud platform
- The cloud provider generates a load balancer with a unique IP address
- The load balancer distributes client requests to the appropriate nodes hosting the pods
- Configure LoadBalancer as Service
- Distribute traffic to external IP

Advantages of Load Balancer:

- **Automatic provisioning:** Simplifies exposing services externally.
- **Traffic distribution:** Automatically balances traffic across pods.
- **Managed service:** Especially useful in cloud environments where you don't need to manually configure or maintain the load balancer.

Types of LoadBalancer:

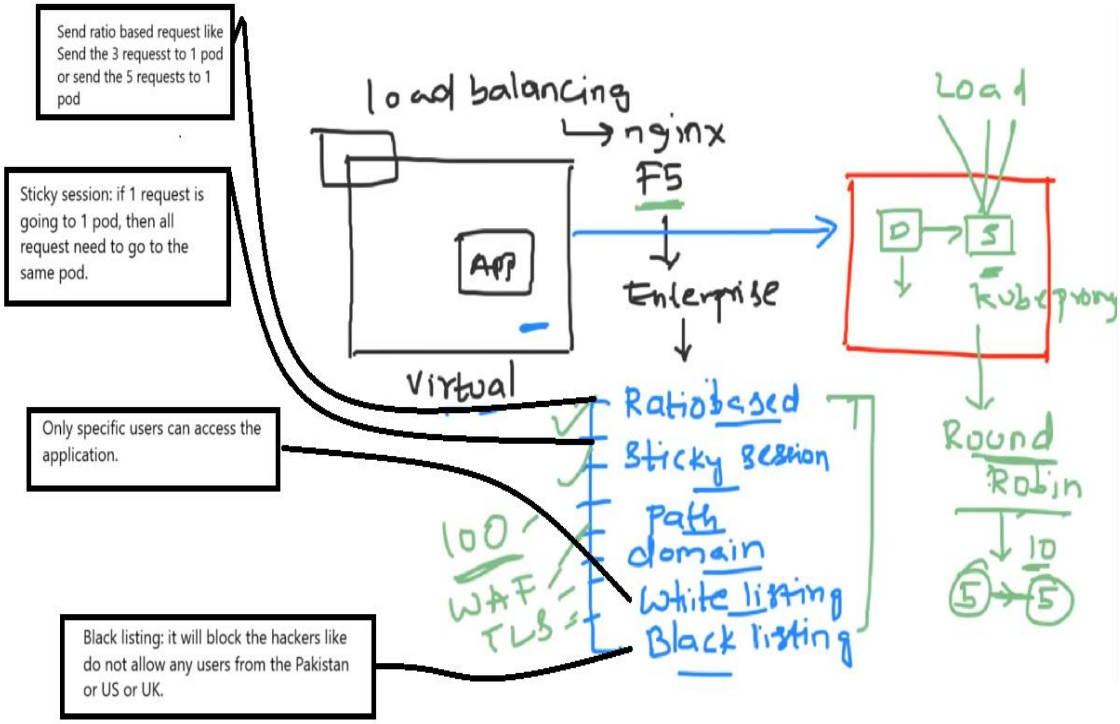
AWS: Elastic Load Balancer (ELB)

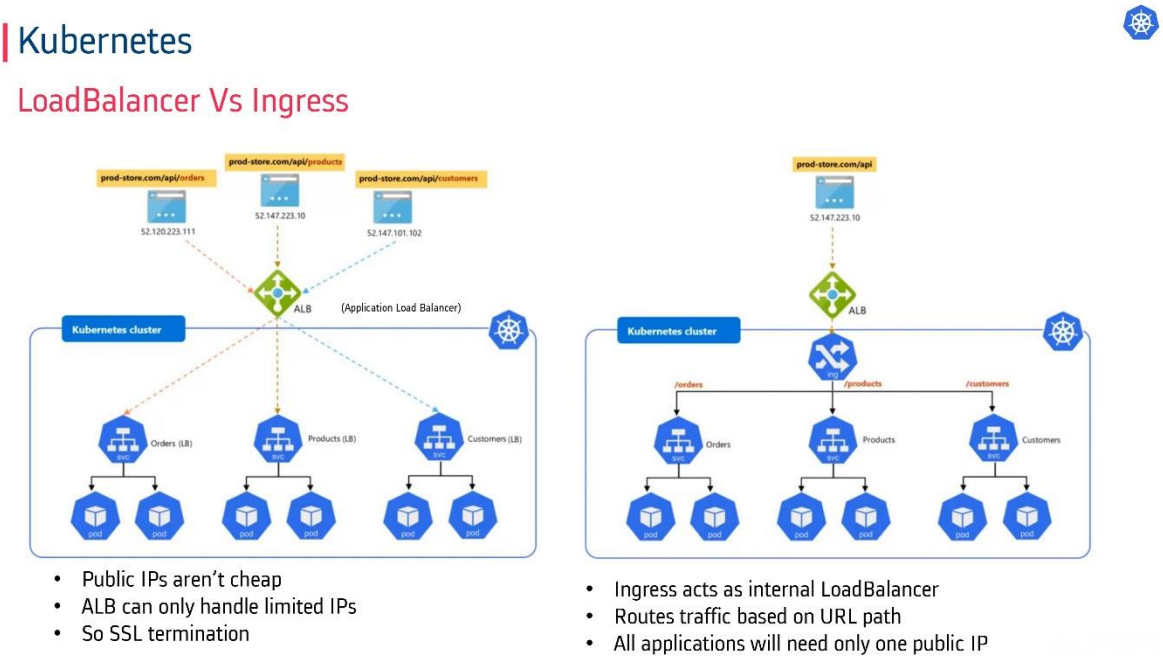
i) Application Load Balancer (ALB)
ii) Network Load Balancer (NLB)
iii) Classic Load Balancer (CLB).

Summary of Key Types:

- **Cloud Provider Managed Load Balancer:** Automatically provisions and manages external load balancers.
- **Internal Load Balancer:** Used for internal traffic within the cluster (cloud environment).
- **MetalLB:** A load balancer solution for bare-metal Kubernetes clusters.
- **Ingress Controller:** Manages routing of HTTP(S) traffic via rules and works with external load balancers.
- **Layer 4 Load Balancer:** Routes traffic based on IP address and port (TCP/UDP).
- **Layer 7 Load Balancer:** Routes traffic based on content of the request (HTTP/S).
- **DNS Load Balancer:** Distributes traffic via DNS resolution, often used for geo-distribution.
- **Global Load Balancer:** Distributes traffic across geographically distributed data centres.

In Virtual machines





Load balancer limitations:

- Limited URL routing
- Cloud provider dependent.
- Lacks SSL termination.

- It doesn't support URL rewriting and rate limiting.
- Enterprise level support and TLS Loadbalancer capabilities.
  (Ratio Based routing, Sticky, path, domain, writing listing and black listing-based routing.)
- Loadbalancer static IP address will be charged by cloud provider. if we created more loadbalancer static IP address

Ingress, Routes and Ingress Controllers:

Load Balancer Type Service

Q: Is Load Balancer service type only restricted to Cloud providers?
A: Bare Metal LB Implementation - https://github.com/metallb/metallb

Q: If Load Balancer service type can do the thing for you, why use an Ingress resource?

A. 1. If you have the large-scale ecommerce application, consider it has 200 to 300 services, if you want to user cloud provider load balancer, then you must create 200 to 300 static external IP addresses, which will cloud provider will charge huge amount. To overcome you can create ingress resource, it will manage all services with one static external IP address.

2. It defines routing for specific service using host base routing and path-based routing, session-based routing, or cookies-based routing etc.

Before 2015, in K8s v1.1, it provides only basic load balancer support. Ingress concept is not there.

Load balancing mechanism is providing simple round robin mechanism, If 10 request come, if you have 2 pods, then round robin mechanism distribute it equal to 2 to pods. 5 requests must go to 1 pod and another request must go to another pod.

Ingress:

What is a Kubernetes Ingress?

- **Kubernetes Ingress** is an **API object** that manages external access to services within a cluster. It is primarily used to configure and manage HTTP and HTTPS routing to services running inside the cluster.
- Ingress **exposes HTTP and HTTPS endpoints** to external users and **routes incoming traffic to internal services** based on **rules defined** in the **Ingress resource**.
- **Traffic routing** is controlled by **rules defined** in the **Ingress resource** based on hostnames and paths. **(**Specifying how external traffic should be directed to internal services within the Kubernetes cluster.**)**
- **The ingress controller** is responsible for routing HTTP(S) traffic based on defined rules
- **Ingress Controllers** mainly **work alongside a load balancer** (often a cloud-managed one or MetalLB) to handle HTTP(S) traffic.
- Once the **Ingress controller** is installed on a Kubernetes cluster, it continuously watches for Ingress resources (like nginx.conf, where all the routing details are updated) within the cluster. Along with the Ingress controller, NGINX applications are also deployed.

- Compared to **NodePort** or **LoadBalancer services**, **Ingress** offers advanced features like **path-based** or **host-based routing**, providing more flexible traffic management.
- Ingress allows exposing multiple services using a single external IP or domain, combining features of both a LoadBalancer and an API Gateway.

Key Features:

- Used for HTTP(S) traffic routing, mainly for web applications.
- Works well with SSL termination (decrypting) and path-based routing.
- Involves more advanced routing, including features like URL rewriting, traffic splitting, and authentication.

How ingress works?

- **The ingress controller** is responsible for routing traffic based on defined rules.
- The **Ingress Controller** routes incoming requests based on URL paths or hostnames (e.g., /app1, api.example.com). It often sits behind a load balancer to distribute the traffic across multiple Pods or services.
- Manages routing of HTTP(S) traffic via rules and works with external load balancers.
- NGINX Ingress Controller, Traefik, and AWS ALB Ingress

Examples of Ingress Advantages and Use Cases:

- **Path-based routing**: Routes requests to different services based on the URL path. For example:
  - example.com/api goes to the api-service.
  - example.com/web goes to the web-service.
- **Host-based routing**: Routes traffic based on the requested domain name (hostname). For example:
  - app1.example.com routes to service1.
  - app2.example.com routes to service2.
- **SSL/TLS termination**: Handles HTTPS traffic, terminating (decrypting) SSL connections, reducing the burden on backend services. You can configure Ingress to terminate SSL and forward traffic as HTTP. SSL termination
- **Why SSL Termination at LoadBalancer?**

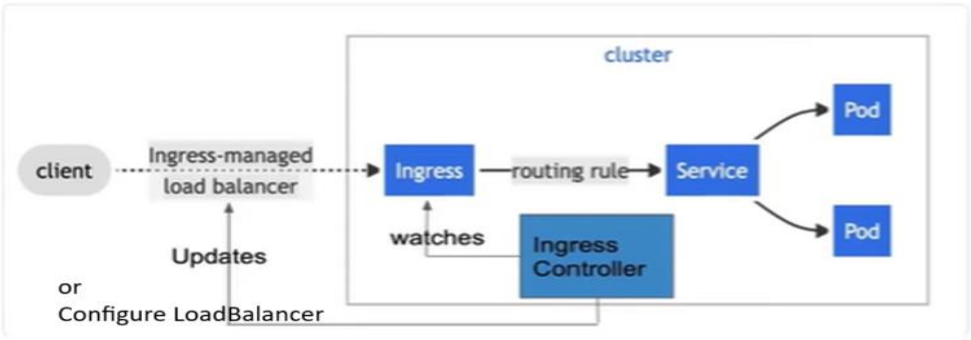
- **Traffic Load balancing**: Distributes incoming traffic across multiple instances (replicas) of a service for high availability and better resource utilization.
- **Rate Limiting and Security**: Limiting the number of requests or enforcing authentication at the Ingress level to prevent abuse or ensure secure access.

Why do we use Ingress?

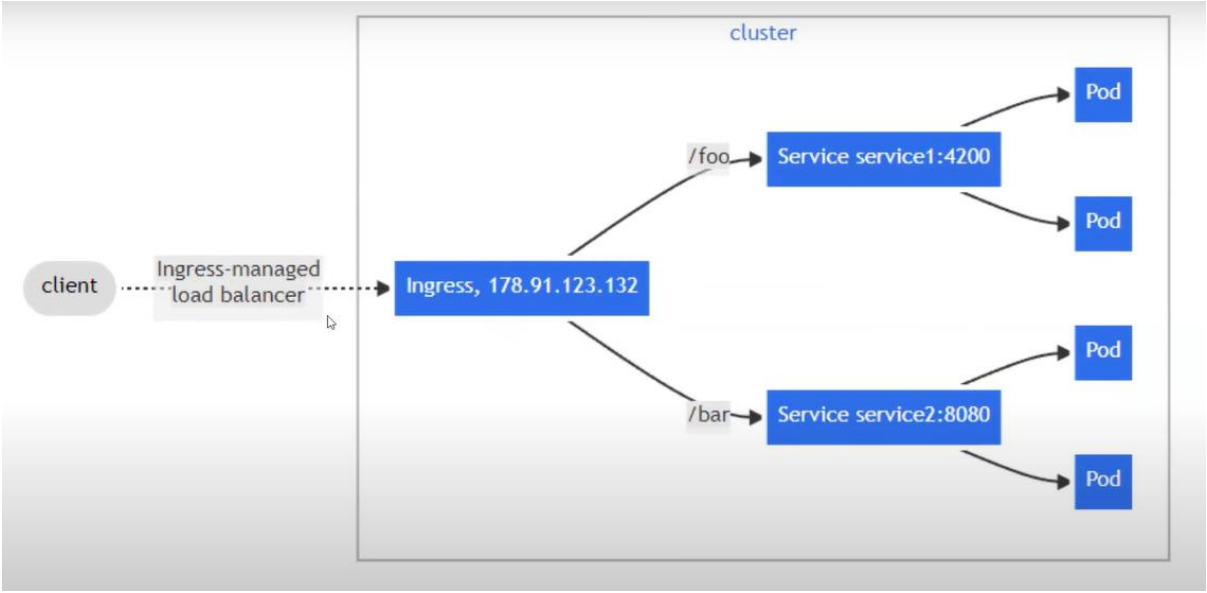
- Expose multiple microservices with domains and subdomains
- Centralized routing
- So ingress is best compared to LoadBalancer and NodePort

Ingress Controller

In order for the Ingress resource to work, the cluster must have an ingress controller running.



.




```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: nginx.you_url.com
    http:
      paths:
      - path: /nginx
        pathType: Prefix
```

```yaml
      backend:
        service:
          name: nginx-loadbalancer-service
          port:
            number: 8080
            name: http
```

Sample Ingress

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  defaultBackend:
    service:
      name: test
      port:
        number: 80
```

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-no-auth
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: http-svc
            port:
              number: 80
```

Host Based Routing

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-with-auth
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: http-svc
            port:
              number: 80
  - host: example.bar.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: meow-svc
            port:
              number: 80
```

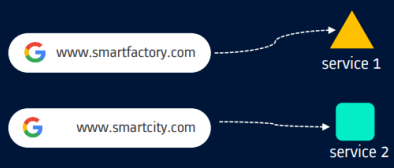
Path Based Routing

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-with-auth
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /first
        pathType: Prefix
        backend:
          service:
            name: http-svc
            port:
              number: 80
      paths:
      - path: /second
        pathType: Prefix
        backend:
          service:
            name: meow-svc
            port:
              number: 80
```

Kubernetes

Nginx Ingress Controller

- Ingress-nginx is an Ingress controller for Kubernetes using NGINX as a reverse proxy and load balancer
- Officially maintained by Kubernetes community
- Routes requests to services based on the request host or path, centralizing a number of services into a single entrypoint.
  Ex: www.mysite.com or www.mysite.com/stats



Deploy Nginx Ingress Controller

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-0.32.0/deploy/static/provider/baremetal/deploy.yaml

https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md#bare-metal

Kubernetes

Ingress Rules

ingress-rules.yml

Path based routing

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-rules
spec:
  rules:
  - host:
    http:
      paths:
      - path: /nginx
        backend:
          serviceName: nginx-service
          servicePort: 80
      - path: /flask
        backend:
          serviceName: flask-service
          servicePort: 80
```

Host based routing

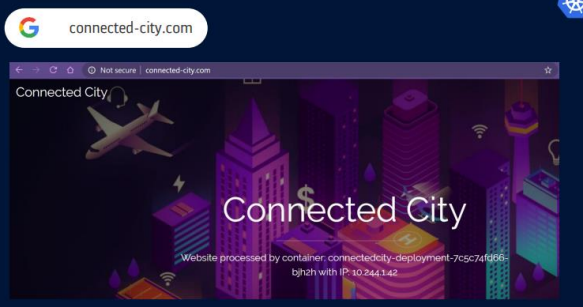
```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-rules
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: nginx-app.com
    http:
      paths:
      - backend:
          serviceName: nginx-service
          servicePort: 80
  - host: flask-app.com
    http:
      paths:
      - backend:
          serviceName: flask-service
          servicePort: 80
```

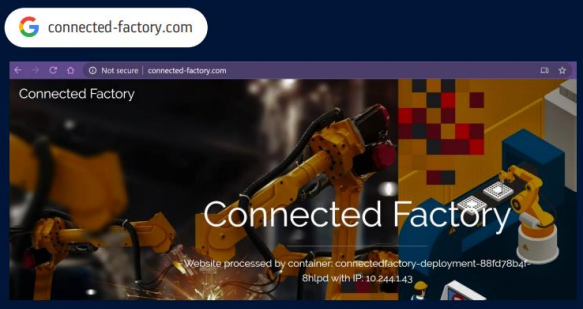
Ingress-controller executes these ingress-rules by comparing with the http requested URL in the http header

Kubernetes

Demo: Ingress

- 3VMs K8s Cluster + 1 VM for Reverse Proxy
- Deploy Ingress controller
- Deploy pods
- Deploy services
- Deploy Ingress rules
- Configure external reverse proxy
- Update DNS names
- Access applications using URLs
  - connected-city.com
  - connected-factory.com

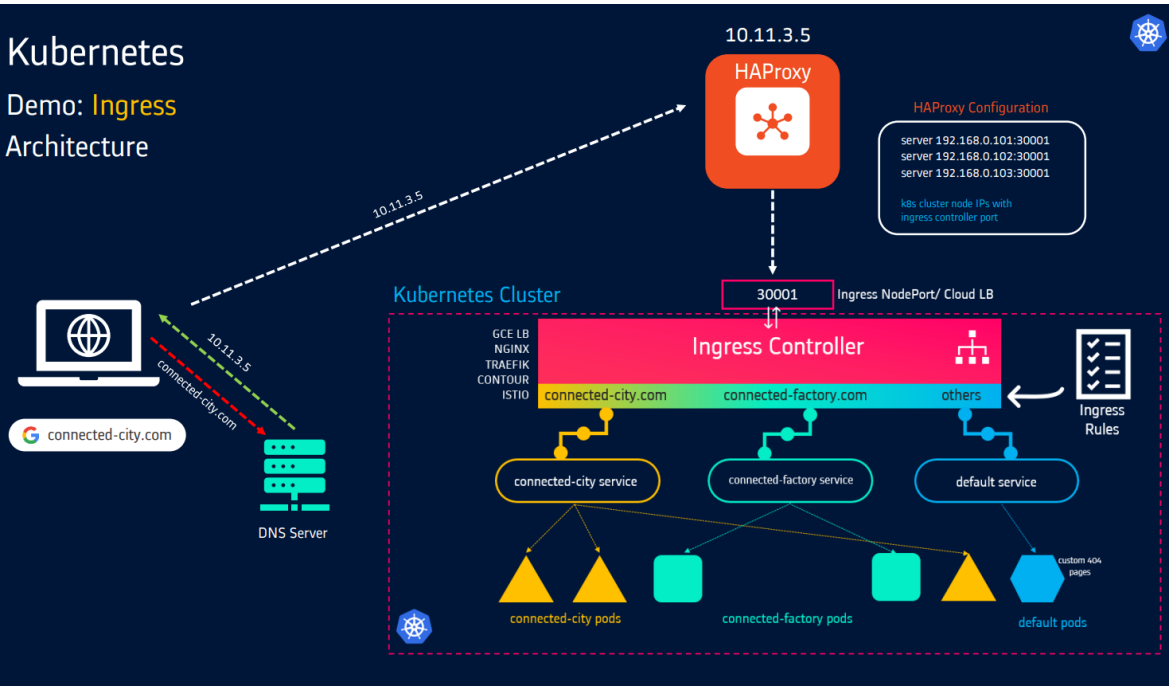




Kubernetes

Demo: Ingress

Architecture



Kubernetes

Demo: Ingress

1. Deploy Nginx Ingress Controller

kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-0.32.0/deploy/static/provider/baremetal/deploy.yaml

2. Deploy pods and services

kubectl apply –f <object>.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: connectedcity-service
spec:
  ports:
  - port: 80
    targetPort: 5000
  selector:
    app: connectedcity
```

Application-1
Deployment + ClusterIP service

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: connectedcity-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: connectedcity
  template:
    metadata:
      labels:
        app: connectedcity
    spec:
      containers:
      - name: connectedcity
        image: kunchalavikram/connectedcity:v1
        ports:
        - containerPort: 5000
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: connectedfactory-service
spec:
  ports:
  - port: 80
    targetPort: 5000
  selector:
    app: connectedfactory
```

Application-2
Deployment + ClusterIP service

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: connectedfactory-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: connectedfactory
  template:
    metadata:
      labels:
        app: connectedfactory
    spec:
      containers:
      - name: connectedfactory
        image: kunchalavikram/connectedfactory:v1
        ports:
        - containerPort: 5000
```

Kubernetes

Demo: Ingress

3. Deploy ingress rules manifest file

- Host based routing rules
- Connects to various services depending upon the host parameter

kubectl apply –f <object>.yml

```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: ingress-rules
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: connected-city.com
    http:
      paths:
      - backend:
          serviceName: connectedcity-service
          servicePort: 80
  - host: connected-factory.com
    http:
      paths:
      - backend:
          serviceName: connectedfactory-service
          servicePort: 80
```

Kubernetes

Demo: Ingress

4. Deploy HA Proxy LoadBalancer

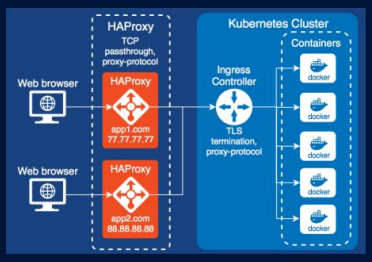
- Provision a VM
- Install HAProxy using package manager
  - apt install haproxy –y
- Restart HAProxy service after modifying the configuration
  - systemctl stop haproxy
  - add configuration to /etc/haproxy/haproxy.cfg
  - systemctl start haproxy && systemctl enable haproxy

10.11.3.5

/etc/haproxy/haproxy.cfg

```
# Configure HAProxy to listen on port 80
frontend http_front
   bind *:80
   default_backend http_back

# Configure HAProxy to route requests to swarm nodes on port 8080
backend http_back
   balance roundrobin
   mode http
   server srv1 192.168.0.101:32174
   server srv2 192.168.0.102:32174
   server srv3 192.168.0.103:32174
root@proxyserver:/home/osboxes#
```



Kubernetes

Demo: Ingress

5. Update dummy DNS entries

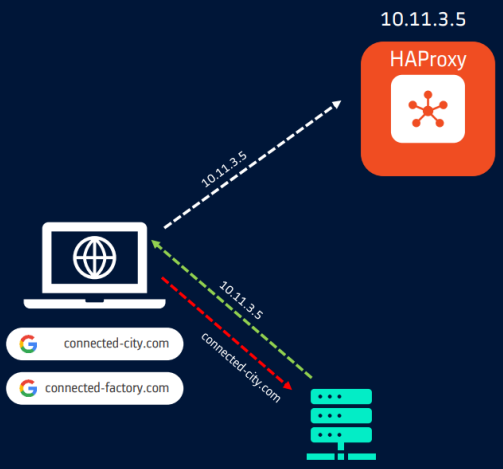
Both DNS names to point to IP of HAProxy server

windows

```
C:\Windows\System32\drivers\etc\hosts
192.168.0.105 connected-city.com
192.168.0.105 connected-factory.com
ipconfig /flushdns
```

linux

```
/etc/hosts
192.168.0.105 flask-app.com
```



Kubernetes

Demo: Ingress

6. Access Application through URLs

Ingress vs. LoadBalancer Services

- **Ingress**:
  - Provides a single-entry point for HTTP and HTTPS traffic.
  - Offers flexible routing based on hostnames, paths, and headers.
  - Can handle SSL/TLS termination.
  - Requires an Ingress controller to implement the Ingress resource.
- **LoadBalancer Service**:
  - Directly exposes a service to the internet using a cloud provider's load balancer.
  - Simpler to set up but less flexible in terms of routing rules.
  - Mainly used for non-HTTP/HTTPS traffic, like TCP and UDP.

Common Ingress Controllers:

1. **NGINX Ingress Controller**: One of the most widely used Ingress controllers, offering robust features like load balancing, SSL termination and path-based routing.
2. **Traefik**: A dynamic and modern reverse proxy, Traefik is highly adaptable and integrates seamlessly with Kubernetes.
3. **HAProxy Ingress**: A reliable and high-performance option, HAProxy supports advanced load balancing features.
4. **Istio**: Primarily a service mesh, Istio can also serve as an Ingress Controller with advanced routing, security, and observability features.
5. **Envoy**: A high-performance proxy used for microservices architectures, integrated with Kubernetes as an Ingress Controller for advanced traffic management.

Troubleshooting Common Ingress Configuration Errors:

1. **Ingress Not Routing Traffic Correctly**:
   - **Cause**: Incorrect path or host definitions.
   - **Solution**: Verify the path and host rules in the Ingress definition. Check for typos or mistakes in URL matching and ensure services are correctly defined under the rules.
2. **503 Service Unavailable Error**:
   - **Cause**: The Ingress controller cannot reach the backend service (often due to misconfigured service names or ports).
   - **Solution**: Check that the service specified in the Ingress resource exists, is exposed, and is properly configured with the correct port. Use **kubectl describe ingress** to check for errors related to backend services.
3. **SSL/TLS Errors (e.g., Certificate Mismatch)**:
   - **Cause**: A mismatch between the SSL/TLS certificate in the Ingress controller and the domain or expired certificates.
   - **Solution**: Ensure that the **TLS secret** associated with your Ingress has a valid certificate for the domain in question. Use **Cert-Manager** to automate certificate renewal.
4. **404 Not Found**:
   - **Cause**: The path or hostname in the request doesn't match any defined Ingress rules.

    - **Solution**: Verify that the Ingress paths and hostnames are correctly defined and that they match the incoming request. Check for correct service names and that the Ingress rules are pointing to the correct backend services.
5. **Incorrect Annotations or Configuration Settings**:
    - **Cause**: Misconfigured annotations or settings in the Ingress definition (e.g., missing or incorrect load balancing settings).
    - **Solution**: Review and validate the Ingress resource annotations. For example, NGINX has specific annotations like **nginx.ingress.kubernetes.io/rewrite-target** that must be configured correctly.
6. **Ingress Controller Not Running or Misconfigured**:
    - **Cause**: The Ingress controller might not be deployed or properly configured.
    - **Solution**: Ensure the Ingress controller pod is running by checking with kubectl get pods -n <namespace>. Check logs using kubectl logs <ingress-controller-pod-name> to look for errors.
7. **Timeout or Slow Response**:
    - **Cause**: The backend service might be slow or misconfigured, causing timeouts.
    - **Solution**: Investigate the backend service logs to ensure it is responsive. Increase timeout settings in the Ingress controller if necessary.
8. **Inconsistent Ingress Controller and Service Ports**:
    - **Cause**: Mismatch between the port configured on the Ingress and the port exposed by the service.
    - **Solution**: Ensure that the service port specified in the Ingress matches the actual service port.
9. **Check Logs**: Review the logs of the Ingress controller to identify any errors or misconfigurations.
10. **Validate Ingress Resources**: Ensure that Ingress resources are correctly defined and that annotations are valid.
11. **Inspect ConfigMaps**: Verify that ConfigMap keys and values are correctly configured.
12. **Health Checks**: Ensure that backend services are healthy and responding as expected.
13. **DNS Resolution**: Confirm that DNS resolution is working correctly and that the Ingress controller can resolve service names.
14. **Check Events**: Review Kubernetes events related to Ingress resources to identify any issues.

TLS:

TLS in Kubernetes: Secure Communication for Cluster Components

- **TLS (Transport Layer Security)** in Kubernetes is used to secure communication between components within the cluster and between external clients and Kubernetes services.
- TLS ensures that **data is encrypted**, **preventing unauthorized access** and **protects sensitive data.**

Where TLS is Used in Kubernetes?

- 🔹 **API Server –** TLS secures communication between the Kubernetes API server and clients (kubectl, controllers, etc.).
- 🔹 **etcd Database –** Data stored in etcd is encrypted using TLS to prevent unauthorized access.
- 🔹 **Kubelet & API Server –** TLS ensures **mutual authentication** between nodes and the control plane.
- 🔹 **Ingress Controller –** TLS enables HTTPS traffic for applications exposed externally.
- 🔹 **Service-to-Service Communication –** Applications within the cluster use TLS certificates for **secure pod-to-pod communication.**

TLS Certificates in Kubernetes

- ✅ **Self-Signed Certificates –** Default certificates generated for internal communication within Kubernetes.
- ✅ **Cert-Manager –** A Kubernetes tool that automates certificate issuance and renewal for TLS.
- ✅ **Let's Encrypt –** A popular service for issuing free TLS certificates for Kubernetes applications.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-secure-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  tls:
  - hosts:
      - example.com
    secretName: tls-secret
  rules:
  - host: example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 443
```

✅ **This Ingress configuration enables HTTPS using a TLS secret (tls-secret).**

Understanding SSL Passthrough, SSL Offloading, and SSL Bridging

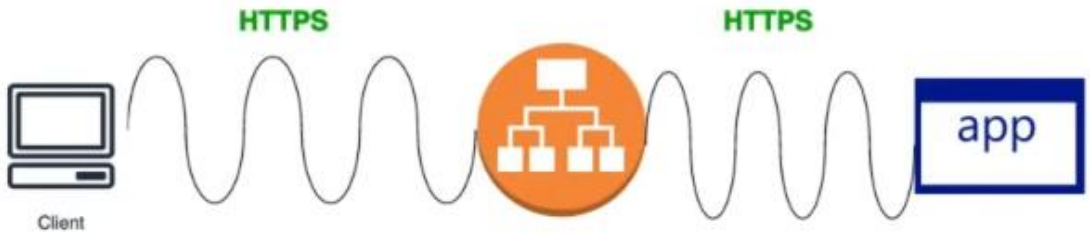
These are techniques used to handle **SSL/TLS encryption** in web applications, particularly within **load balancers, proxies, and Kubernetes ingress controllers**.

In Kubernetes, **SSL/TLS termination** can be handled in different ways depending on security and performance requirements. Here's a breakdown of **SSL Passthrough, SSL Offloading (Termination), and SSL Bridging (Re-Encryption)**, along with their use cases in Kubernetes (especially with Ingress Controllers like Nginx, Traefik, or HAProxy).

1. SSL Passthrough

- **Definition:** The load balancer (**Ingress Controller) forwards encrypted HTTPS traffic** directly to the backend server **without decrypting it**.
- The backend service (e.g., a Pod running a web server) is responsible for **SSL termination** (handles decryption and serves the request).
- **Use case:** Best for **security-sensitive applications**, since the encryption remains **intact end-to-end**.

Pros	Cons
End-to-end encryption	Backend must handle TLS (decryption)
No decryption at Ingress	Harder to debug (traffic stays encrypted)
Good for compliance	Higher workload on backend



Load Balancer capabilities are merely used.
Attacker can pass hacking codes in the traffic and will be directly passed to the backend server.
SSL Passthrough is also a costly process. Might require more CPU.

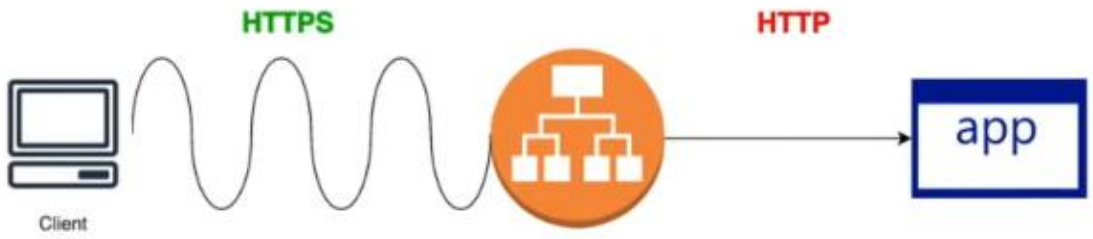
```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"  # Enables passthrough
spec:
  tls:
  - hosts:
    - myapp.example.com  # Client-facing domain
  rules:
```

```
  - host: myapp.example.com
    http:
      paths:
      - path: /
        backend:
          service:
            name: my-service
            port:
              number: 443  # Backend must handles HTTPS
```

2. SSL Offloading

- **Definition:** The load balancer (**Ingress Controller) decrypts all HTTPS traffic** and forwards it as **plain HTTP** to the backend servers.
- The load balancer handles **SSL termination** to improve backend performance.
- **Use case:** AWS Application Load Balancer (**ALB**) uses **SSL Offloading** to handle HTTPS requests efficiently. (**Layer 7 routing**)

Pros	Cons
Reduces backend workload	Traffic is unencrypted between Ingress and Pods
Easier to debug	Requires network policies to secure internal traffic
Supports advanced routing	Not compliant for strict end-to-end encryption



Vulnerable to data theft, man-in-the-middle attacks.
Faster

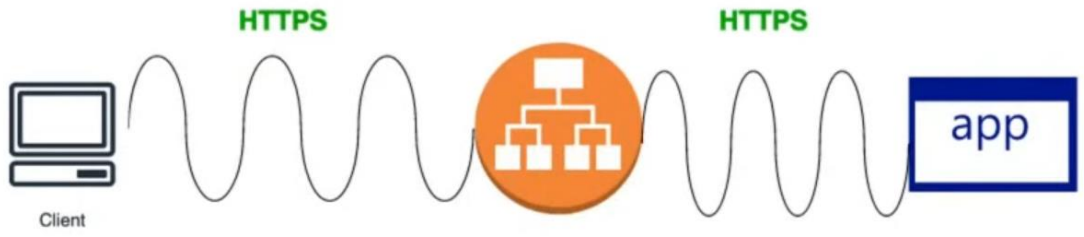
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ssl-offload-ingress
spec:
  tls:
  - hosts:
```

```
    - myapp.example.com
    secretName: my-tls-secret  # Cert stored in a Secret
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        backend:
          service:
            name: my-service
            port:
              number: 80  # Backend receives HTTP
```

3. SSL Bridging

- **Definition:** The load balancer (**Ingress Controller) decrypts TLS traffic** (like SSL Offloading) but then **re-encrypts it** before sending it to the backend.
- **How it works:** Maintains **end-to-end encryption** while allowing traffic inspection.
- **Use case:** When **internal traffic must also be encrypted** (e.g., multi-tenant clusters).

Pros	Cons
Internal traffic stays encrypted	Slightly higher latency
Allows L7 inspection	More complex setup
Good for compliance	Needs backend TLS certs



E2E encrypted and validated for malware attacks by Load Balancer.
More secure more costly in processing as server has to decrypt the traffic.

```
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: ssl-bridge-route
spec:
  entryPoints:
    - websecure
```

```yaml
routes:
- match: Host(`myapp.example.com`)
  services:
  - name: my-service
    port: 443
    scheme: https  # Forces HTTPS to backend
    tls: {}       # Enables re-encryption
tls:
  secretName: my-tls-secret  # Cert for client-side
```

Pros and Cons !!

SSL Passthrough	SSL-Offloading/Termination	SSL-Bridge/Re-encrypt
Costly in processing for Server	Fast in processing	Costly in processing for Server
Secure in many cases	Insecure	Secure
L4 (TCP) Load Balancing	L7 Load Balancing	L7 Balancing
Choose when you don't bother about access rules, blocking, cookie e.t.c.,.	When you want less latency and can compromise on security.	When you need security and advanced load balancer capabilities.
No Load Balancer Inspection.	Load Balancer Inspects the packets.	Load Balancer Inspects the packets.
Recommended*	Highly unrecommended.	Recommended

OpenShift Routes

SSL Offloading == Edge Termination

SSL Bridge == Re-encrypt Termination

SSL Passthrough == Passthrough Termination

```yaml
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: hello-openshift
spec:
  host: hello-openshift-hello-openshift.
  port:
    targetPort: 8080
  to:
    kind: Service
    name: hello-openshift
```

Secure Routes

*Routes does not support storing the TLS certs in secrets - Issue

Routes are simple, you cannot add multiple services, paths or hosts in a single route.

Service mesh

Service mesh
It will help to manage the traffic between services in cluster. Mainly (East - West)

Why service mesh needs?

- A **service mesh** enhances **service-to-service communication** within a cluster and supports features like **mTLS** (Mutual TLS) for secure communication.
- It will also add the advanced capabilities such as **deployment strategies** (Canary / A-B / Blue Green)
- **Observabilities** – Kiali – it will keeps a track of service to service communication information. It will helps to understand how services are behaving and metrics health of services.
- **Circuit breaking** – it can helps to split traffic splitting

In **Kubernetes**, a **Service Mesh** is an infrastructure layer that manages **service-to-service communication** within a cluster. It provides advanced features like **traffic control, observability, security**, and **reliability** without requiring changes to your application code.

🔧 **What Does a Service Mesh Do?**

1. **Traffic Management**
    - Controls how requests are routed between services.
    - Supports features like retries, timeouts, and circuit breakers.
2. **Security**

- Implements **mTLS (Mutual TLS)** to encrypt traffic and authenticate services.
- Ensures secure communication between microservices.

3. **Observability**
    - Provides metrics, logs, and tracing for service interactions.
    - Helps monitor performance and troubleshoot issues.
4. **Policy Enforcement**
    - Allows defining access control and rate-limiting policies.

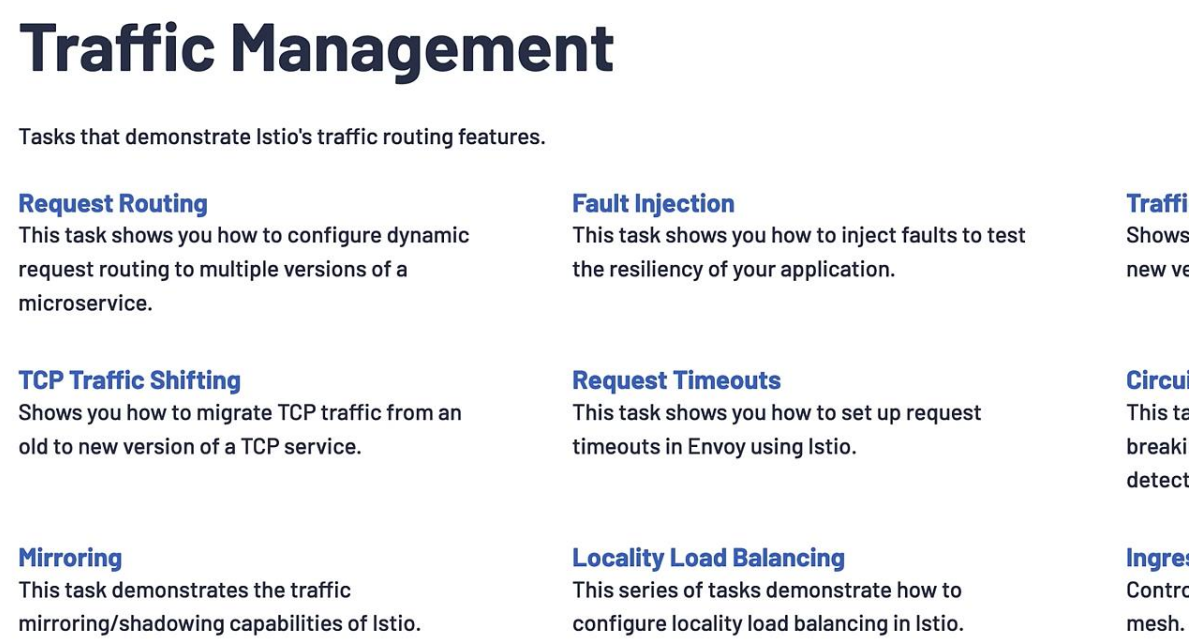
🧱 How It Works in Kubernetes

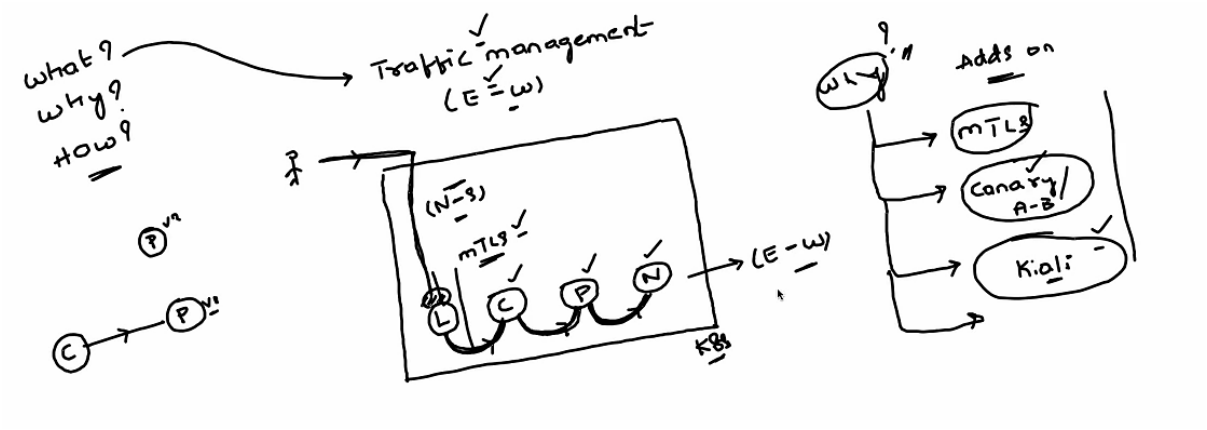
A service mesh typically uses **sidecar proxies** (like Envoy) that are injected into each pod. These proxies handle all incoming and outgoing traffic for the service, enabling the mesh to manage communication transparently.

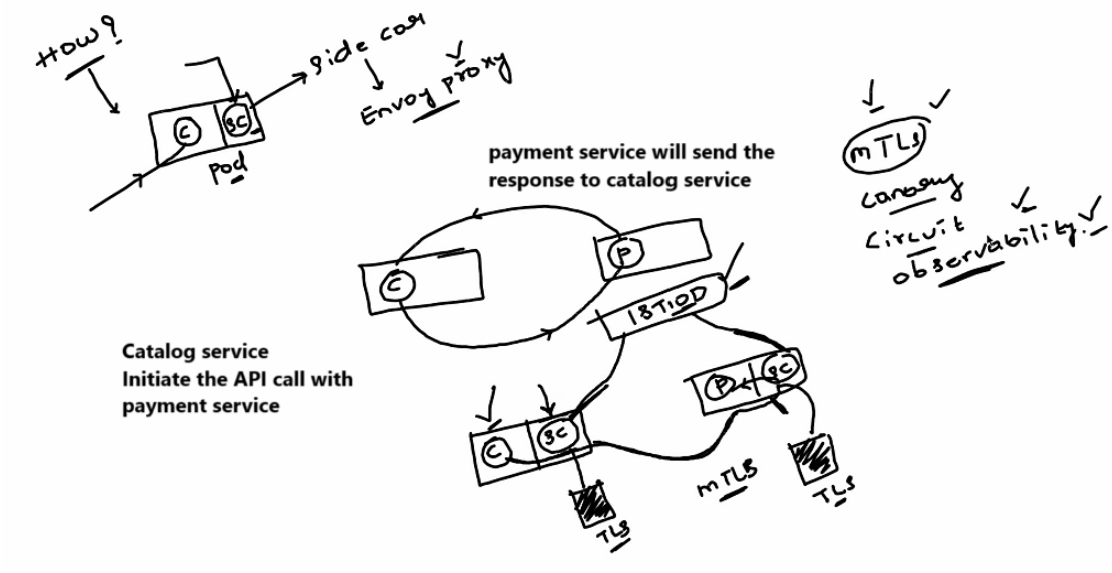
📌 Example Use Case

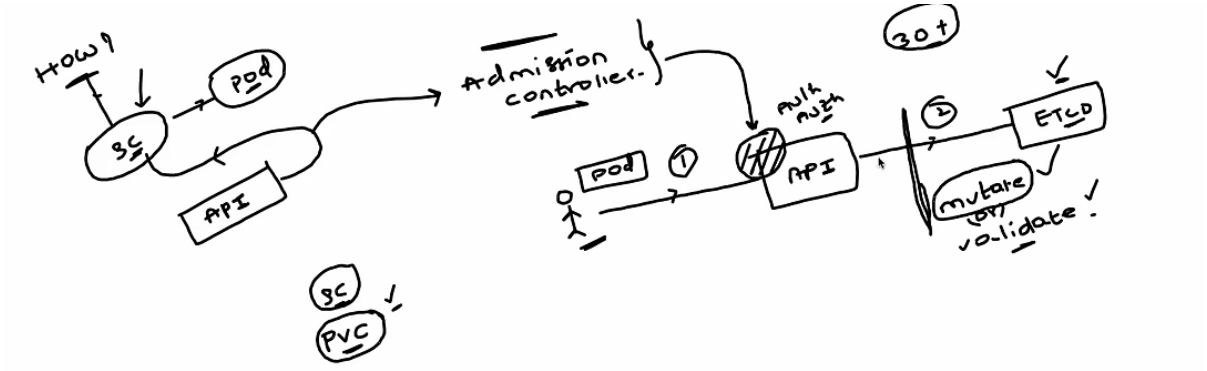
Imagine you have three microservices: **frontend**, **backend**, and **database**. A service mesh can:

- Encrypt traffic between them using mTLS.
- Automatically retry failed requests.
- Monitor latency and error rates.
- Apply policies like "only frontend can talk to backend."









Admission controller
It will validate and mutate the authenticated and authenticated user request

- Mutate
- Validate

📘 8. Summary Table

Feature	Purpose
Pod IPs	Every Pod has a unique IP
Services	Stable IP to access Pod groups
DNS	Name-based access to services
Ingress	Expose HTTP/HTTPS services
CNI Plugin	Manages actual networking setup
Network Policy	Controls traffic between Pods
Service Types	ClusterIP, NodePort, LoadBalancer, ExternalName

3) Replicaset

ReplicaSets:

- **A ReplicaSet** is a Kubernetes object that ensures that the exact number of pods(replicas) are always running in the cluster by replacing any failed pods with new ones. It's primarily used to maintain **high availability** and **scalability** of applications.
- The replica count is controlled by the replicas field in the resource definition file
- **Replicaset** uses **set-based selectors** whereas **replicacontroller** uses **equality-based selectors**

🔁 What Does a ReplicaSet Do?

- **Ensures desired state**: If a pod crashes or is deleted, the ReplicaSet automatically creates a new one to maintain the desired number.
- **Supports rolling updates**: Often used behind the scenes by **Deployments** to manage updates.
- **Scales applications**: You can increase or decrease the number of replicas easily.

🧱 Key Components

- **replicas**: Number of pod copies you want running.
- **selector**: Labels used to identify which pods the ReplicaSet should manage.
- **template**: The pod specification (like image, ports, etc.).

ReplicationController:

A **ReplicationController** is an older Kubernetes resource with a similar purpose to ReplicaSets. It ensures that a specified number of pod replicas are running at all times.

Differences between ReplicaSet and ReplicationController:

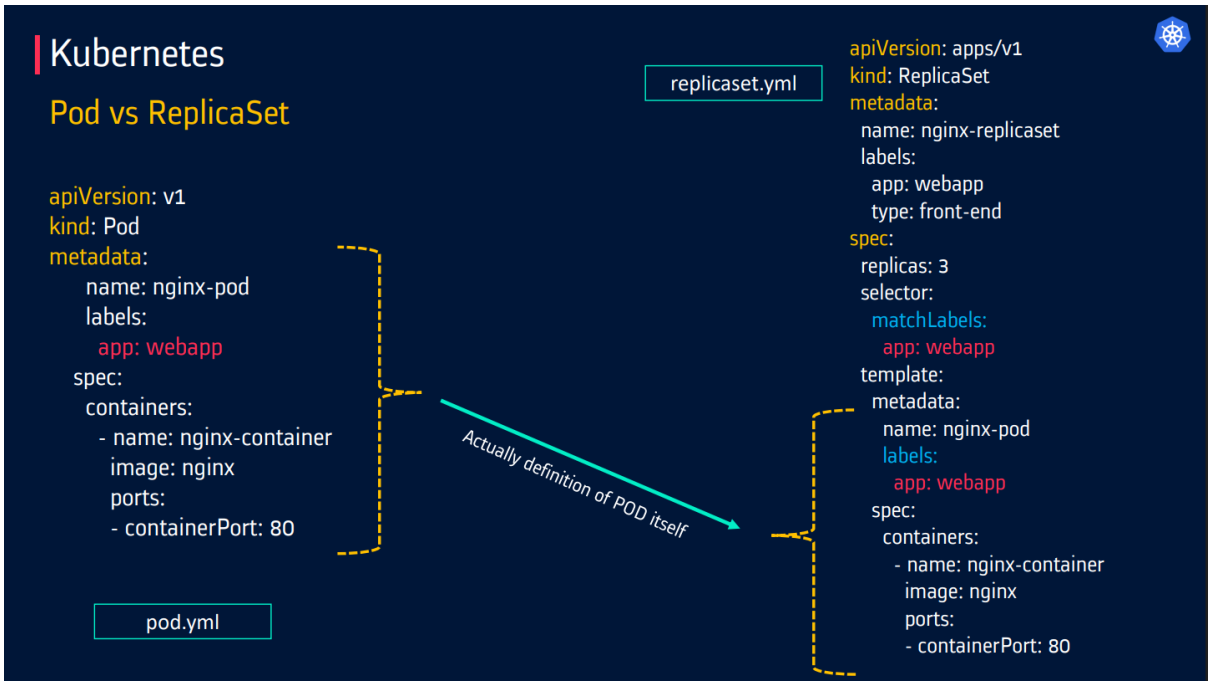
➢ **Replicaset:**

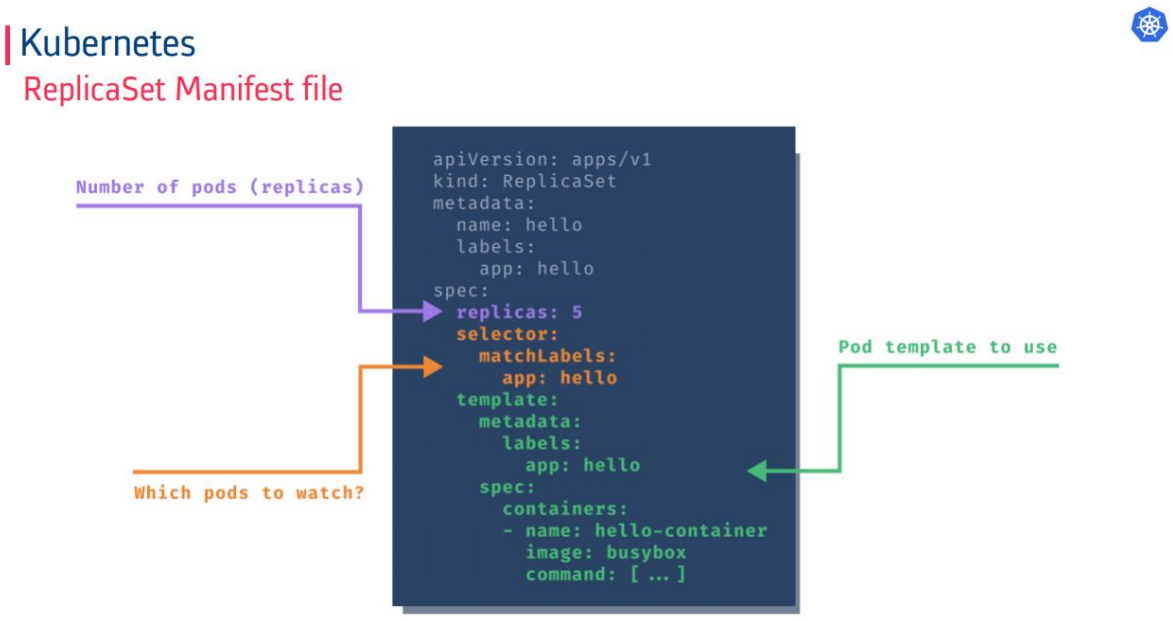
- **Selectors:** Supports set-based selectors (more flexible).
- **Use Case:** Used with Deployments for modern applications.
- **Efficiency:** More advanced and flexible.

➢ ReplicationController

- **Selectors:** Supports only equality-based selectors.
- **Use Case:** Considered legacy, replaced by ReplicaSet.
- **Efficiency:** Limited to basic replication tasks.

Example:





Example:

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3  # modify replicas according to your case
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```

4) Deployments

Deployments:

- A **Deployment** is a Kubernetes object used to manage Pods and ReplicaSets through a declarative configuration. It defines the desired state of an application including the number of Pods, deployment strategies, container images and manages the rollout and rollback of changes.
- The **Deployment Controller** ensures the actual state matches the desired state by replacing failed pods.
- Deployments support several **deployment strategies** like "**recreate**" and "**rolling update**" and can be customized for advanced methods such as blue/green or canary deployments.
- A **Deployment** provides replication functionality and auto-healing and auto-scaling features with the help of ReplicaSets.
- It also facilitates rolling out and rolling back changes effectively.

🚀 **What is a Deployment?**

A **Deployment** ensures that a specified number of **pods** are running and up-to-date. It automatically manages the creation and replacement of pods using **ReplicaSets**, and it allows you to:

- **Roll out updates** to your application with zero downtime.
- **Roll back** to a previous version if something goes wrong.

- **Scale** your application up or down easily.
- **Monitor** the status of your application.

🧱 Key Features

Feature	Description
Rolling Updates	Gradually replaces old pods with new ones.
Rollback	Revert to a previous version if needed.
Declarative Updates	You declare the desired state, and Kubernetes makes it happen.
Self-healing	Automatically replaces failed or deleted pods.

Key Features of Deployments

1. **Declarative Management:** You describe the desired state in a YAML or JSON file. Kubernetes takes care of the changes.
2. **Rolling Updates:** Updates occur incrementally, ensuring zero downtime.
3. **Rollback Capability:** Kubernetes can revert to previous versions if an update fails.
4. **Version History:** Deployments maintain a history of ReplicaSets, enabling easier rollbacks.
5. **Scaling:** Deployments manage the number of replicas dynamically.

Steps to Manage Application Updates

1. **Define the Deployment:** A Deployment describes the application's desired state, including the image version, number of replicas, and update strategy.
2. **Apply Updates Using kubectl:** You can update the application by changing the image tag in the Deployment file and applying it
3. **Monitor the Update Progress:** Use the following commands to observe the update:
   Check Deployment status: **kubectl rollout status deployment/my-app**
   View update history: **kubectl rollout history deployment/my-app**
4. **Rollback if Needed:** If the update causes issues,
   rollback to the previous version:
   kubectl rollout undo deployment/my-app
   You can also specify a specific revision to rollback to:
   kubectl rollout undo deployment/my-app --to-revision=2
5. **Blue-Green Deployment (Optional)**
   Instead of a rolling update, you can create a new Deployment with the updated version and use a Service to switch traffic between the old and new versions.
6. **Canary Deployment (Optional)**
   This approach deploys the new version to a small subset of users first, allowing testing in a production-like environment. Gradually, the new version replaces the old one.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Explanation of the YAML:

- **apiVersion:** Specifies which API version the object is using (in this case, apps/v1).
- **kind:** Specifies that this resource is a Deployment.
- **metadata:** Includes information like the name of the Deployment.
- **spec:** The specifications of the Deployment.
    - **replicas:** Defines how many Pods you want to run (3 in this case).
    - **selector:** Defines the label selector to match the Pods managed by this Deployment.
    - **template:** Defines the template for the Pods that will be created by the Deployment. It includes the metadata and spec for the Pods (such as the container image and port to expose).

Complete Example Workflow

1. **Create Deployment:**
   kubectl apply -f nginx-deployment.yaml
2. **Expose Deployment via Service:**
   kubectl apply -f nginx-service.yaml
3. **Scale Deployment:**
   kubectl scale deployment nginx-deployment --replicas=5
4. **Update Deployment:**
   kubectl set image deployment/nginx-deployment nginx=nginx:1.21
5. **Check Deployment Status:**
   kubectl rollout status deployment/nginx-deployment
6. **Rollback if Needed:**

kubectl rollout undo deployment/nginx-deployment

7. **View Services:**
   kubectl get services

Failed Deployment:

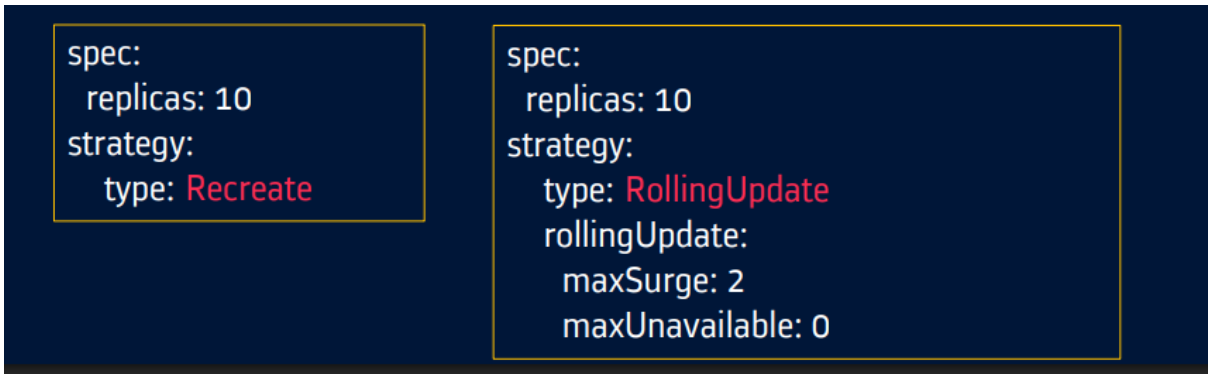
Your Deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

- Insufficient quota
- Readiness probe failures
- Image pull errors
- Insufficient permissions
- Limit ranges
- Application runtime misconfiguration

Deployment Strategy:

Deployment strategies are different ways of rolling out a new version of an application in a Kubernetes cluster.

- Whenever we create a new deployment, K8s triggers a Rollout
- Rollout is the process of gradually deploying or upgrading your application containers.
- For every rollout/upgrade, a version history will be created, which helps in rolling back to working version in case of an update failure
- In Kubernetes there are a few different ways to release updates to an application
  - **Recreate:** terminate the old version and release the new one. Application experiences downtime.
  - **RollingUpdate:** release a new version on a rolling update fashion, one after the other. **It's the default strategy in K8s. No application downtime is required.**
  - **Blue/green:** release a new version alongside the old version then switch traffic



Kubernetes

Rolling Update Strategy

- By default, deployment ensures that only 25% of your pods are unavailable during an update and does not update more that 25% of the pods at a given time
- It does not kill old pods until/unless enough new pods come up
- It does not create new pods until a sufficient number of old pods are killed
- There are two settings you can tweak to control the process: maxUnavailable and maxSurge. Both have the default values set - 25%
- The maxUnavailable setting specifies the maximum number of pods that can be unavailable during the rollout process. You can set it to an actual number(integer) or a percentage of desired pods

  Let's say maxUnavailable is set to 40%. When the update starts, the old ReplicaSet is scaled down to 60%.
  As soon as new pods are started and ready, the old ReplicaSet is scaled down again and the new ReplicaSet is scaled up. This happens in such a way that the total number of available pods (old and new, since we are scaling up and down) is always at least 60%.
- The maxSurge setting specifies the maximum number of pods that can be created over the desired number of pods

  If we use the same percentage as before (40%), the new ReplicaSet is scaled up right away when the rollout starts. The new ReplicaSet will be scaled up in such a way that it does not exceed 140% of desired pods. As old pods get killed, the new ReplicaSet scales up again, making sure it never goes over the 140% of desired pods

Kubernetes

Deployments

- kubectl create deployment nginx --image nginx --dry-run -o yaml
- kubectl create -f deployment.yml --record (--record is optional, it just records the events in the deployment)

```
root@k-master:/home/osboxes# kubectl create -f deployment.yml
deployment.apps/nginx-deployment created
root@k-master:/home/osboxes#
```

- kubectl get deployments

```
root@k-master:/home/osboxes# kubectl get deployments -o wide
NAME               READY   UP-TO-DATE   AVAILABLE   AGE    CONTAINERS
nginx-deployment   5/5     5            5           117s   nginx-container
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
```

Kubernetes

Deployments

- kubectl describe deployment <deployment-name>

```
root@k-master:/home/osboxes# kubectl describe deployment nginx-deployment
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Tue, 19 May 2020 03:40:19 -0400
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision: 1
                        kubernetes.io/change-cause: kubectl create --filename=deployment.yml
Selector:               app=nginx
Replicas:               5 desired | 5 updated | 5 total | 5 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx-container:
    Image:        nginx
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-96577bc6d (5/5 replicas created)
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
```

Kubernetes

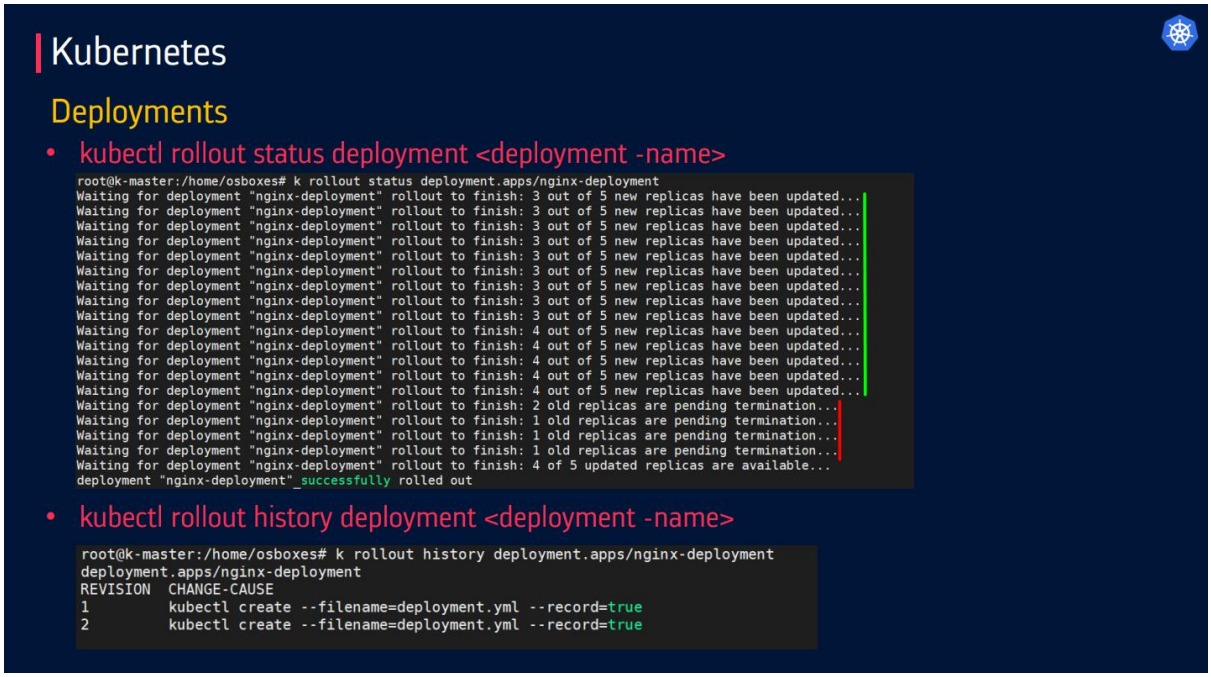
Deployments

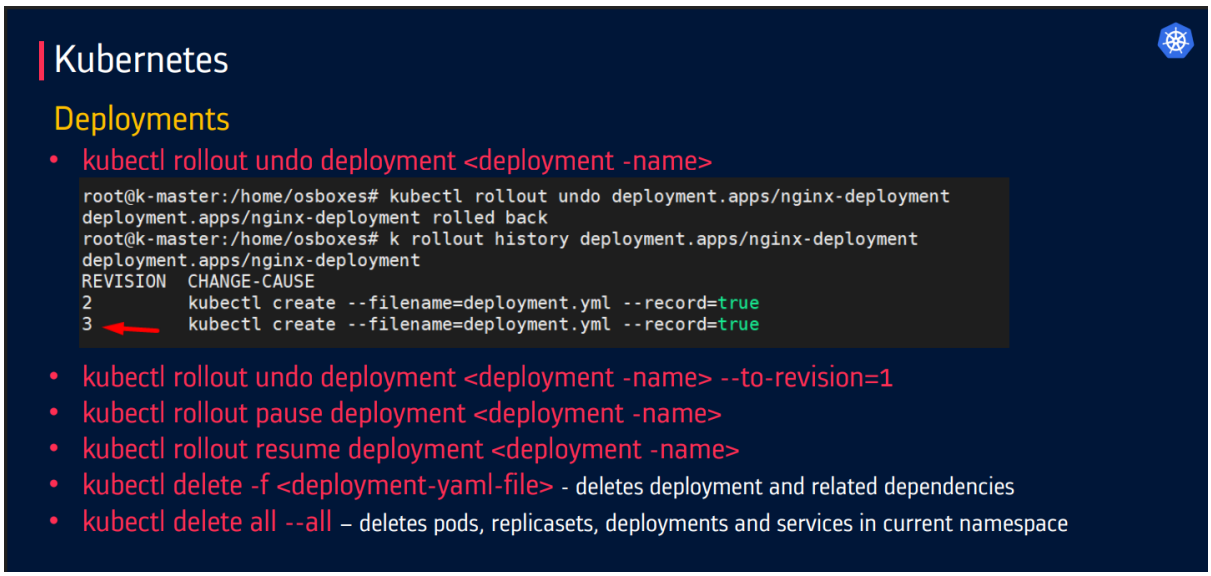
- kubectl get pods –o wide

```
root@k-master:/home/osboxes# kubectl get pods -o wide
NAME                                READY   STATUS    RESTARTS   AGE     IP             NODE
TES
nginx-deployment-96577bc6d-2hkpk    1/1     Running   0          3m34s   10.244.2.20    k-slave02
nginx-deployment-96577bc6d-h5gdv    1/1     Running   0          3m34s   10.244.2.19    k-slave02
nginx-deployment-96577bc6d-nqtn6    1/1     Running   0          3m34s   10.244.2.22    k-slave02
nginx-deployment-96577bc6d-pd4cg    1/1     Running   0          3m34s   10.244.2.21    k-slave02
nginx-deployment-96577bc6d-tphhh    1/1     Running   0          3m34s   10.244.2.23    k-slave02
root@k-master:/home/osboxes#
```

- kubectl edit deployment <deployment -name> - perform live edit of deployment
- kubectl scale deployment <deployment -name> --replicas2
- kubectl apply –f deployment.yml – redeploy a modified yaml file; Ex: replicas changed to 5, image to nginx:1.18

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 10
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx-container
        image: nginx
        ports:
        - containerPort: 80
```





In **Kubernetes**, a **Deployment Strategy** defines **how updates** to your application are rolled out. It controls **how new versions of pods replace old ones**, ensuring minimal downtime and maximum reliability.

🚀 Common Deployment Strategies

1. Rolling Update (Default)

- **Gradually replaces** old pods with new ones.
- Ensures **zero downtime**.
- You can control the speed using:
  - maxUnavailable: How many old pods can be unavailable during the update.
  - maxSurge: How many extra pods can be created temporarily.

strategy:
type: RollingUpdate
rollingUpdate:
maxUnavailable: 1
maxSurge: 1

2. Recreate

- **Stops all old pods first**, then starts new ones.
- Causes **downtime**, but ensures a clean start.
- Useful when old and new versions **can't run together** (e.g., database schema changes).

strategy:
type: Recreate

Blue/Green and **Canary** deployments are advanced deployment strategies used in Kubernetes and DevOps to release new versions of applications **safely and with minimal risk**.

🔵🟢 **Blue/Green Deployment**

Concept: You have two identical environments:

- **Blue**: The current live version.
- **Green**: The new version you're preparing to release.

How it works:

1. Deploy the new version to the **Green** environment.
2. Test it thoroughly.
3. Switch traffic from **Blue** to **Green** (usually via a load balancer).
4. If something goes wrong, you can quickly roll back to **Blue**.

Pros:

- Instant rollback.
- No downtime during switch.

Cons:

- Requires double the infrastructure (two environments).

🐤 **Canary Deployment**

Concept: Gradually roll out the new version to a small subset of users before a full rollout.

How it works:

1. Deploy the new version to a small percentage of users (e.g., 5%).
2. Monitor for errors or performance issues.
3. Gradually increase traffic to the new version.
4. If stable, complete the rollout; if not, roll back.

Pros:

- Safer, controlled rollout.
- Real-world testing with minimal risk.

Cons:

- More complex monitoring and traffic routing setup.

Kubernetes Deployment VS StatefulSet:

Deployments are suitable for stateless applications, where each instance of the application is identical and can be scaled horizontally. **StatefulSets**, on the other hand, are used for stateful applications that require stable network identities and persistent storage. **StatefulSets** provide ordering guarantees and allow for scaling vertically

A StatefulSet is a workload API object for managing stateful applications. They let you ensure that pods are scheduled in a specific order, that they have persistent storage volumes available, and that they have a persistent network ID that is maintained even when a pod shuts down or is rescheduled.

Similar to Deployments, StatefulSets manage the behavior of pods. However, it differs from deployments in that it maintains a static ID for each pod. Pods are created from the same template, but each one has a separate identity, making it possible to maintain a persistent state across the full lifecycle of the pod.

Deployment	**StatefulSet**
Intended for stateless applications	Intended for stateful applications
Pods are interchangeable	Pods are unique and have a persistent ID
All replicas share the same volumes and PersistentVolumeClaims	Each pod has its own volumes and PersistentVolumeClaims

https://codefresh.io/learn/kubernetes-deployment/

5) DaemonSet

DaemonSet is a Kubernetes object that ensures that a specific pod runs on every node in a cluster. It is useful for running background tasks, system monitoring, or log collection on every node.

Example YAML:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: logging
  labels:
    app: fluentd-logging
```

```yaml
spec:
  selector:
    matchLabels:
      name: fluentd
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
```

🧱 **What is a DaemonSet?**
A **DaemonSet** is used to deploy **background or system-level services** that need to run on **all nodes** (or a subset of nodes) in a Kubernetes cluster.

🔧 **Common Use Cases**

Use Case	Description
Log collection	Run agents like Fluentd or Logstash on every node.
Monitoring	Deploy Prometheus Node Exporter or Datadog agents.
Networking	Run CNI plugins or kube-proxy on each node.
Security	Install security agents or scanners on all nodes.

Key Features of DaemonSets

1. **Node-Level Deployment**:

- A DaemonSet deploys one pod per node in the cluster (or selected nodes based on label selectors).
- When a new node is added, the DaemonSet automatically creates a pod on it. If a node is removed, the pod associated with it is also removed.
- We can restrict DaemonSets to **Selective Nodes** using node selectors, node affinity, or taints and tolerations.

2. **Updates**:
   - DaemonSets support rolling updates, allowing pods to be updated incrementally across nodes.
   - DaemonSets automatically maintain the desired pod specification on nodes. If you update the DaemonSet, the pods on the nodes will be updated accordingly
3. **Self-Healing**:
   - If a new node is added to the cluster, a DaemonSet automatically creates the pod on the new node.
   - If a node is removed, the pod is automatically deleted.

How DaemonSets Ensure Running a Pod on Each Node:

1. **Controller Mechanism:**
   - **DaemonSet Controller** monitors the cluster state. It ensures the desired number of pods (one per eligible node) is maintained by reconciling the cluster's actual state with the desired state defined in the DaemonSet.
2. **Pod Scheduling:**
   - **When you create a DaemonSet, Kubernetes:**
   - Schedules the DaemonSet pod on all eligible nodes using the kube-scheduler.
   - Ensures these pods run even if there are changes to the node pool, such as adding or removing nodes.
3. **Node Addition or Removal:**
   - **New Node Added:** When a new node is added, the DaemonSet controller detects it and schedules a pod on the node.
   - **Node Removed:** If a node is removed, the corresponding pod is deleted.
4. **Immutable Management**:
   - Each DaemonSet pod is managed as an individual unit, but Kubernetes ensures that every eligible node runs exactly one instance of the pod.

When to use DaemonSets:

- **Log Collection:** Running logging agents like Fluentd or Logstash on every node for gathering metrics or logs from all nodes.
- **Monitoring:** Running system monitoring agents like Prometheus Node Exporter or datadog.
- **Networking:** Running network-related tools, such as a CNI plugin or a network proxy.

Use Case	Example
Log collection	Fluentd, Logstash
Monitoring agents	Prometheus Node Exporter, Datadog
Storage plugins	Ceph, GlusterFS
Network plugins	Calico, Weave

Security tools	Falco, antivirus agents

How DaemonSets Differ from Other Controllers

- **Deployment**: Focuses on stateless applications and scalable services, with pods distributed across nodes based on resource needs.
- **StatefulSet**: Manages stateful applications where stable pod identities and persistent storage are required.
- **DaemonSet**: Ensures pods are deployed uniformly across all (or selected) nodes, mainly for system-level tasks.

6) StatefulSet

StatefulSet:

- A **StatefulSet** is a Kubernetes controller designed to **manage and deploy stateful applications** that require **persistent storage**, **stable network identity**, and **ordered deployment or scaling.**
- It ensures each Pod has a **unique and stable identity** (like pod-0, pod-1) and a **dedicated persistent storage volume** that persists even during scaling, node failures, or replacements.
- Unlike **Deployments** used for **stateless applications**, **StatefulSets** are specifically built for **stateful workloads** where the **order of Pod creation**, **deletion**, and **stable identities** are crucial.
- They are ideal for running **databases** and **clustered applications** (e.g., MongoDB, Kafka, Cassandra).
- StatefulSets are defined using a YAML manifest that includes:
    - A **Pod template**
    - A **Headless Service** (clusterIP: None) for stable DNS
    - A **VolumeClaimTemplate** to create PVCs per Pod

🧱 What is a StatefulSet?

A **StatefulSet** ensures that:

- Each pod has a **unique, stable name** (e.g., web-0, web-1, web-2).
- Each pod gets its own **persistent volume**, which is **not shared** and **not deleted** when the pod is removed.
- Pods are **created, updated, and deleted in order**.

🔧 Key Features

Feature	Description
Stable Identity	Each pod keeps the same name and network identity.
Persistent Storage	Each pod gets a dedicated volume that persists across restarts.

Feature	Description
Ordered Operations	Pods are started, updated, and terminated in a defined order.
Use Case	Databases (MySQL, Cassandra), Kafka, Zookeeper, etc.

🧠 **When to Use StatefulSet?**
Use it when your application:

- Needs **stable hostnames**.
- Requires **persistent storage** tied to each pod.
- Must be **started or stopped in a specific order**.

Key Features of StatefulSets:

1. **Stable Pod Identity:**
   - Each pod in a StatefulSet has a unique, persistent hostname (e.g., app-0, app-1) that persists through restarts. Pattern **<statefulset-name>-<ordinal>.**
2. **Ordered Deployment and Scaling:**
   - Pods are created, updated, or terminated in a specific (sequential) order (e.g., Pod 0 before Pod 1), ensuring consistent application behavior during scaling or updates. When scaling down, the highest-numbered pod is removed first
3. **Persistent Storage:**
   - Each pod is associated with its own persistent volume, allowing data retention even if a pod is deleted or rescheduled.
4. **Ordered Rolling Updates:**
   - Updates to pods are performed in a controlled, sequential manner, ensuring minimal disruption to the application.

Example YAML:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis
  namespace: default
  labels:
    app: redis
spec:
  ports:
  - port: 6379
    protocol: TCP
  selector:
    app: redis
  type: ClusterIP
  clusterIP: None
```

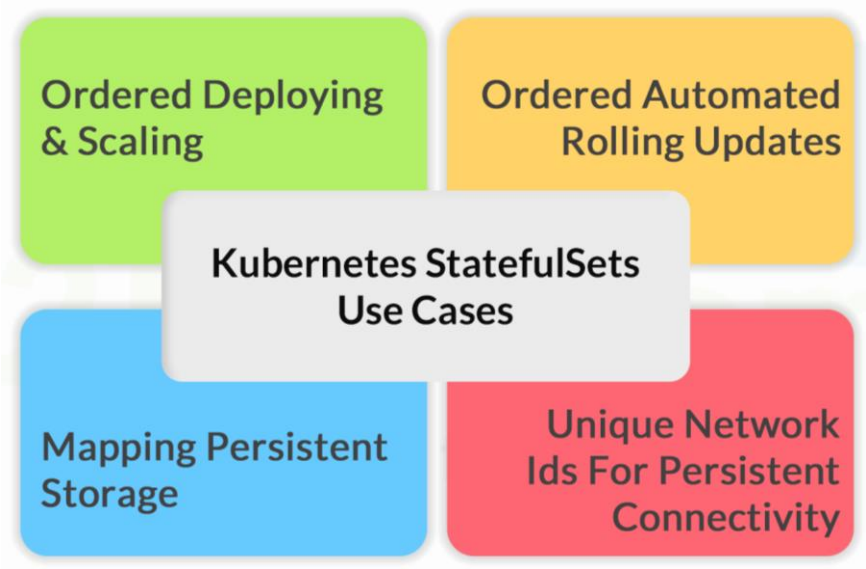
```yaml
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  selector:
    matchLabels:
      app: redis
  serviceName: "redis"
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
      - name: redis
        image: redis:5.0.4
        command: ["redis-server"]
        ports:
        - containerPort: 6379
          name: web
        volumeMounts:
        - name: redis-aof
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: redis-aof
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "gp2"
      resources:
        requests:
          storage: 1Gi
```

Explanation:

1. **Pod Naming**:
    - Pods are named sequentially, like mongodb-0, mongodb-1, mongodb-2.
2. **Persistent Volumes**:
    - Each pod is assigned a unique volume (mongo-data) that stores its data, ensuring that even if a pod is recreated, its data is retained.
3. **Order**:
    - Pods are created one after another (from mongodb-0 to mongodb-2) and terminated in reverse order.

Common Use Cases for StatefulSets

1. **Databases**:
    - Applications like MySQL, PostgreSQL, MongoDB, or Cassandra, where data consistency is critical.
2. **Distributed File Systems**:
    - Systems like Ceph or GlusterFS require persistent storage and stable network identities.
3. **Message Queues**:
    - Services like Kafka or RabbitMQ that rely on ordered operations.



Difference Between StatefulSet and Deployment

- **StatefulSet**:
    - Used for applications that need stable identities, persistent storage, and ordered operations.
    - Suitable for databases, distributed systems, or stateful applications.
- **Deployment**:
    - Designed for stateless applications that don't require stable storage or identities.
    - Ideal for web servers, APIs, or frontend applications.

Feature	StatefulSet	Deployment
Pod Identity	Stable, unique for each pod	Dynamic, shared across replicas
Persistent Storage	Dedicated per pod	Shared or ephemeral storage
Scaling Behavior	Ordered scaling	All pods can scale simultaneously
Rolling Updates	Sequential updates	Parallel updates

7) Jobs

Jobs:

- **A Job** is a Kubernetes controller used to **run a specific task to completion or batch job**. It is designed for one-time or on-demand tasks that terminate automatically after completion.
- Jobs are ideal for **short-lived tasks** such as batch processing, data analysis, or backups.
- A Job creates **one or more pods** to run the task and **monitor their completion status**. If a pod fails, the Job automatically replaces it to ensure the task is successfully completed.
- Jobs are configured using a **YAML manifest** that includes:
  - A **Pod template**
  - **Completion criteria** (completions, parallelism)
  - Optional retry settings (backoffLimit)

🧱 What is a Kubernetes Job?

A **Job** creates one or more pods and ensures that a specified number of them **successfully complete**. Once the task is done, the pods are terminated.

🔧 Key Features

Feature	Description
One-time execution	Runs a task once and exits.
Retry on failure	Automatically retries failed pods.
Parallelism	Can run multiple pods in parallel.
Completion tracking	Tracks how many pods have completed successfully.

🧠 When to Use a Job?

Use a Job when you need to:

- Run **data processing** or **ETL tasks**.
- Perform **database migrations**.
- Execute **scheduled tasks** (with CronJobs).
- Run **one-time scripts** or **maintenance tasks**.

Key Features of Jobs

1. **One-Time Tasks**: Jobs are intended for tasks that need to be run once or a specific number of times, like data processing or backups.
2. **Parallelism**: Jobs can be configured to run multiple Pods concurrently to speed up processing.
3. **Completion Tracking**: Kubernetes ensures that the Job completes successfully, even if Pods fail.

4. **Automatic cleanup:** Once the task is finished, the Job and its Pods are removed.
5. **Retry Mechanism**: If a pod fails during job execution, Kubernetes automatically retries the job until it succeeds or meets the set limit.

Example YAML:

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: kubernetes-parallel-job
  labels:
    jobgroup: jobexample
spec:
  completions: 6
  parallelism: 2
  template:
    metadata:
      name: kubernetes-parallel-job
      labels:
        jobgroup: jobexample
    spec:
      containers:
      - name: c
        image: devopscube/kubernetes-job-demo:latest
        args: ["100"]
      restartPolicy: OnFailure
```

Explanation:

1. **completions**: Specifies how many times the job needs to complete successfully (here, 1 time).
2. **parallelism**: Sets the number of pods to run simultaneously (here, 1 pod).
3. **restartPolicy**: Ensures the pod doesn't restart on completion (Never).

Common Use Cases for Jobs

1. **Batch Processing**:
    - Process large datasets (e.g., image or video processing).
2. **Data Backups**:
    - Perform database dumps and store them securely.
3. **Maintenance Tasks**:
    - Automate cleanup tasks like deleting outdated resources.
4. **Analytics**:
    - Run complex analytics computations on datasets.
5. **Migrations**:
    - When you need to run **one-time tasks** such as database migrations or batch processing.

8) CronJobs

CronJobs:

- A CronJob is a Kubernetes resource that schedules Jobs to run periodically based on a specified schedule. It's like a time-based trigger that initiates Jobs at predefined intervals.
- In Kubernetes, a **CronJob** is a specialized resource used to **schedule and run Jobs at specific time intervals or recurring schedules.**
- A **CronJob** runs **Jobs on a schedule** similar to Linux cron.

🕒 What is a CronJob?

A **CronJob** creates a **Job** object on a defined schedule. Each time the schedule is triggered, a new Job is created and executed.

🔧 Key Features

Feature	Description
Scheduled execution	Runs at specific times using cron syntax.
Automatic retries	Retries failed jobs based on configuration.
History limits	You can control how many successful or failed jobs are retained.
Time zone support	Can be configured to run in specific time zones (K8s v1.24+).

🧠 When to Use a CronJob?

Use a CronJob for:

- **Nightly backups**
- **Log rotation**
- **Database cleanup**
- **Scheduled report generation**
- **Recurring health checks**

Key Features of CronJobs

1. **Scheduled Execution:**
    - CronJobs run tasks at specified intervals, defined using a cron expression.
2. **Recurring Tasks:**
    - Automates repetitive tasks, such as backups, cleanup operations, log rotation, or data synchronization.
3. **Job Template:**
    - Each CronJob creates a regular Kubernetes Job when triggered, managing its execution.
4. **Failure Management:**

    - You can define limits for retries, concurrency, and how failed jobs should be handled.
5. **Time Zone Support:**
    - CronJobs can support time zones for specific scheduling needs.
6. **History:**
    - CronJobs keep a history of past executions, allowing you to track Job completion and failures.

Cron Expression Format:

The schedule for a CronJob is defined using a cron expression, which consists of five fields:

```
* * * * *
| | | | |
| | | | +--- Day of the week (0 - 7, both 0 and 7 represent Sunday)
| | | +----- Month (1 - 12)
| | +------- Day of the month (1 - 31)
| +--------- Hour (0 - 23)
+----------- Minute (0 - 59)
```

For example:

- `"*/5 * * * *"`: Runs every 5 minutes.
- `"0 0 * * *"`: Runs daily at midnight.

Example YAML:

```yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: kubernetes-cron-job
spec:
 schedule: "*/5 * * * *" # Every 5 minutes
 jobTemplate:
  spec:
   template:
    metadata:
     labels:
      app: cron-batch-job
    spec:
     restartPolicy: OnFailure
     containers:
     - name: kube-cron-job
      image: busybox:1.28
      command: ["echo", "Hello, Kubernetes CronJobs!"]
```

Explanation:

1. **schedule:** Defines when the CronJob should run using a cron expression.
2. **jobTemplate:** Specifies the template for the job that the CronJob will create.
3. **restartPolicy:** Ensures that the pod doesn't restart after completion (Never).

Common Use Cases for CronJobs

1. **Data Backups**:
   - Automating regular backups of databases, filesystems, or applications.
2. **Log Rotation/Cleanup**:
   - Automate deleting old logs files or temporary files to save disk space.
3. **Data synchronization:**
   - Keeping data consistent across different systems or databases
4. **Scheduled Emails**:
   - Send periodic notifications or updates.
5. **Monitoring and alerting:**
   - Running scripts to monitor system health and send alerts.
6. **Recurring Reports generation**:
   - Generating reports at regular intervals.

How Jobs and CronJobs Work Together:

1. **CronJob Scheduling:** The CronJob controller monitors the cluster for CronJob objects. When a CronJob's schedule matches the current time, it creates a new Job.
2. **Job Execution:** The Job controller creates Pods to execute the task defined in the Job spec.
3. **Task Completion:** The Pods run the task and report their status to the Job controller.
4. **Job Completion:** Once all Pods associated with the Job complete successfully, the Job is considered finished.
5. **Cleanup:** The Job and its Pods are automatically deleted.

By effectively utilizing Jobs and CronJobs, you can automate routine tasks, optimize resource utilization, and ensure the reliability and efficiency of your Kubernetes applications.

Feature	Job	CronJob
Purpose	Executes tasks once or until complete.	Schedules recurring tasks.
Trigger	Manually or programmatically triggered.	Runs on a defined schedule (cron syntax).
Use Case	Database migration, file processing.	Nightly backups, scheduled reports.
Retries	Supports retries on failure.	Inherits retries from the Job template.

Feature	Pod	ReplicaSet	Deployment	DaemonSet	StatefulSet	Job	CronJob
Main Purpose	Runs a single container	Maintains a fixed number of pods	Manages app updates & scaling	Runs a pod per node	Manages stateful apps	Runs a task once	Runs scheduled tasks
Self-healing?	No	Yes	Yes	Yes	Yes	No	No
Scaling?	No	Yes	Yes	No	Yes	No	No
Persistent Storage?	No	No	No	No	Yes	No	No
Use Case	Simple apps	Ensures pod count	Web apps, APIs	Logging, Monitoring	Databases, Kafka	Data processing	Scheduled tasks

9) Namespace in Kubernetes

Namespace:

- **A Kubernetes Namespace** is a virtual cluster within a Kubernetes cluster. It provides a way to organize and isolate resources such as Pods, Services, Deployments, and other objects within the cluster. Namespaces allows different teams or projects to use the same cluster without affecting each other's resources, ensuring resource isolation and access control.
- You can think of Namespaces as separate "rooms" within a "house" (the cluster), where each room contains its own set of furniture (resources) and can be managed independently.

🧱 What is a Namespace?

A Namespace is used to:

- **Organize and isolate resources (like pods, services, deployments).**
- **Apply different policies (like resource limits or access control).**
- **Support multi-tenancy in a single cluster.**

📦 Default Namespaces in Kubernetes

Namespace	Purpose
default	**Used when no other namespace is specified.**
kube-system	**Contains system components like kube-dns, kube-proxy.**
kube-public	**Readable by all users (used for public info).**
kube-node-lease	**Tracks node heartbeats for node health monitoring.**

🧠 **When to Use Namespaces?**

Use namespaces when you want to:

- **Separate environments (e.g., dev, staging, prod).**
- **Isolate teams or projects.**
- **Apply different resource quotas or RBAC rules.**

Key Features of Namespaces:

- **Resource Isolation:** Namespaces allows isolation of environments for different teams or projects to use the same cluster without interference.
- **Resource Quota Management:** It defines limits (e.g., CPU, memory) for resources within a namespace to avoid resource contention.
- **Access Control:** Kubernetes uses Role-Based Access Control (RBAC) to restrict access to specific users or teams in different namespaces.
- **Network Isolation (Policies):** Namespaces are used to control communication between resources in different namespaces. However, by default, Pods in different namespaces can communicate with each other.
- **Default Namespace**
- **Ease of Management**

How Does a Namespace Organize Resources in Kubernetes?

1. **Scoping Resources:** Resources like Pods, Services, ConfigMaps, and Secrets are created within a specific namespace.
   - kubectl get pods -n dev
   - kubectl get pods -n prod
2. **Default Namespace:** If no namespace is specified, resources are created in the default namespace.
3. **System Namespaces:** Kubernetes reserves certain namespaces for system resources:
   1. **kube-system:** For core Kubernetes components runs (system components) (e.g., the scheduler, controller manager).
   2. **kube-public:** For publicly accessible resources. This namespace is generally reserved for cluster usage like Configmaps and Secrets.
   3. **kube-node-lease:** For node heartbeat leases.
4. **Resource Isolation:** Namespaces isolate resources logically. Different teams or projects can use separate namespaces to prevent conflicts. A Service in the dev namespace cannot directly resolve or access a Pod in the prod namespace without explicit configuration.
5. **Cross-Namespace Operations:** Admin-level operations can view or manage resources across all namespaces using the --all-namespaces flag.
   - kubectl get pods --all-namespaces.
6. **Quotas and Limits**: You can set resource quotas to control CPU and memory usage within a namespace

Kubernetes

Namespaces

kubectl get namespaces

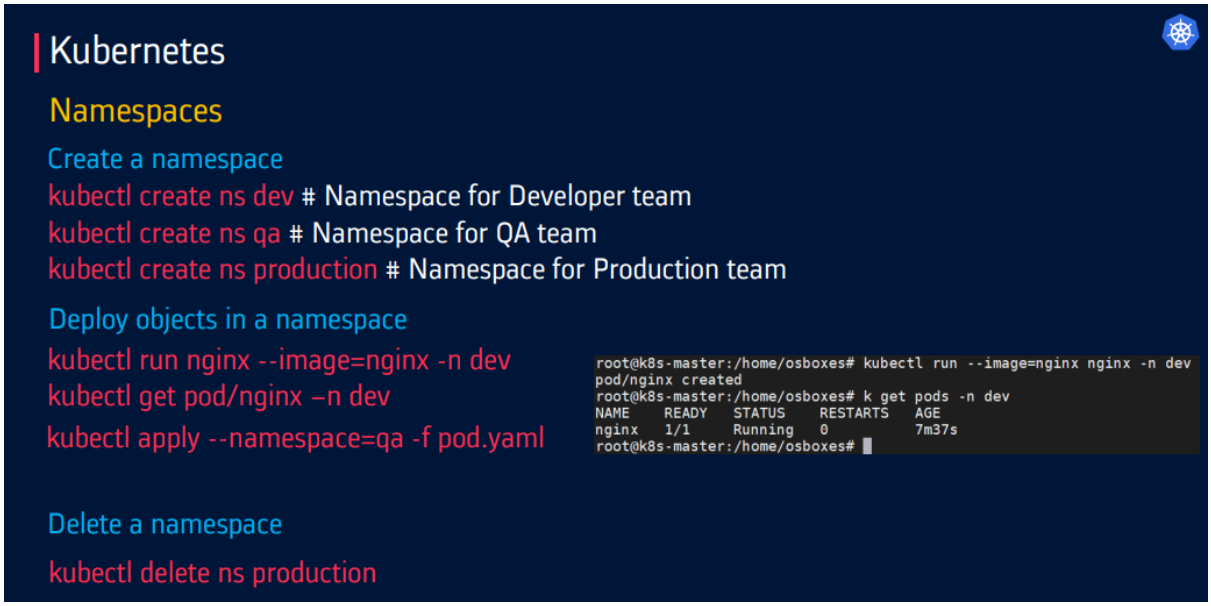
```
root@k8s-master:/home/osboxes# kubectl get ns
NAME              STATUS   AGE
default           Active   68m
kube-node-lease   Active   68m
kube-public       Active   68m
kube-system       Active   68m
```

kubectl get all -n kube-system (lists available objects under a specific namespace)

```
root@k8s-master:/home/osboxes# kubectl get all -n kube-system
NAME                                       READY   STATUS    RESTARTS   AGE
pod/coredns-66bff467f8-bm6kr               1/1     Running   0          68m
pod/coredns-66bff467f8-hj9ll               1/1     Running   0          68m
pod/etcd-k8s-master                        1/1     Running   0          68m
pod/kube-apiserver-k8s-master              1/1     Running   0          68m
pod/kube-controller-manager-k8s-master     1/1     Running   0          68m
pod/kube-flannel-ds-amd64-bc5vg            1/1     Running   0          67m
pod/kube-flannel-ds-amd64-cg48x            1/1     Running   0          68m
pod/kube-flannel-ds-amd64-xz8qn            1/1     Running   0          67m
pod/kube-proxy-rq77v                       1/1     Running   0          68m
pod/kube-proxy-tl99m                       1/1     Running   0          67m
pod/kube-proxy-w7bqz                       1/1     Running   0          67m
pod/kube-scheduler-k8s-master              1/1     Running   0          68m

NAME               TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
service/kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   68m
```

kubectl get all --all-namespaces (lists available objects under all available namespaces)



10) Volumes

Volumes:

- In Kubernetes, **Volumes** are used to provide **persistent storage** to containers. Pods are temporary, but **Volumes help keep the data alive** even if the Pod dies or restarts.
- a Volume is a directory accessible to containers in a Pod that allows data to persist beyond the container's lifecycle.

In Kubernetes, Volumes are used to manage persistent or shared storage for containers running in Pods. Unlike the ephemeral storage inside a container, volumes provide a way to persist data across container restarts or share data between containers in the same Pod.

- **Key Concepts of Kubernetes Volumes**
  1. **Ephemeral vs Persistent Storage**
     - **Ephemeral: Data is lost when the container restarts.**
     - **Persistent: Data is retained using volumes, even if the container restarts.**
  2. **Volume Lifecycle**
     - **A volume exists as long as the Pod exists.**
     - **For longer persistence, use Persistent Volumes (PVs) and Persistent Volume Claims (PVCs).**

- **Types of Volumes**

Here are some commonly used volume types:

Volume Type	Description
emptyDir	Temporary storage shared between containers in a Pod. Deleted when the Pod is removed.
hostPath	Mounts a file or directory from the host node's filesystem.
configMap	Mounts configuration data as files.
secret	Mounts sensitive data like passwords or tokens.
persistentVolumeClaim	Connects to a Persistent Volume for durable storage.
nfs	Mounts a Network File System share.
awsElasticBlockStore, gcePersistentDisk, azureDisk	Cloud provider-specific persistent storage.

- **Persistent Volumes (PV) and Persistent Volume Claims (PVC)**
  - **PV: A piece of storage in the cluster provisioned by an admin or dynamically.**
  - **PVC: A request for storage by a user.**

Key Components:

PV (PersistentVolume)
A pre-provisioned storage resource in the cluster, like a hard disk.

PVC (PersistentVolumeClaim)
A request made by a Pod to use storage (i.e., "I need 5Gi of space").
StorageClass
Defines **how storage is created,** including type (like SSD, HDD), speed, etc. It enables **dynamic provisioning** of storage.

Key Features of Kubernetes Volumes

Feature	Description
💾 **Persistent Storage**	Volumes can retain data even when a container is restarted (as long as the pod exists). Keeps data alive beyond Pod lifetime.
🔌 **Dynamic & Static**	Supports both pre-created (static) and auto-created (dynamic) volumes
🧠 **Decoupled**	PVs are independent of Pods
📦 **Claim-based access**	Pods request storage via PVCs
🛠️ **Plugin support**	Works with many storage backends (AWS EBS, GCP PD, NFS, etc.)
🎛️ **Configurable**	Define performance, access modes, reclaim policies, etc.
Shared Storage	Multiple containers in a pod can access the same volume, facilitating data sharing

Types of Kubernetes Volumes

There are different types of volumes you can use in a Kubernetes pod:
❑ **Node-local memory** (emptyDir and hostPath)
❑ **Cloud volumes** (e.g., awsElasticBlockStore, gcePersistentDisk, and azureDiskVolume)
❑ **File-sharing volumes,** such as Network File System (NFS)
❑ **Distributed-file systems** (e.g., CephFS and GlusterFS)
❑ **Special volume types** such as PersistentVolumeClaim, secret, configmap and gitRepo

1. **EmptyDir**:
    - Temporary storage shared between containers in a Pod. Lives as long as the Pod exists.
2. **HostPath**:
    - Mounts (Maps) a file or directory from the node's filesystem into the Pod.
3. **PersistentVolume (PV) and PersistentVolumeClaim (PVC)**:
    - PV: Pre-provisioned storage in the cluster, independent of pods.
    - PVC: A request for PV by a pod to claim storage resources.

    - PersistentVolumes can be backed by cloud storage (e.g., AWS EBS, Azure Disk, GCE Persistent Disk).
4. **ConfigMap and Secret**:
    - ConfigMap: Injects configuration data into containers.
    - Secret: Injects sensitive data like passwords or tokens.
5. **NFS (Network File System)**:
    - Allows multiple pods across nodes to access shared storage on an NFS server.
    - Mounts a remote NFS share.
6. **CSI (Container Storage Interface)**:
    - Enables Kubernetes to integrate with external storage systems.
7. **Cloud Storage Options**:
    - Includes specific volume types such as awsElasticBlockStore, azureDisk, and gcePersistentDisk for cloud provider storage.
8. **Ephemeral Volumes**:
    - Types like emptyDir or configMap last only for the lifecycle of a pod.

Example Volume Configuration

Here's an example of a pod with a PersistentVolume and PersistentVolumeClaim:

PersistentVolume (PV):

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

PersistentVolumeClaim (PVC):

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Pod Using PVC:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  volumes:
  - name: my-volume
    persistentVolumeClaim:
      claimName: my-pvc
  containers:
  - name: my-container
    image: busybox
    volumeMounts:
    - mountPath: /data
      name: my-volume
    command: ["sh", "-c", "echo Hello Kubernetes > /data/hello.txt && sleep 3600"]
```

When to Use Volumes

- **EmptyDir**: Sharing temporary data between containers in a pod.
- **PersistentVolumes**: When data needs to persist beyond the lifecycle of a pod or container.
- **Secrets/ConfigMaps**: For injecting configuration or sensitive data securely.
- **Cloud Volumes**: For large-scale deployments with cloud-native storage.

Access Modes

Mode	Description
ReadWriteOnce (RWO)	**One node can read/write**
ReadOnlyMany (ROX)	**Many nodes can read**
ReadWriteMany (RWX)	**Many nodes can read/write (e.g., NFS)**

Reclaim Policy (For PVs)

Policy	What Happens When PVC is Deleted
Retain	**Keeps the data (manual cleanup needed)**
Delete	**Deletes the storage automatically**
Recycle	**(Deprecated) Basic cleanup and reuse**

Real-Time Use Cases

Use Case	How It Helps
🌐 Web server storing uploads	PVC used to persist user files
💉 Databases (MySQL, PostgreSQL)	Data survives Pod restart
🧠 AI/ML workloads	Persist training data or model checkpoints
⏫ Backups and logs	Store logs outside the container lifecycle
☁️ Cloud-native apps	Auto provision volumes with StorageClass

📘 **Summary Table**

Component	Description
PV	Actual storage unit, like a disk
PVC	Claim/request for storage
StorageClass	Template to dynamically create storage
Access Modes	RWO, ROX, RWX
Reclaim Policy	Retain, Delete, Recycle
Use Cases	Databases, file servers, ML, backups

11) ConfigMaps and Secrets in Kubernetes

ConfigMaps and Secrets are essential resources in Kubernetes used for managing configuration data and sensitive information for applications running within the cluster.

1. ConfigMaps

- A **ConfigMap** is an object in Kubernetes used to **store configuration data** as **key-value pairs**.
- Think of it like a **settings container** for your app. Instead of writing settings (like environment = "production") in your app code, you store them separately in a ConfigMap and **inject them into your app at runtime**.

In Kubernetes, ConfigMaps and Secrets are used to manage configuration data and sensitive information separately from application code. They help keep your applications portable and secure.

🔹 **ConfigMaps**

A ConfigMap is used to store non-sensitive configuration data in key-value pairs. This can include things like:

- **Application settings**
- **Environment variables**
- **Command-line arguments**
- **Configuration files**

✅ **Use Cases:**

- **Injecting environment variables into containers**
- **Mounting configuration files into Pods**
- **Decoupling configuration from container images**

🔸 **Secrets**

A Secret is used to store sensitive data, such as:

- **Passwords**
- **API keys**
- **TLS certificates**

Secrets are base64-encoded and can be encrypted at rest by the Kubernetes API server.

✅ **Use Cases:**

- **Storing credentials securely**
- **Injecting secrets into containers as environment variables or mounted files**

Key Features of ConfigMaps:

1. **Dynamic Configuration**: Update configurations without redeploying containers.
2. **Key-Value Format**: Organize data in an easy-to-use format.
3. **Application Environment Variables**: Pass configuration details as environment variables or volume mounts.

Example of ConfigMap:

Create a ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  ENV: production
  LOG_LEVEL: info
  API_URL: https://api.example.com
```

Use ConfigMap in a Pod (as Environment Variables)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
    - name: app-container
      image: myapp:latest
      env:
```

```
        - name: ENV
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: ENV
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: LOG_LEVEL
```

Use ConfigMap as a File (Mounted Volume)

```
volumes:
  - name: config-volume
    configMap:
      name: app-config
containers:
  - name: app
    volumeMounts:
      - name: config-volume
        mountPath: /etc/config
```

2. Create a ConfigMap to store application settings:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_ENV: production
  LOG_LEVEL: debug
```

Use the ConfigMap in a Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: example-container
    image: busybox
    env:
    - name: APP_ENV
      valueFrom:
```

```
      configMapKeyRef:
        name: app-config
        key: APP_ENV
  - name: LOG_LEVEL
    valueFrom:
      configMapKeyRef:
        name: app-config
        key: LOG_LEVEL
```

Explanation:

- **APP_ENV:** The environment variable is set to production from the ConfigMap.
- **LOG_LEVEL:** Configures the logging level as debug.

Inside your container, the config will appear as files:

- /etc/config/ENV
- /etc/config/LOG_LEVEL

🌍 **Real-Time Use Cases**

Dev/Prod Configuration

Switch between different environments:

- **Dev:** ENV=dev
- **Prod:** ENV=production

🌐 **Store External URLs**

Store and manage service URLs:

- API_URL=https://api.example.com

🚀 **Feature Flags**

Enable/disable features:

- FEATURE_X=true

⚙️ **Tuning Parameters**

Like logging levels, timeouts, retries:

- LOG_LEVEL=debug
- RETRY_COUNT=3

Types of ConfigMaps

There's **no official "type" field**, but there are **ways to create ConfigMaps**:

A. **From Literal Values**
   kubectl create configmap my-config --from-literal=ENV=prod

B. **From a File**
   kubectl create configmap my-config --from-file=config.properties

C. **From a Directory**
   kubectl create configmap my-config --from-file=./config-dir/

Each file or line in the directory becomes a key in the ConfigMap.

Common Use Case:
Dynamically update the environment configuration for a web application without rebuilding or redeploying its containers.

- **Dynamic Updates**:
  - If used as environment variables: changes **won't reflect** unless the Pod is restarted.
  - If mounted as a volume: changes **can auto-refresh** (depending on how Kubernetes is set up).
- 📦 **Helm + ConfigMaps**: In production, you often manage ConfigMaps using Helm charts for templating and versioning.
- 🔁 **Rolling Updates**: You can trigger a rolling update by modifying the ConfigMap and restarting the Pods.

2. Secrets

A Secret is a Kubernetes object used to **store sensitive data** like**:**

- **Passwords**
- **API keys**
- **Tokens**
- **SSH keys**

Instead of hardcoding sensitive values into your app, store them in a Kubernetes Secret and pass them safely at runtime.

Key Features of Secrets:

Feature	Description
🔒 **Secure Data Handling**	Used to store sensitive information, like password, token and SHH key etc.
⚠️ **Encrypted Storage**	Encodes sensitive information using base64 (can enable encryption at rest).
📥 **Integration with Pods**	Pass secrets as environment variables, mounted files, or arguments.
♻️ **Reusable**	One Secret can be used by multiple Pods
🧼 **Keeps Code Clean**	Keeps passwords, tokens, and keys out of source code
🛡️ **Better Access Control**	Restricts access to secrets can be controlled via RBAC policies.

Example of Secrets:
Create a Secret to store database credentials:

```
apiVersion: v1
```

```yaml
kind: Secret
metadata:
  name: db-credentials
type: Opaque
stringData:
  username: admin
  password: SuperSecretPass123
```

Use the Secret in a Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: db-pod
spec:
  containers:
  - name: db-container
    image: busybox
    env:
    - name: DB_USERNAME
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: username
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-credentials
          key: password
```

Explanation:

- **DB_USERNAME:** Retrieves the username admin from the Secret.
- **DB_PASSWORD:** Retrieves the password SuperSecretPass123.

2. Create a Secret YAML

This stores a DB username and password (values must be base64 encoded):

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=       # base64 for "admin"
  password: cGFzc3dvcmQ=   # base64 for "password"
```

Use Secret in a Pod as Environment Variables

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-demo
spec:
  containers:
    - name: app
      image: myapp:latest
      env:
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: username
        - name: DB_PASS
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
```

Use Secret as a File (Mounted Volume)

```yaml
volumes:
  - name: secret-volume
    secret:
      secretName: db-secret
containers:
  - name: app
    volumeMounts:
      - name: secret-volume
        mountPath: "/etc/secret-data"
```

📁 **Now inside the container:**

- **/etc/secret-data/username will contain admin**
- **/etc/secret-data/password will contain password**

Common Use Case: Securely pass credentials for a database or external service to a container.

Comparison Between ConfigMaps and Secrets

Feature	ConfigMaps	Secrets
Purpose	Stores application configurations	Stores sensitive data
Data Encoding	Plain text	Base64 encoded (not encrypted)
Example Data	APP_ENV, LOG_LEVEL	DB_PASSWORD, API_TOKEN
Use Case	General configuration settings	Secure credentials for applications

Applying ConfigMaps and Secrets

Use the following commands:

- **Apply ConfigMap:** kubectl apply -f configmap.yaml
- **Apply Secret:** kubectl apply -f secret.yaml

Advanced Usage:

- Combine ConfigMaps and Secrets with Helm charts to manage configurations across environments.
- Use Secret integrations with cloud providers (e.g., AWS Secrets Manager) for enhanced security.

🌎 Real-Time Use Cases

Scenario	How Secrets Help
🔑 Database Access	Store DB credentials securely
🔐 API Authentication	Store API keys or tokens
🔐 TLS/SSL Certificates	Store TLS certs and private keys for HTTPS
🧩 Third-party services	Keep credentials for services like AWS, Stripe, etc.
💉 CI/CD Integration	Inject secrets into build pipelines safely

🧱 Types of Secrets

Type	Description
Opaque (default)	Generic key-value pairs (most common)
kubernetes.io/basic-auth	For username/password pairs
kubernetes.io/ssh-auth	SSH keys
kubernetes.io/tls	TLS cert and private key
kubernetes.io/dockerconfigjson	Docker registry credentials for private image pulls

⚙️ Additional Functionalities

- **Create Secret from CLI**
  kubectl create secret generic db-secret --from-literal=username=admin --from-literal=password=password

🔁 Auto-rotation

- **Kubernetes doesn't auto-rotate secrets.**
- **Use external tools like Vault, Sealed Secrets, or External Secrets Operator for auto-rotation or syncing from secret managers.**

🔐 Encryption at Rest

- **Secrets are stored base64 encoded by default.**
- **For better security, enable encryption at rest in your cluster using KMS or a custom encryption provider.**

👮 RBAC Control

- You can control who can access secrets using Kubernetes RBAC policies.

12) Resource Limits in Kubernetes

Resource Limits:

- In Kubernetes, **Resource Limits** are used to control how much **CPU and memory (RAM)** a container can **use or request**.
- They ensure efficient resource allocation, prevent containers from exhausting cluster resources, and allow better control over application performance.
- This helps **prevent one container from hogging all the cluster resources** and affecting others.

In Kubernetes, **Resource Limits** are used to control how much **CPU and memory (RAM)** a container can use. This helps ensure fair resource distribution, prevents resource hogging, and improves cluster stability.

🔹 Why Use Resource Limits?

- Prevent a single container from consuming all node resources.
- Ensure high availability and performance.
- Enable Kubernetes to make better scheduling decisions.
- Protect against memory leaks or runaway processes.

🔸 Key Terms

Term	Description
Requests	Minimum amount of CPU/memory guaranteed for the container. Used by the scheduler to place Pods.
Limits	Maximum amount of CPU/memory a container can use. If exceeded, the container may be throttled or killed.

🔐 What Happens If Limits Are Exceeded?

- **CPU**: The container is **throttled** (slowed down), not killed.
- **Memory**: The container is **terminated** (OOMKilled) if it exceeds the memory limit.

🔧 Best Practices

- Always set both **requests** and **limits**.
- Use **resource quotas** at the namespace level to control total usage.
- Monitor usage with tools like **Prometheus**, **Grafana**, or **Kubernetes Metrics Server**.

Key Features of Resource Limits

1. **Requests**: Define minimum resources (CPU & memory) to a container needs to run.
2. **Limits**: Specify the maximum resources a container is allowed to use.
3. **Quality of Service (QoS)**: Kubernetes uses resource requests and limits to categorizes Pods based on their resource settings:
    - **Guaranteed**: Requests = Limits → Stable & reliable performance
      Both requests and limits are equal, ensuring reliable performance.
    - **Burstable**: Limits > Requests → Can use extra resources if available
      Pods with higher limits than requests can use extra resources when available.
    - **BestEffort**: No requests or limits → May get less priority
      No requests or limits are defined, so resource allocation is uncertain.

Real-Time Example of Resource Limits

Define resource limits for a container running a web server:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: resource-limits-example
spec:
  containers:
  - name: nginx
    image: nginx:latest
    resources:
      requests:
        memory: "128Mi"
        cpu: "500m"
      limits:
        memory: "256Mi"
        cpu: "1"
```

Explanation:

- **Requests**:
    - memory: "128Mi": Guarantees the container will have at least 128MB of memory.
    - cpu: "500m": Ensures the container gets 0.5 CPU cores minimum.
- **Limits**:
    - memory: "256Mi": Caps memory usage to 256MB.
    - cpu: "1": Caps CPU usage to 1 full core.

2. Define CPU & Memory Resource Requests and Limits

```yaml
apiVersion: v1
kind: Pod
metadata:
```

```yaml
  name: resource-demo
spec:
  containers:
    - name: app
      image: myapp:latest
      resources:
        requests:
          memory: "128Mi"
          cpu: "250m"
        limits:
          memory: "256Mi"
          cpu: "500m"
```

- 🔹 This means:
  - The container **requests** 128Mi memory and 250 millicores of CPU
  - The container can **use up to** 256Mi memory and 500 millicores of CPU

Apply the configuration:

- kubectl apply -f resource-limits-example.yaml

Quality of Service Classes

Based on the above example:

- Pods will be classified as **Burstable** because requests (128Mi) are lower than limits (256Mi).

Common Use Cases for Resource Limits

1. **Prevent Resource Contention**:
   🏗️ **Example:** Multiple applications running on the same node. Without limits, one pod might consume excessive resources, affecting others.
   ✅ **Solution:** Set strict resource limits to ensure fair allocation and stability.

2. **Optimize Performance for Critical Applications**:
   🌐 **Example:** An API server handling high traffic. Predictable resource usage is essential to maintain performance.
   ✅ **Solution:** Define equal resource requests and limits to ensure steady performance under varying load conditions.

3. **Dynamic Resource Allocation**:
   ⚡ **Example:** Batch jobs requiring bursts of CPU power when additional capacity is available.
   ✅ **Solution:** Configure low requests with high limits to allow usage spikes while preventing resource starvation.

Use Case	How Resource Limits Help
💻 **Prevent Crashing Nodes**	**Ensures no single pod consumes excessive memory or CPU, avoiding instability.**
🧪 **Testing Performance**	**Start with small resource limits in development, scale up in production.**
📦 **Multi-Tenant Clusters**	**Assign different resource limits per team or application to maintain fairness.**
🔄 **CI/CD Pipelines**	**Keep jobs within predefined resource caps to avoid impacting live workloads.**
🛡️ **Prevent Noisy Neighbors**	**Ensures one application doesn't degrade performance for others sharing the same node.**
📊 **Improve Scheduling**	**Helps Kubernetes efficiently place pods on available nodes for better resource distribution.**

Real-Time Scenario

Imagine a production environment with:

- **Critical Services** (e.g., payment gateway): Use Guaranteed QoS to ensure reliable performance with equal requests and limits.
- **Batch Jobs** (e.g., video processing): Use Burstable QoS with lower requests to allow flexible allocation when the system has idle resources.
- **Testing Pods** (e.g., debugging tools): Use BestEffort QoS without requests or limits to avoid impacting critical workloads.

Monitoring Resource Usage

Use Kubernetes tools like:

1. **kubectl describe pod**: View resource requests and limits for a pod.
2. **Metric Server**: Monitor live resource usage via kubectl top pod.
3. **Prometheus**: Collect and analyze resource metrics for detailed insights.

Best Practices

1. Set both requests and limits for predictable behavior.
2. Use performance testing to determine optimal resource configurations.
3. Implement pod autoscaling (via HPA) alongside resource limits for dynamic scaling during traffic spikes.

🧱 Types of Resources

- There are two main types:

Type	Unit	Example
CPU	millicores (m)	250m = 0.25 CPU core
Memory	Bytes (Mi, Gi)	128Mi, 1Gi

You can also set limits on **Ephemeral Storage**:

```
requests:
  ephemeral-storage: "500Mi"
limits:
  ephemeral-storage: "1Gi"
```

⚙️ **Additional Functionalities**

✅ **Default Resource Limits (LimitRange)**

You can set **default limits** at the namespace level:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
  namespace: dev
spec:
  limits:
  - default:
      memory: 256Mi
    defaultRequest:
      memory: 128Mi
    type: Container
```

So even if a developer forgets to set resource values, defaults will apply.

⚖️ **Resource Quotas**

You can combine Resource Limits with **Resource Quotas** to control the **total amount of resources** used by all Pods in a namespace:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-quota
spec:
  hard:
    requests.cpu: "2"
    requests.memory: "1Gi"
    limits.cpu: "4"
    limits.memory: "2Gi"
```

13) Custom Resource Definitions (CRDs) and Custom Resources (CRs) in Kubernetes

Custom Resource Definitions (CRDs) allow users to extend Kubernetes by defining their own resource types, while **Custom Resources (CRs)** are instances of those resource types. These mechanisms empower developers to create domain-specific abstractions that suit their application needs.

Custom Resource Definition:

- A **Custom Resource Definitions (CRDs)** is a way to **extend Kubernetes** by creating **your own resources**, just like built-in ones (like Pods, Services, etc.).

  🧠 Think of CRDs as:
  "I want Kubernetes to recognize and manage a **new kind of object** that I define, like Database, Backup, or LogCollector."
  Once defined, you can use kubectl to manage it like any native Kubernetes object.

In Kubernetes, Custom Resource Definitions (CRDs), Custom Resources (CRs), and Custom Controllers are powerful tools that allow you to extend Kubernetes with your own APIs and logic. Here's a breakdown of each:

🔹 1. Custom Resource Definitions (CRDs)

A CRD is a way to define a new resource type in Kubernetes, just like built-in types such as Pod, Service, or Deployment.

✅ **Purpose:**

- **Extend the Kubernetes API without modifying the core code.**
- **Define your own objects (e.g., MyApp, Database, BackupJob).**

📄 **Example CRD:**

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                schedule:
                  type: string
                                type: string
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
    - ct
```

🔸 **2. Custom Resources (CRs)**

A Custom Resource is an instance of a CRD. Once a CRD is created, you can create and manage CRs just like any other Kubernetes object.

📄 **Example Custom Resource:**

```
apiVersion: stable.example.com/v1
kind: CronTab
metadata:
  name: my-cron-job
spec:
  schedule: "*/5 * * * *"
  image: "my-cron-image"
```

🔧 **3. Custom Controllers (Operators)**

A Custom Controller watches for changes to your Custom Resources and performs actions in response. This is how you implement the logic behind your custom resource.

✅ **Responsibilities:**

- **Watch for events (create/update/delete) on CRs.**
- **Reconcile the actual state with the desired state.**
- **Automate complex operations (e.g., backups, scaling, healing).**

🛠️ **Tools to Build Controllers:**

- **Kubebuilder (official SDK)**
- **Operator SDK**
- **Client-go (low-level Go client)**

🔁 **How They Work Together**

1. **CRD defines a new resource type.**
2. **CR is an instance of that type.**
3. **Controller watches the CR and acts accordingly.**

📊 **Visual Diagram Offer**

Would you like a diagram showing how CRDs, CRs, and Controllers interact in the Kubernetes control loop? I can generate one for you!

Key Features of CRDs

1. **API Extension**: CRDs allow users to extend Kubernetes' API dynamically by creating new resource types like Database or Backup.
2. **Declarative Resource Management**: Custom Resources (CRs) can be created, updated, and deleted like native Kubernetes objects.
3. **Custom Controllers**: Controllers can manage CRs, ensuring the desired state of the system.

Real-Time Example of CRD and CR

Step 1: Define a CRD and Use it

Let's define a Database resource type:

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: databases.example.com
spec:
  group: example.com
  names:
    kind: Database
    plural: databases
    singular: database
    shortNames:
    - db
  scope: Namespaced
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              type:
                type: string
              version:
                type: string
              storage:
                type: string
```

Explanation:

- **group:** Specifies the API group for the custom resource (example.com).
- **names:** Defines how the resource will be identified (e.g., Database with shorthand db).
- **scope:** Indicates whether the resource is namespaced or cluster-wide (Namespaced here).
- **schema:** Defines the structure of the resource's specification, including fields like type, version, and storage.

Step 2: Creating a Custom Resource (CR)

With the Database CRD defined, we can create an instance (CR) of it:

```yaml
apiVersion: example.com/v1
kind: Database
metadata:
  name: my-database
spec:
```

```
  type: MySQL
  version: "8.0"
  storage: "10Gi"
```

Explanation:

- type: Specifies the type of the database (e.g., MySQL).
- version: Defines the version of the database (e.g., 8.0).
- storage: Requests 10Gi of storage for the database.

Step 3: Applying the CRD and CR

1. Apply the CRD:
   kubectl apply -f database-crd.yaml
2. Apply the CR:
   kubectl apply -f my-database.yaml

📜 **Step 1: Define a Custom Resource Definition (CRD)**

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: databases.mycompany.com
spec:
  group: mycompany.com
  names:
    kind: Database
    plural: databases
    singular: database
  scope: Namespaced
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                engine:
                  type: string
                version:
                  type: string
                storage:
                  type: string
```

📜 **Step 2: Create a Custom Resource Based on the CRD**

```yaml
apiVersion: mycompany.com/v1
kind: Database
metadata:
  name: my-database
spec:
  engine: postgres
  version: "14"
  storage: "20Gi"
```

Now you can run:
kubectl get databases
And see your custom object!

✅ Custom Controller (Optional)

A controller can be written to monitor and manage Database Custom Resources (CRs). The controller ensures the desired state, such as provisioning database instances or managing backups.
Key Points:

- You can write a controller/operator to watch your custom resource and act on it.
- Example: When a Database CR is created, the controller could:
  - Create a StatefulSet
  - Configure a PVC
  - Set up a Service
  - Update CR status

Common Use Cases for CRDs

1. **Domain-Specific Resources**:
   - Example: Creating resources like Database, Backup, or MonitoringRule to model application-specific abstractions.
2. **Simplify Complex Applications**:
   - Example: Automate lifecycle management for applications like databases or machine-learning models using controllers and CRDs.
3. **Integration with Kubernetes Ecosystem**:
   - Example: Create CRDs to integrate third-party services like cloud databases or monitoring tools.

🌎 Real-Time Use Cases

Use Case	Example
🧩 **Databases as a service**	Create a Database CRD to manage Postgres/MySQL instances

📦 **Custom Deployments**	Create a WebApp resource with auto-scaling and config built-in
🔁 **Backups**	Define a Backup CRD that automates snapshots for PVCs or DBs
📊 **Monitoring/Logging**	Use LogCollector or MetricsJob CRDs to define monitoring pipelines
🧪 **Test environments**	Create TestSuite CRDs that launch temp test pods and clean up after run

🧱 **Types / Components in CRDs**
While CRDs themselves are of type CustomResourceDefinition, **custom resources created using CRDs** can be:

Element	Description
Kind	What the object is (e.g., Database, BackupJob)
Group	API group (e.g., apps.mycompany.com)
Version	API version (e.g., v1, v1alpha1)
Scope	Namespaced or Cluster
Schema	OpenAPI spec to validate CR content

⚙️ **Additional Functionalities**
✅ **Validation with Schema**
- CRDs support **OpenAPI validation**, so you can catch typos or wrong values.

✅ **Versioning Support**
- CRDs can have multiple versions (v1alpha1, v1beta1, v1) for backward compatibility.

✅ **Subresources Support**
- You can enable status and scale subresources:

```
subresources:
  status: {}
  scale:
    specReplicasPath: .spec.replicas
    statusReplicasPath: .status.replicas
```

Best Practices for Using CRDs
1. **Version Management**:
    - Include multiple versions in the CRD (v1, v2beta1) for backward compatibility.
2. **Validation**:

    - Use the openAPIV3Schema to validate CR specifications at the API server level.
3. **Controllers**:
    - Implement controllers to automate the management of CRs and achieve desired states.

14) Plugins in Kubernetes: CNI, CSI, and CRI

Plugins in Kubernetes enhance its functionality by enabling seamless networking, storage, and runtime operations.

1. Container Network Interface (CNI)

CNI is responsible for **managing network connectivity** for Kubernetes pods. It provides a way to configure networking resources dynamically when a container is created.

CNI plugins manage **networking** in Kubernetes. They handle:

- **Pod IP assignment**
- **Pod-to-pod communication**
- **Network policies**

In Kubernetes, plugins are modular components that allow the system to be extended and integrated with various infrastructure providers. The three most important types of plugins are:

🔹 **1. CNI (Container Network Interface)**

- **Purpose: Manages networking for containers.**
- **What it does:**
    - **Assigns IP addresses to Pods.**
    - **Connects Pods to the network.**
    - **Enables communication between Pods and services.**
- **Popular CNI Plugins:**
    - **Calico – Network policy enforcement and routing.**
    - **Flannel – Simple overlay network.**
    - **Weave – Automatic IP address management.**
    - **Cilium – eBPF-based networking and security.**

🔸 **2. CSI (Container Storage Interface)**

- **Purpose: Manages storage for containers.**
- **What it does:**
    - **Allows Kubernetes to use different storage systems (cloud, on-prem).**
    - **Supports dynamic provisioning, attaching, and mounting volumes.**
- **Popular CSI Drivers:**
    - **AWS EBS CSI Driver**
    - **GCE PD CSI Driver**
    - **Ceph CSI**
    - **OpenEBS**

🔧 **3. CRI (Container Runtime Interface)**

- **Purpose: Manages container runtimes.**
- **What it does:**
  - **Allows Kubernetes to communicate with container runtimes like containerd or CRI-O.**
  - **Abstracts the container runtime implementation.**
- **Popular CRI Implementations:**
  - **containerd – Lightweight, production-grade container runtime.**
  - **CRI-O – Kubernetes-native container runtime.**
  - **(Docker was deprecated as a runtime in Kubernetes 1.20+)**

🧩 **Summary Table**

Plugin Type	Interface	Purpose	Examples
CNI	**Networking**	**Pod-to-Pod and external communication**	**Calico, Flannel, Cilium**
CSI	**Storage**	**Persistent volume management**	**AWS EBS, Ceph, OpenEBS**
CRI	**Runtime**	**Container lifecycle management**	**containerd, CRI-O**

Key Features:

1. **Pod Networking**: Establishes communication between pods and services.
2. **IP Address Assignment**: Allocates IPs for containers.
3. **Multi-Cloud Support**: Works across various cloud and on-premise setups.

Feature	Description
🔗 **Pod Networking**	Assigns IP addresses to Pods
🌐 **Inter-Pod Communication**	Enables Pods to talk across nodes
🔐 **Network Policies**	Apply security rules between Pods
🔄 **Pluggable**	Swap different CNI plugins like Calico, Flannel, Cilium, etc.

📄 **Example YAML (Network Policy using CNI)**

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: default
spec:
  podSelector: {}
  policyTypes:
```

```
  - Ingress
  - Egress
```

➡️ This denies all traffic unless explicitly allowed.

Example: Using Flannel as a CNI Plugin

Flannel is a simple CNI implementation for overlay networks.

- Deploy Flannel using:

kubectl apply -f https://raw.githubusercontent.com/flannel-io/flannel/master/Documentation/kube-flannel.yml

🌍 **Real-Time Use Cases:**

Scenario	Role of CNI
🛡️ **Secure traffic**	**Block/allow Pod access**
📡 **Service mesh**	**Used by Istio, Linkerd, etc.**
🌐 **Cloud-native networking**	**Ensure Pods get routable IPs**
🕸️ **Multi-tenant clusters**	**Isolate traffic between teams or apps**

Real-Time Use Case: Flannel creates an overlay network that enables pods on different nodes to communicate seamlessly. It's commonly used in clusters hosted on bare-metal or cloud providers.

🧱 **Popular CNI Plugins:**

- **Calico** – (Networking + Network Policies (very common).) Adds network policies for security and granular control.
- **Weave** – (Peer-to-peer mesh networking.) Offers automatic encryption for cluster networking.
- **Flannel** – Simple overlay networking
- **Cilium** – Powered by eBPF, very fast and secure

2. Container Storage Interface (CSI)

CSI standardizes the integration of external storage systems with Kubernetes. It allows storage vendors to create plugins that manage their resources directly.

CSI plugins manage **storage** in Kubernetes. They allow:

- Dynamic volume provisioning
- Attaching/detaching storage to Pods
- Working with block/file storage

🧠 Think of CSI as **"plug-and-play" storage** for Kubernetes.

🔑 **Key Features:**

1. **Abstraction**: Unifies storage management for diverse systems.
2. **Dynamic Provisioning**: Creates PersistentVolumes (PVs) automatically using StorageClasses.
3. **Cloud and Local Storage**: Supports cloud storage (e.g., AWS EBS, Azure Disk) and on-premise storage.

Feature	Description
💾 **Dynamic Volumes**	Volumes created on demand
🔌 **Vendor Neutral**	Supports EBS, Azure Disk, Ceph, NFS, etc.
🔄 **Attach/Detach Automation**	Kubernetes manages mounting for you
🔐 **Access Modes**	ReadWriteOnce, ReadOnlyMany, etc.

Example: Using AWS EBS CSI Driver
Create a PersistentVolume with the EBS CSI plugin:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-pvc
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 10Gi
```

- Apply the plugin:

kubectl apply -f ebs-csi-plugin.yaml

2. YAML (Persistent Volume Claim using CSI)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

```
storageClassName: standard
```

➡️ The CSI driver (like AWS EBS) handles provisioning and attaching the volume.

🌍 **Real-Time Use Cases:**

Use Case	CSI Role
🗃️ DB Storage	Attach fast block storage to databases
🧪 CI/CD Builds	Temporary storage for jobs
📁 Shared volumes	NFS or GlusterFS volumes shared between Pods
🧠 Data backup/restore	Snapshots & clones with CSI snapshots

Real-Time Use Case: CSI simplifies the management of persistent storage for stateful applications like databases. For example, using CSI, you can dynamically allocate Amazon EBS volumes for pods running MySQL.

🧱 **Popular CSI Drivers:**

- **AWS EBS CSI**
- **GCP PD CSI**
- **Azure Disk/File CSI**
- **CephFS / Rook**
- **Longhorn** (for local clusters)

3. Container Runtime Interface (CRI)

- **Container Runtime Interface (CRI)** is the API that allows Kubernetes to communicate with container runtimes like containerd and CRI-O. It helps Kubernetes run and manage containers efficiently, without being tied to a specific runtime.
- 🔍 Kubernetes doesn't run containers itself — it uses a runtime to do that. **CRI** is the "**translator**" between kubelet and container runtimes.

Key Features:

1. **Runtime Agnostic**: Supports runtimes like containerd, CRI-O, and Docker.
2. **Performance Optimization**: Enables lightweight and efficient container execution.
3. **Unified API**: Offers a consistent API for runtime management.

Feature	Description
⚙️ **Runtime Abstraction**	Kubernetes works with different runtimes (like containerd or CRI-O)
🔌 **Pluggable**	Swap runtimes without changing Kubernetes
🧠 **Simplifies Kubelet**	Kubelet focuses on orchestration, runtime handles execution

🧱 **Security & Performance**	Different runtimes provide different sandboxing (gVisor, Kata, etc.)

📄 **No YAML Needed**
CRI is configured on the **Kubelet side**, not at the Pod level.

Example: Using CRI-O
CRI-O is a lightweight container runtime designed specifically for Kubernetes.

- Configure kubelet to use CRI-O:
  kubelet --container-runtime=remote --runtime-request-url=/var/run/crio/crio.sock

🌎 **Real-Time Use Cases:**

Use Case	CRI Involved
⚡ High-performance apps	Use CRI-O or containerd for speed
🔐 Security-first clusters	Use gVisor/Kata Containers via CRI for sandboxing
🐬 Docker replacement	Kubernetes now uses containerd instead of Docker under the hood

Real-Time Use Case: A Kubernetes cluster running CRI-O improves performance for workloads where runtime efficiency is critical, such as in high-throughput microservices environments.

🧱 **Popular CRI Runtimes:**

- **containerd** – Default runtime for most clusters
- **CRI-O** – Lightweight runtime used by OpenShift
- **gVisor** – Sandboxed, secure containers
- **Kata Containers** – Lightweight VMs for strong isolation

Comparing CNI, CSI, and CRI

Feature	CNI	CSI	CRI
Purpose	**Networking for pods**	**Persistent storage**	**Container runtime**
Examples	**Flannel, Calico**	**AWS EBS, Azure Disk**	**containerd, CRI-O**
Use Case	**Pod communication**	**Dynamic storage allocation**	**Efficient container runtime**

Real-World Scenario
Imagine you're running a Kubernetes cluster for a cloud-based e-commerce application:

1. Use **CNI (Calico)** for pod networking with granular security policies.
2. Use **CSI (AWS EBS)** to allocate storage dynamically for databases.

3. Use **CRI (containerd)** for lightweight container execution.

🧠 **Summary: Plugins at a Glance**

Plugin Type	Purpose	Common Tools
CNI	Networking (Pod IPs, traffic)	Calico, Flannel, Cilium
CSI	Storage (Volumes)	EBS, Ceph, Azure Disk
CRI	Runtime (Container execution)	containerd, CRI-O

15) Service Accounts and RBAC in Kubernetes

1. Service Accounts

- A **Service Account** in Kubernetes is used to provide an **identity for pods** to **authenticate** with the Kubernetes API server and access cluster resources securely.
- It allows pods to securely interact with the Kubernetes API and access resources in the cluster.
- Unlike **regular user accounts** (designed for human interaction), **service accounts** are for non-human entities such as pods.

  🧠 Think of it like:
  "A user ID for an application running inside a Pod."
  It's **not for humans** — it's for **apps, controllers, and workloads** to authenticate and perform actions inside the cluster.

In Kubernetes, Service Accounts and RBAC (Role-Based Access Control) are essential for managing authentication and authorization of workloads and users within the cluster.

🔹 **1. Service Accounts**

A Service Account is an identity used by Pods to interact with the Kubernetes API.

✅ **Purpose:**

- **Allow Pods to authenticate to the Kubernetes API.**
- **Automatically mounted into Pods as a token.**
- **Used by controllers, jobs, and other system components.**

📜 **Example:**

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
```

You can assign it to a Pod like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
```

```
spec:
  serviceAccountName: my-service-account
  containers:
    - name: my-container
      image: my-image
```

🔸 **2. RBAC (Role-Based Access Control)**

RBAC controls what actions users or service accounts can perform on which resources.

✅ **Key Components:**

Component	Description
Role	Defines a set of permissions within a namespace.
ClusterRole	Like Role, but applies cluster-wide.
RoleBinding	Grants a Role to a user or service account in a namespace.
ClusterRoleBinding	Grants a ClusterRole to a user or service account across the cluster.

📜 **Example: Role and RoleBinding**

```
# Role
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
# RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: ServiceAccount
  name: my-service-account
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

🔐 **Summary**

Feature	Service Account	RBAC
Purpose	Identity for Pods	Permissions for identities
Scope	Namespace-scoped	Namespace or cluster-wide

Feature	Service Account	RBAC
Used by	Pods, controllers	Users, service accounts

Key Features of Service Accounts

1. **Secure API Access**: Provides pods with unique credentials to access the Kubernetes API.
2. **Namespace Scope**: Service accounts are scoped to a specific namespace.
3. **Automatic Creation**: Each namespace has a default service account created automatically.
4. **Customization**: You can create custom service accounts with specific permissions.

Real-Time Example of Service Accounts

Creating a Service Account

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-service-account
  namespace: default
```

Apply the service account:

kubectl apply -f service-account.yaml

Using the Service Account in a Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  serviceAccountName: custom-service-account
  containers:
  - name: example-container
    image: busybox
    command: ["sh", "-c", "echo Hello from custom ServiceAccount!"]
```

Explanation:

- **serviceAccountName:** Tells the pod to use the custom-service-account for API access.

📜 **Step 1: Create a Service Account**

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
```

```
  name: my-app-sa
  namespace: dev
```

📜 **Step 2: Use It in a Pod**

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  namespace: dev
spec:
  serviceAccountName: my-app-sa
  containers:
    - name: my-container
      image: myapp:latest
```

This tells Kubernetes:

📣 "Run this Pod using my-app-sa so the app inside can access only what it's allowed to."

🔐 Service Account Tokens

A **JWT token** is created for each SA and mounted in the Pod at:

- /var/run/secrets/kubernetes.io/serviceaccount/token

🧰 Bind SAs to Roles using RBAC

To allow only limited actions:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods-binding
  namespace: dev
subjects:
- kind: ServiceAccount
  name: my-app-sa
  namespace: dev
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Common Use Cases of Service Accounts

1. **Pods Accessing Secrets**:
    - Example: A pod requires access to a Secret for database credentials. A service account enables controlled access.
2. **External Authentication**:

- Example: Applications that need to interact with external services securely using Kubernetes authentication.

🌍 Real-Time Use Cases

Use Case	Explanation
🔐 **Accessing Kubernetes API**	An app wants to list Pods or create Jobs
🔄 **CI/CD Pipelines**	Jenkins or ArgoCD Pods use SAs to deploy to namespaces
🖌️ **Operators & Controllers**	System apps like cert-manager, Prometheus Operator, etc., use SAs to manage resources
🧪 **Test Pods with Limited Permissions**	Use custom SA + RBAC to limit what dev/test workloads can do
📦 **Multi-Tenant Clusters**	Assign different SAs to isolate permissions per team or app

2. Role-Based Access Control (RBAC)

- **Role-Based Access Control (RBAC) in** Kubernetes is a security mechanism that restricts access to cluster resources based on the roles assigned to users, service accounts or groups.
- It provides fine-grained control over access to cluster resources.
- Define roles, role bindings, and service accounts to grant or restrict access to specific resources and actions within the cluster.

It defines:

- **Who** (user, group, service account)
- **Can do What** (get, list, create, delete...)
- **On What** (Pods, Deployments, Secrets, etc.)

Key Features of RBAC

1. **Roles and RoleBindings**:
   - Define permissions (Roles) and associate them with users or service accounts (RoleBindings).
2. **Cluster-Wide and Namespace-Specific**:
   - Use **Roles** for namespace-level permissions.
   - Use **ClusterRoles** for cluster-wide permissions.
3. **Granular Access**:
   - Permissions can be defined for specific resources like pods, deployments, or secrets.

Real-Time Example of RBAC

Creating a Role

```
apiVersion: rbac.authorization.k8s.io/v1
```

```yaml
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

Explanation:

- **apiGroups:** Empty string indicates core Kubernetes resources (e.g., pods, services).
- **resources:** Specifies the resources the role applies to (e.g., pods).
- **verbs:** Defines the actions allowed (e.g., get, list).

Binding the Role to a Service Account

Create a RoleBinding to link the pod-reader role with the custom-service-account:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: custom-service-account
  namespace: default
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Explanation:

- **subjects:** Specifies the entity (service account) to bind to the role.
- **roleRef:** References the pod-reader role created earlier.

📜 **Step 1: Create a Role (Namespace-scoped)**

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev
  name: pod-reader
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list", "watch"]
```

➡️ This Role allows reading Pods in the dev namespace.

📜 **Step 2: Create a RoleBinding (Assign Role to a User)**

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-binding
  namespace: dev
subjects:
  - kind: User
    name: dev-user
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

➡️ This binds the pod-reader Role to the user dev-user.

📜 **Cluster-wide Role Example (ClusterRole + ClusterRoleBinding)**

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: admin-role
rules:
  - apiGroups: ["*"]
    resources: ["*"]
    verbs: ["*"]
```

yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-binding
subjects:
  - kind: User
    name: admin-user
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: admin-role
  apiGroup: rbac.authorization.k8s.io
```

➡️ Gives full admin access to the user admin-user across the entire cluster.

🧱 Types of RBAC Objects

Type	Scope	Purpose
Role	Namespaced	Grants permissions within a namespace
ClusterRole	Cluster-wide	Grants permissions across the cluster (including non-namespaced resources)
RoleBinding	Namespaced	Binds a Role to users/groups/service accounts in a namespace
ClusterRoleBinding	Cluster-wide	Binds a ClusterRole to users/groups/SA across the cluster

Common Use Cases of RBAC

1. **Restricting Access**:
   - Example: Allow developers to only view pods without making changes.
2. **Granular Resource Management**:
   - Example: Grant CI/CD pipelines access to deploy applications but restrict access to secrets.
3. **Service Account Scoping**:
   - Example: Bind service accounts to roles for specific tasks, like accessing only storage-related resources.

🌎 Real-Time Use Cases

Use Case	How RBAC Helps
👤 **Separate Dev/Test/Prod Access**	Developers can access only their namespace
🧪 **CI/CD Pipelines**	Service Accounts get write access to deploy, but not read secrets
🔐 **Security Hardening**	Give only necessary access (Principle of Least Privilege)
🤖 **Automated Apps**	Apps get scoped access (e.g., can read configs but not delete Pods)
🧠 **Audit Trails**	Track who has what access and tighten permissions

⚙️ 6. Additional Functionalities

✅ Aggregation of ClusterRoles

- You can group multiple roles using labels and aggregationRule.
- Useful for combining roles like view, edit, admin.

✅ Default Roles in Kubernetes

- Predefined ClusterRoles like:
  - cluster-admin – Full access
  - admin – Admin access to a namespace
  - edit – Can modify resources in a namespace
  - view – Read-only access

✅ **Integration with Authentication**

- RBAC works **after authentication**. Auth systems (like OIDC, Active Directory, etc.) confirm identity → RBAC decides what they're allowed to do.

Summary

Feature	Service Account	RBAC
Purpose	Authenticate pods to access Kubernetes APIs.	Define and enforce resource permissions.
Scope	Namespace-specific	Namespace or cluster-wide
Real Time Use Case	Pod accessing external APIs	Developer with read-only access to pods

16) Horizontal Pod Autoscaler (HPA) and Metrics Server in Kubernetes

Horizontal Pod Autoscaler (HPA)

- The **Horizontal Pod Autoscaler (HPA)** automatically adjusts the **number of Pods** in a Deployment, ReplicaSet, or StatefulSet based on observed **CPU utilization** or **other custom metrics**.
- It helps maintain application performance during traffic spikes while optimizing resource usage during idle periods.

In Kubernetes, the Horizontal Pod Autoscaler (HPA) and the Metrics Server work together to enable automatic scaling of Pods based on observed resource usage like CPU or memory.

🔹 **1. Horizontal Pod Autoscaler (HPA)**

The HPA automatically adjusts the number of Pods in a Deployment, ReplicaSet, or StatefulSet based on CPU utilization, memory usage, or custom metrics.

✅ **Key Features:**

- **Scales Pods horizontally (adds/removes replicas).**
- **Uses metrics like CPU, memory, or custom metrics.**
- **Helps maintain performance and optimize resource usage.**

📜 **Example HPA YAML:**

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
```

```
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

This configuration keeps CPU usage around 50% by scaling Pods between 2 and 10 replicas.

🔸 **2. Metrics Server**

The Metrics Server is a cluster-wide aggregator of resource usage data. It collects metrics like:

- **CPU and memory usage of Pods and Nodes.**
- **These metrics are used by HPA and other components.**

✅ **Key Points:**

- **Lightweight and scalable.**
- **Does not store historical data (for that, use Prometheus).**
- **Must be installed separately in most Kubernetes clusters.**

📜 **Example command to view metrics:**

```
kubectl top pod
kubectl top node
```

🔁 **How They Work Together**

1. **Metrics Server collects real-time resource usage.**
2. **HPA queries the Metrics Server.**
3. **Based on the metrics, HPA adjusts the number of Pods.**

📊 **Visual Diagram Offer**

Would you like a diagram showing how HPA and Metrics Server interact in Kubernetes? I can generate one for you!

Key Features of HPA

1. **Dynamic Scaling**: Automatically scales pods up or down based on resource usage.
2. **Metrics Integration**: Uses metrics like CPU, memory, or custom application metrics.
3. **Custom Metrics Support**: Works with metrics from Prometheus, Datadog, or other sources using custom adapters.
4. **Minimum and Maximum Pods**: Sets boundaries for scaling.
5. **Event-Driven**: Scales based on real-time metrics.
6. **Configurable Thresholds:** Set custom thresholds to determine when to scale

Real-Time Example of HPA

Step 1: Create a Deployment
Start with a deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
      - name: example-container
        image: nginx
        resources:
          requests:
            cpu: 200m
          limits:
            cpu: 400m
```

Step 2: Enable HPA
Create an HPA resource to auto-scale the deployment based on CPU usage:

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: example-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: example-deployment
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Explanation:

- **scaleTargetRef:** Links the HPA to the example-deployment.
- **minReplicas:** Ensures a minimum of 2 replicas are running at all times.
- **maxReplicas:** Caps the scaling to 10 replicas.
- **averageUtilization:** Targets 50% CPU usage. If CPU usage exceeds 50%, the HPA scales up the pods.

Applying the HPA

1. Apply the deployment:
   kubectl apply -f deployment.yaml
2. Apply the HPA:
   kubectl apply -f hpa.yaml
3. Monitor scaling:
   kubectl get hpa

2. Let's create a Horizontal Pod Autoscaler for a **Deployment** based on **CPU usage**.

📜 **Step 1: Define the Deployment**

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: myapp:latest
          resources:
            requests:
              cpu: "200m"
            limits:
              cpu: "500m"
```

- Here, the app starts with **1 replica**, and we've defined CPU **requests** and **limits** for the container.

📜 **Step 2: Define the Horizontal Pod Autoscaler (HPA)**

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
```

```yaml
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Key points of the HPA YAML:

- **scaleTargetRef**: Points to the Deployment (the object we want to scale).
- **minReplicas**: The minimum number of Pods to maintain (1 in this case).
- **maxReplicas**: The maximum number of Pods to scale to (10 in this case).
- **metrics**: Defines that we are scaling based on CPU utilization. In this example, the target CPU utilization is **50%**.

🌎 Real-Time Use Cases of HPA

Use Case	Explanation
🌐 **Traffic Spikes**	Automatically scale up the number of Pods when website traffic increases.
👷 **Resource-Intensive Jobs**	Scale Pods for applications like data processing or AI training when CPU or memory usage increases.
🛠️ **DevOps CI/CD Pipelines**	Ensure enough resources for continuous integration and deployment tasks, scaling Pods as required.
🏗️ **Microservices**	In microservices architectures, each service can scale independently based on its needs (e.g., scaling a payment service during high load).
📈 **E-commerce Sales**	Automatically scale Pods for the e-commerce platform during promotions, ensuring that the system can handle increased demand.

🧱 Types of Metrics in HPA

🧮 1. Resource Metrics (CPU & Memory)

- **CPU Utilization**: Autoscale based on the CPU usage of Pods.

- **Memory Usage**: Autoscale based on the memory usage of Pods.

🧩 **2. Custom Metrics (v2)**

- Allows scaling based on application-specific metrics (e.g., number of requests, queue length, latency).
- For custom metrics, you'll need **Prometheus** or another monitoring solution to expose those metrics.

📊 **3. External Metrics (v2)**

- Scale based on metrics from outside Kubernetes, such as queue length in a messaging system (e.g., RabbitMQ), or request counts from an external API.
- This requires integration with external monitoring systems.

⚙️ Additional Functionalities of HPA

🔄 **Scaling Behavior**

Kubernetes HPA also provides more fine-grained control over the scaling behavior. For example:

- **Cooldown Periods**: Time to wait before scaling up or down again after a change.
- **Scaling Stabilization**: Helps prevent "flapping" (constant scaling up and down).

Metrics Server

The **Metrics Server** collects resource usage data (CPU and memory) from nodes and pods within a Kubernetes cluster. This data is essential for HPA to make scaling decisions.

Key Features of Metrics Server

1. **Lightweight Resource Monitoring**: Gathers real-time metrics from the kubelet.
2. **Integration with HPA**: Provides CPU and memory metrics for autoscaling.
3. **Default Kubernetes Add-On**: Installed as part of most Kubernetes distributions.

Installing Metrics Server

1. Deploy the Metrics Server:
   `kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml`
2. Verify the installation:
   `kubectl get apiservices | grep metrics`

Common Use Cases of Metrics Server

1. Provides resource usage data for tools like HPA.
2. Monitors pod and node performance metrics using kubectl top:
   `kubectl top pod`
   `kubectl top node`

Common Use Cases of HPA

1. **Web Applications**:
   - Example: Scale an e-commerce app to handle traffic spikes during sales.
2. **Processing Pipelines**:
   - Example: Scale data processing pods during peak workloads.
3. **APIs**:

- Example: Scale API servers dynamically based on the number of incoming requests.

Key Considerations

- The Metrics Server must be installed and running for HPA to work with resource metrics.
- Use custom metrics for advanced autoscaling logic (e.g., scaling based on request count or latency).
- Configure resource requests and limits for all containers to avoid unpredictable scaling.

Real-World Scenario

Consider a cluster hosting an API backend with unpredictable traffic:

1. Deploy Metrics Server to collect CPU and memory metrics.
2. Use HPA to scale the API backend deployment from 2 to 20 pods during traffic spikes.
3. Monitor scaling decisions using:
   kubectl get hpa

17) Cluster Autoscaling

Cluster Autoscaling

- Cluster Autoscaling in Kubernetes dynamically adjusts the number of nodes in a cluster based on resource demands This helps optimize resource utilization and costs.

How Cluster Autoscaler Works

1. **Scale Up** – When pods cannot be scheduled due to insufficient resources, the Cluster Autoscaler **adds nodes**.
2. **Scale Down** – When nodes are underutilized for a certain period, they are removed.
3. **Node Group Integration** – Integrate with cloud-specific autoscaling groups to manage instances dynamically.

🔹 Cluster Autoscaling in Kubernetes

Cluster Autoscaling is a feature in Kubernetes that automatically adjusts the number of **nodes** in your cluster based on the resource demands of your workloads.

🔧 How It Works

The **Cluster Autoscaler** monitors the cluster and performs two main actions:

1. **Scale Out (Add Nodes)**:
   - When Pods are **pending** due to insufficient resources (CPU/memory), the autoscaler adds more nodes to the cluster.
2. **Scale In (Remove Nodes)**:
   - When nodes are **underutilized** and their Pods can be moved to other nodes, the autoscaler removes those nodes to save resources.

✅ Key Features

Feature	Description
Automatic Node Management	Adds/removes nodes based on demand.
Works with HPA	Complements Horizontal Pod Autoscaler.
Cloud Provider Integration	Works with AWS, GCP, Azure, etc.
Node Group Awareness	Can manage multiple node groups with different instance types.

📜 **Example Use Case**

- You have a Deployment with HPA that scales Pods from 2 to 20.
- If the current nodes can't handle 20 Pods, the Cluster Autoscaler will **add more nodes**.
- When traffic drops and fewer Pods are needed, it will **remove unused nodes**.

🔒 **Requirements**

- Must be deployed as a **separate component** (usually as a Deployment in the kube-system namespace).
- Requires **cloud provider support** for dynamic node provisioning.
- Needs proper IAM roles and permissions (e.g., to create/delete VMs).

📌 **Best Practices**

- Use with **HPA** for full auto-scaling (Pods + Nodes).
- Set **resource requests/limits** on Pods to help the autoscaler make decisions.
- Monitor with tools like **Prometheus**, **Grafana**, or **Cloud provider dashboards**.

Would you like a **diagram** showing how Cluster Autoscaler interacts with HPA and the cloud provider?

Key Features of Cluster Autoscaler

1. **Automatic Scaling** – Adds or removes nodes based on pod scheduling requirements.
2. **Cost Optimization** – Adds or removes nodes in **node pools** based on pending pods or underutilized nodes to save costs in cloud environments.
3. **Works with Node Groups** – Integrates with cloud providers like AWS, Azure, and GCP.
4. **Improves High Availability** – Ensures pods have enough compute resources to run smoothly.
5. **Pod Scheduling Awareness** – Reacts to unallocated or pending pods by provisioning nodes.

📄 Example YAML Code for Cluster Autoscaler

Step 1: Cluster Autoscaler Deployment YAML

The Cluster Autoscaler is typically deployed as a **Deployment** or **Pod** in your cluster.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cluster-autoscaler
```

```
  namespace: kube-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cluster-autoscaler
  template:
    metadata:
      labels:
        app: cluster-autoscaler
    spec:
      containers:
        - name: cluster-autoscaler
          image: k8s.gcr.io/cluster-autoscaler:v1.21.0
          command:
            - ./cluster-autoscaler
            - --v=4
            - --cloud-provider=gce
            - --nodes=1:10:k8s-nodepool
            - --scale-down-enabled=true
            - --scale-down-unneeded-time=10m
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 200m
              memory: 256Mi
```

Key Points:

- **cloud-provider=gce**: Specifies that the Cluster Autoscaler is set to run on **Google Cloud (GCE)**.
- **--nodes=1:10:k8s-nodepool**: Specifies the **node pool** (k8s-nodepool) and sets the **minimum** (1 node) and **maximum** (10 nodes) number of nodes.
- **--scale-down-enabled=true**: Enables automatic scaling down of unused nodes.
- **--scale-down-unneeded-time=10m**: A node is removed only if it's unused for at least 10 minutes.

Step 2: Example of ConfigMap (Optional)

Cluster Autoscaler often uses a ConfigMap to configure specific settings. Here's how you can use a ConfigMap to configure the behavior of Cluster Autoscaler:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-autoscaler-config
  namespace: kube-system
```

```
data:
  cluster-autoscaler.cfg: |
    scale-down-unneeded-time=10m
    scale-down-delay-after-add=10m
    scale-up-from-zero=true
```

- **scale-down-unneeded-time=10m**: Ensures that a node is only scaled down after being unused for 10 minutes.
- **scale-up-from-zero=true**: Allows scaling the cluster from zero nodes if required.

Pod not schedulable.

1. **Node Selector - will update details in pod yml**
2. **Node Affinity - will update details in pod yml**
3. **Taints - will update configure in nodes**
4. **Tolerations**

18) Node Selector

✅ Node Selector:

- A **Node Selector** in Kubernetes is **used to control which nodes a pod can be scheduled on** by **specifying key-value pair labels** that must match labels on the target nodes.
- If no node matches, the Pod will not be scheduled.

🧩 How It Works:

1. **Nodes are labeled** with key-value pairs (e.g., disktype=ssd).
2. In the **Pod spec**, you use nodeSelector to define the labels that a node must have.
3. Kubernetes **schedules the pod only on nodes** that match the specified labels.

🔹 Node Selector in Kubernetes

A **Node Selector** is the simplest way to **assign Pods to specific nodes** in a Kubernetes cluster based on **labels**.

✅ Purpose

- Control **where** a Pod runs.
- Ensure Pods are scheduled only on nodes that meet specific criteria (e.g., hardware, region, OS).

🔧 How It Works

1. **Label your nodes** with key-value pairs.
2. Use nodeSelector in your Pod spec to match those labels.

📄 Example

Step 1: Label a node

kubectl label nodes node-1 disktype=ssd

Step 2: Use nodeSelector in your Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: ssd-pod
spec:
  containers:
    - name: my-container
      image: nginx
  nodeSelector:
    disktype: ssd
```

This Pod will only be scheduled on nodes labeled with disktype=ssd.

📌 **Limitations**

- **Exact match only**: No expressions or complex logic.
- **Static**: You must manually label nodes and update selectors.

🔄 **Alternatives for More Flexibility**

Feature	Description
Node Affinity	More expressive (supports operators like In, NotIn, Exists).
Taints and Tolerations	Prevent Pods from being scheduled unless they tolerate the node's taint.
Pod Affinity/Anti-Affinity	Schedule Pods based on other Pods' locations.

Would you like a **diagram** showing how node selectors work in a cluster?

📄 Example YAML for Node Selector

The nodeSelector field in a Pod specification is used to define which nodes the Pod can be scheduled on.

1. Example: Node Selector to Schedule Pod on Specific Nodes

This example shows how to use a Node Selector to schedule a Pod on nodes with a specific label (e.g., zone=us-west-1).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  nodeSelector:
    zone: us-west-1
  containers:
    - name: my-container
      image: nginx
```

Explanation:

- **nodeSelector**: A key-value pair where the key is zone and the value is us-west-1.
- This configuration ensures that the Pod will only be scheduled on nodes that are labeled with zone=us-west-1.

Example of Node Labeling:

- For this nodeSelector to work, the nodes in your cluster need to have a matching label, like:
  kubectl label nodes <node-name> zone=us-west-1

This command adds the label zone=us-west-1 to the specified node. Now, the Pod in the previous YAML can be scheduled on this node.

2. Example with Multiple Node Selectors:
You can specify multiple label selectors in the nodeSelector field, but all conditions must match exactly. Here's an example:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  nodeSelector:
    zone: us-west-1
    hardware: gpu
  containers:
    - name: my-container
      image: nginx
```

In this case, Kubernetes will only schedule the Pod on nodes that have both zone=us-west-1 **and** hardware=gpu labels.

Differences Between Node Selector and Node Affinity

- **Node Selector**:
  - Simple key-value pair.
  - No flexibility or complex rule-based matching.
  - Only exact matches are allowed.
- **Node Affinity**:
  - More complex and flexible.
  - Can use operators like In, NotIn, Exists, and DoesNotExist.
  - Allows **preferred** and **required** rules for scheduling.

For more complex scheduling needs, **Node Affinity** or **Node Anti-Affinity** might be a better choice.

🛠️ Other Functionalities and Considerations

Taints and Tolerations with Node Selector

- **Node Selector** can be combined with **taints and tolerations** for advanced scheduling.
- **Taints** are applied to nodes to prevent Pods from being scheduled on certain nodes unless they have a matching **toleration**.

For example, a node may have a taint like this:
kubectl taint nodes <node-name> dedicated=high-performance:NoSchedule

Then, the Pod must include a toleration to be scheduled on that node:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  nodeSelector:
    hardware: gpu
  tolerations:
    - key: "dedicated"
      operator: "Equal"
      value: "high-performance"
      effect: "NoSchedule"
  containers:
    - name: my-container
      image: nginx
```

This configuration ensures the Pod is only scheduled on nodes labeled with hardware=gpu and can tolerate the dedicated=high-performance:NoSchedule taint.

✅ **Use Cases:**

- Ensuring pods run on nodes with specific hardware (e.g., GPU, SSD).
- Separating workloads by environment (e.g., env=prod vs. env=dev).
- Managing licensing or resource constraints.

⚠️ **Limitations:**

- Only supports **exact match** (no expressions or complex logic).
- For more advanced scheduling, use **node affinity** instead of nodeSelector.

19) Node Affinity and Anti-Affinity

✅ **Node Affinity and Anti-Affinity**

- **Node Affinity** and **Node Anti-Affinity** are advanced ways to control **pod scheduling** by specifying rules about the nodes a pod should or should not be placed on, based on node labels.

🔹 Node Affinity and Anti-Affinity in Kubernetes

Node Affinity and **Anti-Affinity** are advanced scheduling features that allow you to control **where Pods are placed** in a more expressive way than nodeSelector.

✅ 1. Node Affinity

Node Affinity lets you specify rules about which **nodes a Pod can be scheduled on**, based on **node labels**.

🔧 Types of Node Affinity:

- requiredDuringSchedulingIgnoredDuringExecution: **Hard requirement** — Pod won't be scheduled unless the rule is met.
- preferredDuringSchedulingIgnoredDuringExecution: **Soft preference** — Scheduler tries to meet the rule but can fall back.

📜 Example:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: disktype
          operator: In
          values:
          - ssd
```

This Pod will only run on nodes labeled with disktype=ssd.

🚫 2. Node Anti-Affinity

Node Anti-Affinity prevents Pods from being scheduled on nodes with **specific labels**, often used to **spread Pods across nodes**.

📜 Example:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: zone
          operator: NotIn
          values:
          - zone-a
```

This Pod will **not** be scheduled on nodes in zone-a.

🔄 Comparison with nodeSelector

Feature	nodeSelector	Node Affinity
Expressiveness	Simple key-value	Supports operators like In, NotIn, Exists
Flexibility	Static	Supports soft and hard rules
Use Cases	Basic scheduling	Advanced placement strategies

Would you like a **diagram** showing how Node Affinity and Anti-Affinity affect Pod placement across nodes? I can generate one for you!

1. Node Affinity:

Node Affinity allows Kubernetes to schedule pods on **specific nodes with specific labels**, similar to Node Selector but offers more flexibility
(e.g., "run this Pod on a node labeled as zone=us-west-1").

✳ Types of Node Affinity:

1. **requiredDuringSchedulingIgnoredDuringExecution**
   - **Hard requirement:** Pod must be scheduled on matching nodes.
   - If no match, pod remains unscheduled**.**
2. **preferredDuringSchedulingIgnoredDuringExecution**
   - **Soft requirement:** Pod is scheduled on preferred nodes if possible, otherwise anywhere.

Example YAML for Node Affinity

This example demonstrates how to use node affinity to ensure that a Pod is scheduled only on nodes with a specific label.

Example: Node Affinity to Schedule Pod on Specific Nodes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "zone"
                operator: In
                values:
                  - us-west-1
  containers:
    - name: my-container
      image: nginx
```

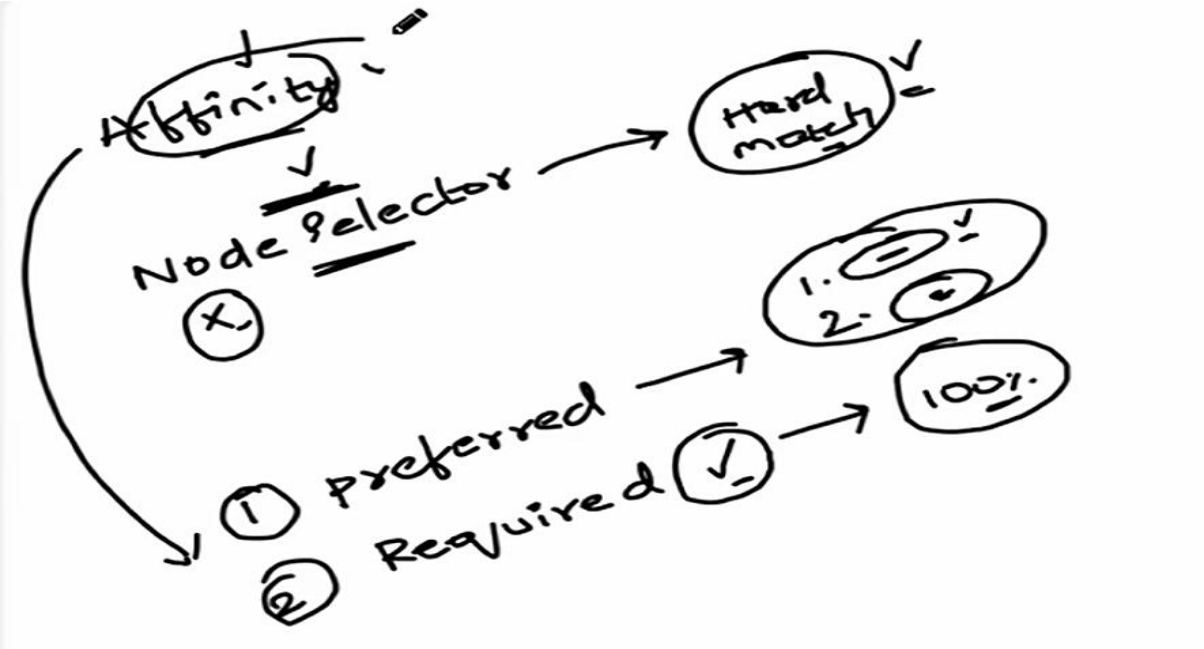
Explanation:

- nodeSelectorTerms: Specifies the conditions that the node must meet.
- key: "zone": The node must have a label with the key zone.
- operator: In: The value of the node's zone label must be in the specified set (here, "us-west-1").

This configuration ensures that the Pod is scheduled only on nodes labeled with zone=us-west-1.

2. Node Anti-Affinity:

Node Anti-Affinity **prevents** pods from being scheduled on nodes with specific labels. It's useful for **spreading pods across nodes** or **avoiding certain node types**. (e.g., "don't run this Pod on a node labeled as zone=us-west-1").



Types of Node Anti-Affinity

Similar to node affinity, anti-affinity has two types:

1. **requiredDuringSchedulingIgnoredDuringExecution**:
   - **Strict enforcement**: Pods **will not be scheduled** unless the anti-affinity rule is satisfied.
   - Example: "I do not want to run this Pod on a node with zone=us-west-1."
2. **preferredDuringSchedulingIgnoredDuringExecution**:
   - The scheduler will try to find a node that meets the anti-affinity rules, but if it cannot find one, the pod will still be scheduled on a node that does not meet the rules.
   - Example: "I prefer not to run this Pod on nodes in the us-west-1 zone, but it's not strictly required."

Example YAML for Anti-Affinity

Node Anti-Affinity Example:

This example shows how to use **node anti-affinity** to avoid scheduling Pods on nodes with specific labels.

Example: Node Anti-Affinity to Avoid Certain Nodes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "zone"
                operator: NotIn
                values:
                  - us-west-1
  containers:
    - name: my-container
      image: nginx
```

Explanation:

- **operator: NotIn**: Specifies that the zone label on the node should **not** be one of the listed values.
- **values: ["us-west-1"]**: The Pod should **not** be scheduled on nodes labeled zone=us-west-1.

This configuration ensures that the Pod **won't** be scheduled on any nodes in the "us-west-1" zone.

⚙️ Other Functionalities of Node Affinity and Anti-Affinity

Advanced Matching

- **matchExpressions**: You can combine multiple conditions to refine scheduling decisions.
    - **Example**: "Pod must be scheduled on a node labeled with zone=us-west-1 and environment=production."
- **matchFields**: You can use **field selectors** to match other attributes, like node name or status.
    - **Example**: "Pod should only be scheduled on nodes with status=Ready."

Combining Affinity with Taints and Tolerations

- You can combine **Node Affinity** with **taints and tolerations** for even more fine-grained control over Pod scheduling.

20) Pod Affinity and Anti-Affinity

- Pod affinity and anti-affinity control **how pods are scheduled in relation to other pods**, based on labels. These mechanisms help distribute workloads efficiently or ensure certain pods run closer together for performance optimization.

- These mechanisms allow fine-grained control over workload placement to optimize for performance, fault tolerance, or availability

✅ **Pod Affinity:**

- **Definition**: Pod Affinity rules ensures that a pod is scheduled **on the same node (or zone) or close to** other specified pods based on labels.

  🔹 **Use Case**: Used for applications that benefit from **low latency** or **better communication** (e.g., microservices that frequently interact).

❌ **Pod Anti-Affinity:**

- **Definition**: Pod Anti-Affinity rules **prevent** pods from being scheduled **on the same node (or zone) as other specified pods** based on labels. This is useful for **high availability** or to reduce resource contention.

  🔹 **Use Case**: Helps **spread workloads** across nodes to improve **availability and redundancy** (e.g., replicas of an application should not be placed on the same node to prevent downtime).

🔹 **Pod Affinity and Anti-Affinity in Kubernetes**

Pod Affinity and **Pod Anti-Affinity** are advanced scheduling rules that control **how Pods are placed relative to other Pods** in the cluster.

✅ **1. Pod Affinity**

Pod Affinity allows you to schedule a Pod **on the same node (or topology domain)** as other Pods with specific labels.

🔧 **Use Cases:**

- Co-locating microservices that communicate frequently.
- Ensuring Pods that share data are on the same node or zone.

📜 **Example:**

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app
          operator: In
          values:
          - frontend
      topologyKey: "kubernetes.io/hostname"
```

This Pod will be scheduled on the **same node** as Pods labeled app=frontend.

🚫 **2. Pod Anti-Affinity**

Pod Anti-Affinity prevents Pods from being scheduled **on the same node (or zone)** as other Pods with specific labels.

🔧 **Use Cases:**

- High availability (spread replicas across nodes).
- Avoiding resource contention.

📜 **Example:**

```
affinity:
 podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
     matchExpressions:
     - key: app
       operator: In
       values:
       - frontend
    topologyKey: "kubernetes.io/hostname"
```

This Pod will **not** be scheduled on the same node as other Pods labeled app=frontend.

🔄 **Comparison Table**

Feature	Pod Affinity	Pod Anti-Affinity
Purpose	Co-locate Pods	Spread Pods apart
Based on	Other Pods' labels	Other Pods' labels
Topology	Node, zone, region	Node, zone, region
Use Case	Low latency, shared cache	High availability, fault tolerance

---

Would you like a **diagram** showing how Pod Affinity and Anti-Affinity affect Pod placement across nodes? I can generate one for you!

🔧 **YAML Examples**

✅ **Pod Affinity Example: Place Pods together**

This Pod wants to run on a node **where another Pod with label app=nginx** is already running.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                  - nginx
          topologyKey: "kubernetes.io/hostname"
```

```
  containers:
    - name: web
      image: nginx
```

📝 **Explanation:**

- podAffinity: Means "I want to be close to..."
- requiredDuringSchedulingIgnoredDuringExecution: A **hard rule** for scheduling.
- topologyKey: kubernetes.io/hostname: Means Pods should be on the **same node**.

❌ **Pod Anti-Affinity Example: Place Pods apart**

This Pod should **not** run on the same node as another Pod with app=nginx.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-app
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: app
                operator: In
                values:
                  - nginx
          topologyKey: "kubernetes.io/hostname"
  containers:
    - name: web
      image: nginx
```

📝 **Explanation:**

- podAntiAffinity: Means "I want to stay away from..."
- Keeps high-availability workloads separated across nodes.

You can use one or both in a Pod spec depending on how strict you want the rule to be.

🔑 **Key Concepts**

- **topologyKey**: Defines the domain over which the rule applies (e.g., node, zone).
  - Common values: kubernetes.io/hostname, topology.kubernetes.io/zone
- **requiredDuringSchedulingIgnoredDuringExecution**:
  - **Hard rule**: Pod won't be scheduled if the condition isn't met.
- **preferredDuringSchedulingIgnoredDuringExecution**:
  - **Soft rule**: Scheduler tries to meet it, but may ignore if necessary.

21) Taints and Tolerations

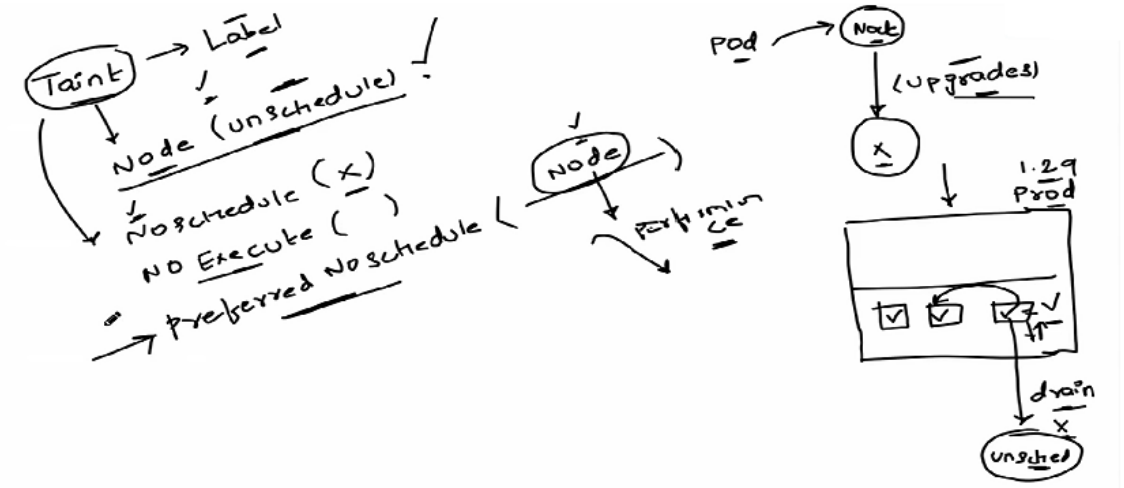
Taints and tolerations are **scheduling mechanisms** in Kubernetes that **control which pods can or cannot run on specific nodes**. They help prevent pods from being scheduled on inappropriate nodes while allowing exceptions when necessary.

Taints and Tolerations in Kubernetes are mechanisms used to **control which Pods can be scheduled on which Nodes**. They are essentially the inverse of affinity rules. Instead of saying "I want to run near this," you're saying "stay away unless you're explicitly allowed."

✅ Taints:

- A **taint** is applied to a **Node** and **prevents Pods** from being scheduled on it unless those Pods have a matching toleration.
- It tells Kubernetes:
  "**Don't schedule any Pod here** unless it can tolerate this taint."

- 🔹 **Use Case**: Prevent regular workloads from running on specialized nodes (e.g., GPU nodes, storage-heavy nodes).



🔹 Taints and Tolerations in Kubernetes

Taints and Tolerations work together to ensure that **Pods are only scheduled onto appropriate nodes**. They are used to **repel** Pods from certain nodes unless those Pods explicitly **tolerate** the taint.

✅ 1. Taints

A **taint** is applied to a node and marks it as **unsuitable** for general workloads unless a Pod has a matching **toleration**.

📜 Example: Add a taint to a node

kubectl taint nodes node1 key=value:NoSchedule
This means: *"Do not schedule any Pod on node1 unless it tolerates this taint."*

✅ 2. Tolerations

A **toleration** is applied to a Pod and allows it to be scheduled on nodes with matching taints.

📜 Example: Pod with a toleration

```
apiVersion: v1
kind: Pod
metadata:
  name: tolerant-pod
spec:
  containers:
    - name: my-container
      image: nginx
  tolerations:
    - key: "key"
      operator: "Equal"
      value: "value"
      effect: "NoSchedule"
```

This Pod can now be scheduled on nodes tainted with key=value:NoSchedule.

🔄 Taint Effects

Effect	Description
NoSchedule	Pod will **not** be scheduled unless it tolerates the taint.
PreferNoSchedule	Scheduler **tries to avoid** placing the Pod on the node.
NoExecute	Existing Pods are **evicted** unless they tolerate the taint.

📌 Use Cases

- **Dedicated nodes** for specific workloads (e.g., GPU, high-memory).
- **Node isolation** for security or compliance.
- **Graceful eviction** of Pods during maintenance.

Would you like a **diagram** showing how taints and tolerations interact during scheduling?

📊 Types of Taints (by Effect)

Effect	What It Does
NoSchedule	❌ Don’t schedule Pods here unless they tolerate the taint.
PreferNoSchedule	☝️ Try to avoid scheduling here, but not strictly required.
NoExecute	⛔ Don’t schedule **and evict existing Pods** that don’t tolerate it.

✅ Tolerations:

- **Definition**: **Tolerations** are applied to Pods to allow them to be scheduled on Nodes with matching taints.
- It tells Kubernetes:
  "This Pod is **okay to run on** a node with this specific taint."
  So, taints repel Pods, and tolerations let Pods tolerate (i.e., ignore) that taint.

- 🔹 **Use Case**: Allow certain workloads, like system-critical services, to run on specific nodes despite taints.

A **toleration** is added to a Pod to let it **bypass a Node's taint**.

🔑 **Key Features**

Feature	Description
🚫 Prevent unwanted scheduling	Taints can block certain Pods from running on sensitive or special nodes.
✅ Allow only specific Pods	Only Pods with matching tolerations can run on tainted nodes.
🎯 Node-level control	Taints are applied per node.

🔧 **Example: YAML + Commands**

🧱 **Step 1: Add a taint to a node**

kubectl taint nodes <node-name> key=value:NoSchedule

📝 Example:

kubectl taint nodes worker1 env=prod:NoSchedule

This tells Kubernetes: ❌ "Don't schedule Pods here unless they tolerate env=prod."

📦 **Step 2: Pod with a matching toleration**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-prod-pod
spec:
  tolerations:
    - key: "env"
      operator: "Equal"
      value: "prod"
      effect: "NoSchedule"
  containers:
    - name: nginx
      image: nginx
```

✅ This Pod says: “I tolerate the env=prod taint, so it's okay to schedule me there.”

📜 **More Examples**
⛔ **NoExecute Taint Example**
kubectl taint nodes worker2 key=value:NoExecute

Pods **without toleration** are evicted immediately.

Toleration with Duration

```
tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoExecute"
    tolerationSeconds: 60
```

⏱️ This Pod will be evicted **after 60 seconds** if the taint is added.

📘 **Summary Table**

Component	Description
Taint	Applied to a node to reject unwanted Pods
Toleration	Applied to a Pod to allow it to run on tainted nodes
Effect Types	NoSchedule, PreferNoSchedule, NoExecute
Use Cases	Node isolation, special workloads, critical systems
Commands	kubectl taint nodes for tainting, tolerations go in Pod YAML

💡 **Pro Tips**

- You can **remove a taint** using this command:
  kubectl taint nodes <node-name> env=prod:NoSchedule-
- Taints and tolerations don’t **guarantee** Pod placement — they just **allow** or **prevent** it.
- Combine with **nodeSelector** or **nodeAffinity** for stricter control.

22) Network Policies

- **Network Policies** in Kubernetes define rules for controlling **ingress (incoming)** and **egress (outgoing)** traffic between pods. They help **secure communication** inside the cluster by restricting or allowing connections based on labels, namespaces, and IP blocks.

🧠 In simple words:
It's like a **firewall for Pods** — you allow or deny traffic **based on labels, ports, and namespaces**.
By default, **all Pods can communicate** with each other. Network Policies let you **lock that down**.

🔹 **Network Policies in Kubernetes**

Network Policies in Kubernetes are used to **control traffic flow** between Pods and/or namespaces. They act like **firewall rules** for Pods, allowing you to define **which Pods can communicate with each other** and with external endpoints.

✅ **Purpose**

- Improve **security** by restricting unwanted traffic.
- Enforce **least privilege** networking.
- Control **ingress** (incoming) and **egress** (outgoing) traffic.

🔧 **Requirements**

- Your cluster must use a **CNI plugin** that supports Network Policies (e.g., Calico, Cilium, Weave).

📜 **Example: Deny All Ingress by Default**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all-ingress
  namespace: default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

This policy denies all incoming traffic to Pods in the default namespace.

📜 **Example: Allow Ingress from Specific App**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-frontend-to-backend
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
  - from:
    - podSelector:
        matchLabels:
```

```
      app: frontend
  policyTypes:
  - Ingress
```
This allows only Pods labeled app=frontend to send traffic to Pods labeled app=backend.

📜 **Example: Allow Egress to External DNS**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-dns
  namespace: default
spec:
  podSelector: {}
  egress:
  - to:
    - ipBlock:
        cidr: 8.8.8.8/32
  policyTypes:
  - Egress
```
This allows Pods to access Google DNS (8.8.8.8).

📌 **Key Concepts**

Term	Description
podSelector	Selects which Pods the policy applies to.
ingress	Rules for incoming traffic.
egress	Rules for outgoing traffic.
policyTypes	Specifies whether the policy applies to ingress, egress, or both.

Would you like a **diagram** showing how Network Policies control traffic between Pods and namespaces?

Key Features of Network Policies

1. **Pod-to-Pod Traffic Control** – Define which pods can communicate with each other.
2. **Namespace-Level Isolation** – Restrict traffic between namespaces for better security.
3. **Ingress and Egress Rules** – Control incoming and outgoing network traffic.
4. **Label-Based Rules** – Apply policies dynamically using pod labels.
5. **Works with CNI Plugins** – Requires a networking provider that supports policies (e.g., Calico, Cilium, Weave).

How Network Policies Work

1. **Pods are labeled** to identify their traffic rules.
2. **Network Policies define allowed or restricted communication** between labeled pods.
3. **Only pods matching the policy can send or receive traffic**, ensuring controlled access.

🧩 Key Features

Feature	Description
🔒 Traffic control	Restrict **incoming (ingress)** and **outgoing (egress)** network traffic
🏷️ Label-based	Rules apply to Pods using **labels**
🌐 Protocols & Ports	You can filter traffic by **port** and **protocol** (TCP/UDP)
📦 Namespace-aware	You can restrict traffic based on **namespaces**
✅ Opt-in model	Only applies to Pods selected by podSelector
⚙️ Requires CNI	Only works if your CNI plugin supports it (e.g., Calico, Cilium, etc.)

🔀 Types of Network Policies

Type	Description
Ingress	Controls incoming traffic to a Pod
Egress	Controls outgoing traffic from a Pod
Both	You can define both at once

📄 Example YAMLs

✅ a. Allow traffic from specific Pods (Ingress)

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-frontend
spec:
  podSelector:
    matchLabels:
      app: my-backend
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: frontend
```

🔐 Only Pods with label role: frontend can access Pods labeled app: my-backend.

✅ b. Allow traffic only from a specific Namespace

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: allow-from-namespace
spec:
  podSelector:
    matchLabels:
      app: secure-api
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              team: dev
```

🔐 Only Pods from the namespace with label team: dev can access the secure-api Pods.

✅ **c. Allow outgoing traffic only to a specific IP (Egress)**

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-egress
spec:
  podSelector:
    matchLabels:
      app: myapp
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
```

🔐 myapp Pods can only send traffic to IPs in 10.0.0.0/24.

🚨 **Important Notes**

- **By default, all traffic is allowed**. Once you apply a Network Policy to a Pod, **only allowed traffic is permitted**.
- **You must label your Pods properly** for policies to work.
- **Test carefully**! It's easy to accidentally block traffic you need.

📘 **Summary Table**

Term	Meaning
podSelector	Select which Pods the policy applies to
ingress	Who can talk **to** the Pod
egress	Where the Pod can talk **to**
ipBlock	Allow/deny external IP ranges
namespaceSelector	Limit based on namespace

23) Kustomize

- **Kustomize** is a tool built into kubectl that lets you customize Kubernetes YAMLs **without modifying the original files**.

  🧠 In simple words:
  It helps you reuse, override, and patch your YAML files in a clean, structured way — like templates, but **without using Helm or templating engines**.

  It's great for managing different environments like **dev, staging, and prod** using the same base YAML files.

🔹 **Kustomize in Kubernetes**

Kustomize is a Kubernetes-native configuration management tool that allows you to **customize raw, template-free YAML files** for different environments (like dev, staging, and prod) without duplicating them.

✅ **Key Features**

- No need for templating languages (like Helm).
- Layered configuration using **overlays**.
- Supports **patching**, **variable substitution**, and **resource composition**.
- Built into kubectl (kubectl apply -k).

🧱 **Core Concepts**

Concept	Description
Base	A set of common YAML files (e.g., Deployment, Service).
Overlay	Environment-specific customizations (e.g., dev, prod).
kustomization.yaml	A file that defines how to customize the resources.

📄 **Example Directory Structure**

```
my-app/
├── base/
│   ├── deployment.yaml
│   ├── service.yaml
│   └── kustomization.yaml
└── overlays/
    ├── dev/
    │   ├── kustomization.yaml
    │   └── patch.yaml
    └── prod/
        ├── kustomization.yaml
        └── patch.yaml
```

📄 **Example kustomization.yaml (Base)**

```
resources:
  - deployment.yaml
  - service.yaml
```

📄 **Example kustomization.yaml (Overlay)**

```
resources:
  - ../../base
patchesStrategicMerge:
  - patch.yaml
```

📄 **Example Patch**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
```

This patch overrides the number of replicas in the base deployment.

🚀 **Apply with Kustomize**

```
kubectl apply -k overlays/dev
```

Would you like a **visual diagram** showing how Kustomize overlays and bases work together?

⭐ **2. Key Features of Kustomize**

Feature	Description
🧱 Layered config	Define a base and customize it for different environments
🔄 YAML patching	Change parts of YAMLs (e.g., image tag, replicas)
♻️ Reuse friendly	Keeps your base files clean and reusable
🔀 Overlays	Let you override configurations for dev/prod/staging, etc.
🔍 Built-in	Comes with kubectl (from v1.14+) — no extra install needed
💉 No templates	Pure YAML, no templating syntax like Helm

📁 **3. Folder Structure and YAML Examples**

Let's say you want different configs for dev and prod using a single base.

📂 **File Structure:**

```
csharp
CopyEdit
my-app/
├── base/
```

```
|   ├── deployment.yaml
|   ├── service.yaml
|   └── kustomization.yaml
├── overlays/
   ├── dev/
   |   ├── kustomization.yaml
   |   └── patch.yaml
   └── prod/
       ├── kustomization.yaml
       └── patch.yaml
```

📦 **Base Deployment (base/deployment.yaml)**

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: my-app:latest
```

📄 **Base Kustomization (base/kustomization.yaml)**

```yaml
resources:
  - deployment.yaml
  - service.yaml
```

🧪 **Dev Overlay (overlays/dev/kustomization.yaml)**

```yaml
resources:
  - ../../base
patchesStrategicMerge:
  - patch.yaml
```

🧪 **Dev Patch (overlays/dev/patch.yaml)**

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
```

```yaml
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: my-container
          image: my-app:dev
```

🏗️ Apply Dev Config

kubectl apply -k overlays/dev

🧠 4. Real-Time Use Cases

Use Case	How Kustomize Helps
🔁 Dev vs Prod configs	Easily switch between different configurations
📦 Image updates	Override container image version without editing base YAML
🌍 Multi-region deploy	Use same base with regional changes
💉 Testing patches	Apply only what's needed for a specific environment
🔧 Clean repo	Keeps base clean and reusable for CI/CD pipelines

🧩 5. Extra Functionalities

Feature	Description
images:	Override container images
namePrefix:	Add prefixes to resource names (like dev-)
namespace:	Set all resources to a specific namespace
commonLabels:	Add labels to all resources
configMapGenerator / secretGenerator	Auto-create ConfigMaps and Secrets from files or literals

Example:

```yaml
namespace: dev
namePrefix: dev-
commonLabels:
  env: dev
```

📌 6. Does Kustomize Have Types?

Not in the way that CRDs or Services have types — but it supports:

- **Base**: The reusable core config
- **Overlay**: Environment-specific modifications
- **Patches**: Partial changes to resources (strategic or JSON6902)
- **Generators**: Auto-generate ConfigMaps and Secrets

📘 **Summary Table**

Concept	Description
Base	Common core YAMLs shared across environments
Overlay	Environment-specific configs
Patch	Modify only the parts you want
Built into kubectl	No extra tool needed
Pure YAML	No scripting or templating syntax

🧪 **Bonus: Common Commands**

```
kubectl kustomize .              # See the output YAML
kubectl apply -k overlays/dev    # Apply dev config
kubectl apply -k overlays/prod   # Apply prod config
```

24) Logs

🔎 **Fluentd**

- **Fluentd** is an **open-source log processor**. In Kubernetes, it's commonly used to **collect, filter, transform, and forward logs** from all Pods and nodes to a log storage system like **Elasticsearch, Splunk, or Cloud Logging**.

  🧠 Think of Fluentd as a **log router**. It reads logs from the system, changes them if needed, and sends them where you want.

⭐ **2. Key Features**

Feature	Description
📦 Log Aggregation	Collects logs from many sources (containers, nodes, apps)
🔄 Log Transformation	Can filter, format, and enrich log data
🔌 Plugin Architecture	1000+ plugins to send logs to various tools (Elasticsearch, S3, etc.)
🔐 Secure and Reliable	Supports buffering, retries, TLS, etc.

🧰 Lightweight	Runs as a **DaemonSet** in Kubernetes (one per node)
💡 Extensible	Easy to customize with config files and plugins

📄 **How Fluentd Works in Kubernetes**

- Logs are written to /var/log/containers on each node.
- **Fluentd (as a DaemonSet)** reads those logs.
- It then filters and formats the logs.
- Sends logs to **a central location** (like Elasticsearch or a cloud logging service).

📁 **Example YAML (Fluentd as DaemonSet)**

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
spec:
  selector:
    matchLabels:
      name: fluentd
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      containers:
      - name: fluentd
        image: fluent/fluentd:v1.14-1
        env:
        - name: FLUENTD_ARGS
          value: "--no-supervisor -q"
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

📝 This sets up Fluentd on **every node**, pulling logs from Docker containers and system logs.

📌 5. Fluentd Config Example (fluent.conf)

conf

```
<source>
  @type tail
  path /var/log/containers/*.log
  format json
  tag kube.*
</source>

<match kube.**>
  @type elasticsearch
  host elasticsearch.logging.svc.cluster.local
  port 9200
  logstash_format true
</match>
```

🔄 This config reads logs from containers and forwards them to **Elasticsearch**.

🌍 6. Real-Time Use Cases

Use Case	Why Fluentd Helps
📦 Centralized Logging	Aggregate logs from all Pods and nodes in one place
📊 ELK Stack Integration	Fluentd works as the "log shipper" to Elasticsearch
🛠️ Troubleshooting	Makes it easy to search and analyze Pod logs
🔐 Compliance	Store logs in secure external systems
🔀 Multi-output Logging	Send logs to multiple destinations (e.g., S3 + Elasticsearch)
📁 Log Enrichment	Add Kubernetes metadata (Pod name, namespace, labels) to logs

🔗 Types / Plugins

Fluentd doesn't have "types" like Services do, but it has **input, filter, and output plugins**.

Plugin Type	Example	Purpose
Input	tail, http, syslog	Collect logs
Filter	record_transformer, grep	Modify logs
Output	elasticsearch, s3, gcs	Send logs somewhere

🔚 Summary

Item	Info
Tool	Fluentd
Purpose	Log aggregation, transformation, and forwarding
Runs As	DaemonSet
Used With	Elasticsearch, Splunk, Cloud Logging
Config File	fluent.conf
Key Use Case	Centralized and enriched logging for Kubernetes clusters

Scaling in Kubernetes

Kubernetes supports two types of scaling:
1. Horizontal Pod Autoscaling (HPA): Dynamically adjusts the number of pod replicas for a deployment or replica set. It monitors metrics like CPU, memory, or custom application metrics.Adds or removes pod replicas based on thresholds.
2. Node Scaling: Adding or removing nodes to/from the cluster.
Managed manually or automatically using tools like Cluster Autoscaler.
Cluster Autoscaler integrates with cloud providers to add remove virtual machines dynamically.

Managing Workloads in Kubernetes
Workloads: Workloads are the applications or services running on Kubernetes.
It is defined in Kubernetes using manifests (YAML or JSON files).

Types of Workloads:
Deployments: For stateless applications; supports scaling and updates.
StatefulSets: For stateful applications that require unique identities and stable storage (e.g., databases).
DaemonSets: Ensures a copy of a pod runs on every node (e.g., log collectors).
Jobs and CronJobs: For running one-time or scheduled tasks.

Other Features:
Load Balancing: Kubernetes ensures workloads are balanced across the cluster using Services and Ingress.
Monitoring and Logging: Tools like Prometheus, Grafana, and ELK Stack (Elasticsearch, Logstash, Kibana) help monitor workloads and log activities.
High Availability and Resilience: Kubernetes automatically restarts failed pods and reschedules them to healthy nodes.

SECURE KUBERNETES

1. Secure API server
2. RBAC
3. Network policies
4. Encrypted at rest (ETCD)
5. Secure Container Images
6. Cluster monitoring
7. upgrades.

Networking:

- Networking plugins and configuration options also facilitate pod-to-pod communication and network isolation

What is AWS RDS?

AWS RDS (Relational Database Service) is a fully managed relational database service provided by Amazon Web Services (AWS). It helps you set up, operate, and scale relational databases in the cloud. RDS supports several popular database engines like MySQL, PostgreSQL, MariaDB, Oracle, and Microsoft SQL Server. The key features of AWS RDS include:

- Automated backups
- Database patching
- Scaling of storage and compute resources
- High availability with Multi-AZ deployments
- Automatic failover
- Easy replication setup

How to Use AWS RDS with Kubernetes?

Using AWS RDS with Kubernetes involves accessing the RDS database from applications running in Kubernetes pods. RDS is a managed service that runs outside the Kubernetes cluster, so the application in Kubernetes will need to connect to it over the network.

Steps to Integrate AWS RDS with Kubernetes:

1. **Set Up AWS RDS Instance**:
    - First, create an RDS instance through the AWS Management Console or via the AWS CLI.
    - Select the database engine of your choice (e.g., MySQL, PostgreSQL).
    - Choose the instance size, storage, VPC, and other configuration details.
    - Ensure that the RDS instance is accessible from the Kubernetes cluster by configuring the **VPC** and **Security Groups** properly.
2. **Create a Secret for Database Credentials**: Kubernetes uses secrets to securely store sensitive information such as database credentials.

Example of creating a secret:
bash

```
kubectl create secret generic db-credentials --from-literal=DB_USER=myuser --from-literal=DB_PASSWORD=mypassword --from-literal=DB_HOST=mydb-instance.c1h1h3d22dkk.us-east-1.rds.amazonaws.com --from-literal=DB_NAME=mydatabase
```

3. **Define Kubernetes Deployment to Connect to RDS**: You can define a Kubernetes Deployment that contains an application (e.g., a web application or a backend service) that connects to the AWS RDS database. The application will need to retrieve the database credentials (from the secret) and connect to the RDS instance.

Here's an example of a deployment YAML that connects to AWS RDS:

```yaml
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:
  name: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp-container
        image: myapp-image:latest
        env:
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: DB_USER
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: DB_PASSWORD
        - name: DB_HOST
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: DB_HOST
        - name: DB_NAME
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: DB_NAME
        ports:
        - containerPort: 80
```

In this YAML:

- We define environment variables that reference the secrets we created earlier. The app will use these variables to connect to the RDS instance.
- The deployment creates multiple replicas of the app, which can scale up or down as needed.

4. **Service to Expose the Application**: If your application needs to be accessed externally (e.g., via a web browser), you should expose it using a Kubernetes Service. Here's an example of how to expose the application:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

In this case, we're exposing port 80 of the application to the outside world using a LoadBalancer service, which will create an external IP.

Example of Full Workflow:

1. **Create AWS RDS**: You would create an RDS instance from the AWS console or using AWS CLI. For example, to create a MySQL RDS instance, you could use:

bash

```
aws rds create-db-instance --db-instance-identifier mydb-instance --db-instance-class db.t3.micro --engine mysql --allocated-storage 20 --master-username myuser --master-user-password mypassword --backup-retention-period 7
```

2. **Access Database from Kubernetes**:
    - Ensure your Kubernetes cluster is inside the same VPC or has access to the VPC where the RDS instance is deployed.
    - You should have proper security groups and networking configuration to allow communication between the Kubernetes pods and RDS.
3. **Pod Deployment with DB Connection**: After deploying your application (as shown above), the pod containers will use the environment variables to access the database and perform queries.

Example MySQL Connection Code:
In your application (for example, a Node.js app), you would use the environment variables to configure the database connection:

javascript

```
const mysql = require('mysql');

const connection = mysql.createConnection({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
```

```
  database: process.env.DB_NAME
});

connection.connect(function(err) {
  if (err) {
    console.error('error connecting: ' + err.stack);
    return;
  }
  console.log('connected as id ' + connection.threadId);
});
```

In this example, your Node.js app retrieves the database credentials from the environment variables and establishes a connection to the AWS RDS instance.

Key Considerations:

- **Security**: Ensure that your Kubernetes cluster has the necessary IAM roles and security groups configured to access the RDS instance securely.
- **VPC and Networking**: The RDS instance must be in a VPC that is accessible from the Kubernetes cluster. This can involve setting up peering between VPCs or ensuring proper networking configuration within the same VPC.
- **Scaling**: AWS RDS is a managed service, so scaling your database (upgrading instance size, adding replicas, etc.) is different from scaling your application. You'll need to manage that separately using the AWS Console or CLI.

What is AWS RDS?

AWS RDS (Relational Database Service) is a fully managed relational database service provided by Amazon Web Services (AWS). It simplifies database management tasks such as provisioning, patching, backup, recovery, and scaling. AWS RDS supports several relational database engines, including:

- **MySQL**
- **PostgreSQL**
- **MariaDB**
- **Oracle**
- **Microsoft SQL Server**

RDS automates tasks like backups, software patching, and replication, and it provides high availability, scalability, and security. This makes it ideal for production-grade applications that require reliable database management without the overhead of managing the database infrastructure.

How to Use AWS RDS in Kubernetes?

When using AWS RDS in Kubernetes, your Kubernetes application (running in pods) will communicate with the RDS instance over the network. The RDS instance itself is external to the Kubernetes cluster, so the Kubernetes pods will access it just like they would access any other external service.

High-Level Steps to Use AWS RDS in Kubernetes (Production-Level Approach)

1. **Create an AWS RDS Instance**.
2. **Configure Secrets in Kubernetes for Database Credentials**.
3. **Create Kubernetes Deployment that Uses AWS RDS**.
4. **Secure Communication (VPC, Security Groups, etc.)**.
5. **Monitor and Scale the Database**.

Let's walk through these steps, focusing on a production-level configuration.

Step 1: Create an AWS RDS Instance

1. **Log in to the AWS Management Console** and navigate to the RDS service.
2. **Create an RDS Instance**:
    - Choose the database engine (e.g., MySQL, PostgreSQL).
    - Set the **instance type** (e.g., db.m5.large for production workloads).
    - Set the **storage type** and size.
    - Set the **backup retention period** and enable **automated backups** for disaster recovery.
    - Enable **Multi-AZ** for high availability (if required).
    - Set up **IAM roles**, **VPC**, **subnets**, and **security groups** to allow access from your Kubernetes cluster.

For example, creating an RDS PostgreSQL instance using AWS CLI:

```
aws rds create-db-instance \
  --db-instance-identifier myprod-db \
  --db-instance-class db.m5.large \
  --engine postgres \
  --allocated-storage 100 \
  --master-username adminuser \
  --master-user-password adminpassword \
  --backup-retention-period 7 \
  --multi-az \
  --vpc-security-group-ids sg-0123456789abcdef0 \
  --availability-zone us-east-1a
```

Step 2: Configure Secrets in Kubernetes for Database Credentials

Storing credentials securely in Kubernetes is crucial for production environments.
Kubernetes **Secrets** are ideal for this purpose.
Example: Create a secret with the RDS credentials.

```
kubectl create secret generic mydb-credentials \
  --from-literal=DB_USER=adminuser \
  --from-literal=DB_PASSWORD=adminpassword \
  --from-literal=DB_HOST=myprod-db.c1h1h3d22dkk.us-east-1.rds.amazonaws.com \
  --from-literal=DB_PORT=5432 \
  --from-literal=DB_NAME=mydatabase
```

This secret will be used by your application to access the database securely.

Step 3: Create a Kubernetes Deployment that Uses AWS RDS

Next, create a **Deployment** that will run your application (e.g., a web app, backend service) in Kubernetes. The application will access the RDS database using the environment variables set from the **Secrets**.

Here's an example Kubernetes deployment for a web application that connects to an AWS RDS PostgreSQL database:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-deployment
spec:
  replicas: 3  # Production-grade app often has multiple replicas for high
availability
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: myapp-image:latest
        env:
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: mydb-credentials
              key: DB_USER
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mydb-credentials
              key: DB_PASSWORD
        - name: DB_HOST
          valueFrom:
            secretKeyRef:
              name: mydb-credentials
              key: DB_HOST
        - name: DB_PORT
          valueFrom:
            secretKeyRef:
              name: mydb-credentials
              key: DB_PORT
        - name: DB_NAME
          valueFrom:
```

```
            secretKeyRef:
              name: mydb-credentials
              key: DB_NAME
        ports:
        - containerPort: 80
```

In this YAML:

- The **Deployment** specifies 3 replicas (production-grade apps are usually scaled horizontally for high availability).
- The **env** section pulls sensitive data (DB credentials) from the Kubernetes **Secrets**.
- Replace myapp-image:latest with the actual image of your app.

Step 4: Secure Communication (VPC, Security Groups)

For **production environments**, ensure that your Kubernetes cluster and the RDS instance are on the same **VPC** (Virtual Private Cloud) for secure and low-latency communication. Also, set up **Security Groups** properly:

- **Security Group for RDS**: The RDS instance should only allow inbound connections on port 5432 (or the port corresponding to your database engine) from the IPs or security groups associated with the Kubernetes cluster.
- **Security Group for Kubernetes Pods**: The Kubernetes nodes (or worker nodes) must be allowed to access RDS.

Example: Add security group rules:

```
aws ec2 authorize-security-group-ingress \
  --region us-east-1 \
  --group-id sg-0123456789abcdef0 \
  --protocol tcp --port 5432 \
  --source-group sg-0987654321abcdef0  # Kubernetes worker node security group
```

Ensure the proper **IAM roles** and **permissions** are in place for your Kubernetes nodes to interact with AWS RDS if needed.

Step 5: Monitoring and Scaling the Database

In a production environment, monitoring the RDS instance and scaling it according to traffic and usage is critical. AWS provides several monitoring tools:

- **Amazon CloudWatch**: Use CloudWatch to monitor RDS performance (CPU, disk usage, network traffic, etc.).
- **Automatic Scaling**: You can set up **Auto Scaling** for RDS read replicas if your application requires increased read throughput.
- **RDS Performance Insights**: Helps you analyze database performance bottlenecks.
- **RDS Enhanced Monitoring**: Provides real-time metrics about RDS instance health.

For scaling, you can adjust the instance size or add read replicas. You can also automate failover with **Multi-AZ deployments** to ensure high availability.

Example: Scaling the RDS Instance

```
aws rds modify-db-instance \
  --db-instance-identifier myprod-db \
  --db-instance-class db.m5.2xlarge \
  --apply-immediately
```

This command scales the RDS instance to a larger instance type for increased performance.

Step 6: Exposing the Application in Kubernetes (Service)
To expose the application outside the Kubernetes cluster, create a **Kubernetes Service** (e.g., a LoadBalancer service for public access):

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
spec:
  selector:
    app: webapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer  # Exposes the app to the internet
```

This service will assign an external IP to the application, making it accessible via HTTP.

Example Full Workflow for Production-Level Setup:

1. **Set up AWS RDS instance** (with Multi-AZ for HA and backup enabled).
2. **Create Kubernetes Secrets** to securely store database credentials.
3. **Create a Kubernetes Deployment** that connects to the RDS instance using the secrets.
4. **Secure communication** by placing RDS and Kubernetes in the same VPC, configuring proper security groups.
5. **Monitor and scale** the RDS instance using AWS CloudWatch, Performance Insights, and Auto Scaling.

Conclusion
By using **AWS RDS** with **Kubernetes**, you can offload the database management to a fully managed service, ensuring high availability, scaling, and performance while keeping your Kubernetes deployment simple. This setup is suitable for production environments where security, scalability, and reliability are critical.

How to set up SSO for application inside the kubernetes pods and with examples

To set up **Single Sign-On (SSO) for an application running inside Kubernetes pods, you need to integrate an **Identity Provider (IdP)** that supports **OAuth2** or **OIDC (OpenID Connect)**. In this example, I will show you how to integrate **OIDC** for SSO using **AWS Cognito** as the IdP, which is one of the popular choices for managing authentication and user access.**

The general approach will involve:
1. Configuring **AWS Cognito as the **IdP**.**
2. Setting up the **application to use **OIDC** for authentication.**
3. Deploying the application in a **Kubernetes cluster.**
4. Setting up appropriate configurations in the application to handle authentication through Cognito.

Steps to Set Up SSO for Application Inside Kubernetes Pods

Prerequisites
1. **AWS Cognito set up as the IdP for managing users.**
2. Kubernetes cluster running with **Ingress configured for external access to the app.**
3. A sample application running in Kubernetes (e.g., a web application like **Node.js, **Spring Boot**, **Express.js**).**
4. OAuth2 or OpenID Connect integration in your application.

Step 1: Configure AWS Cognito for Authentication

1. **Create a Cognito User Pool:**
  - Go to the **Cognito Console in AWS and create a **User Pool** for your application.**
  - Configure attributes like **email, **username**, **password policies**, etc.**

2. **Create an App Client:**
  - Inside the **User Pool, go to **App clients** and create a new **App Client**.**
  - Ensure that **OAuth 2.0 flows like **Authorization Code Grant** are enabled.**
  - Note down the **App Client ID and **App Client Secret** (for use in your application).**

3. **Configure Cognito App Client Settings:**
  - Set the **Callback URL (where Cognito will redirect the user after login, e.g., `http://your-app/callback`).**
  - Set the **Sign-out URL (where users will be redirected after signing out).**
  - Enable **OAuth 2.0 scopes: `openid`, `email`, `profile`.**

4. **Configure Domain:**
  - Set up a **Domain for your Cognito instance so it can provide the login page (e.g., `https://<your-cognito-domain>.auth.us-west-2.amazoncognito.com`).**

5. **Enable the OIDC Settings:**
  - You'll need the **Issuer URL for the OIDC configuration, which will look something like:**

```
https://cognito-idp.us-west-2.amazonaws.com/us-west-2_XXXXXXXXX
```
- This URL will be used to integrate the authentication mechanism into your app.

Step 2: Integrating OIDC into Your Application

Let's assume you are using a **Node.js** application, but the general process applies to other frameworks (e.g., **Spring Boot**, **Express**, **Flask**).

1. **Install OIDC Authentication Library**:
   Use libraries like **passport.js** for Node.js to handle OAuth2 or OIDC authentication.

   For Node.js with **passport.js**:
   ```bash
   npm install passport passport-oidc express-session
   ```

2. **Configure OIDC Authentication**:
   In your **Node.js** application, configure **passport-oidc** to authenticate users using AWS Cognito.

   Example of `server.js` with OIDC authentication:

   ```javascript
   const express = require('express');
   const passport = require('passport');
   const session = require('express-session');
   const OIDCStrategy = require('passport-oidc').Strategy;

   const app = express();

   passport.use(
     new OIDCStrategy(
       {
         issuer: 'https://cognito-idp.us-west-2.amazonaws.com/us-west-2_XXXXXXXXX',
         authorizationURL: 'https://<your-cognito-domain>.auth.us-west-2.amazoncognito.com/oauth2/authorize',
         tokenURL: 'https://<your-cognito-domain>.auth.us-west-2.amazoncognito.com/oauth2/token',
         clientID: '<your-app-client-id>',
         clientSecret: '<your-app-client-secret>',
         callbackURL: 'http://localhost:3000/callback',
       },
       function (issuer, profile, done) {
         return done(null, profile);
       }
```

```
  )
);

app.use(session({ secret: 'secret', resave: true, saveUninitialized: true }));
app.use(passport.initialize());
app.use(passport.session());

app.get('/login', (req, res) => {
  passport.authenticate('oidc')(req, res);
});

app.get('/callback', (req, res) => {
  passport.authenticate('oidc', { failureRedirect: '/' })(req, res, function () {
    res.redirect('/');
  });
});

app.get('/', (req, res) => {
  if (req.isAuthenticated()) {
    res.send(`Hello, ${req.user.displayName}!`);
  } else {
    res.send('Hello, guest! Please <a href="/login">login</a>');
  }
});

app.listen(3000, () => {
  console.log('App is listening on port 3000');
});
```

3. **Start the Application Locally** (for testing):
   Run the app locally first to verify that the authentication flow works before deploying to Kubernetes.

   ```bash
   node server.js
   ```

   Visit `http://localhost:3000` and test the SSO login flow with AWS Cognito.

Step 3: Deploy the Application in Kubernetes

1. **Containerize the Application**:
   Create a `Dockerfile` for your application.

   Example `Dockerfile` for a Node.js app:

```dockerfile
FROM node:16

WORKDIR /app

COPY package.json package-lock.json ./
RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "server.js"]
```

Build and push the image to a container registry (e.g., DockerHub, AWS ECR).

```bash
docker build -t my-app .
docker tag my-app:latest <your-dockerhub-username>/my-app:latest
docker push <your-dockerhub-username>/my-app:latest
```

2. **Create Kubernetes Deployment YAML**:
Define your Kubernetes deployment (`deployment.yaml`).

Example:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: <your-dockerhub-username>/my-app:latest
          ports:
```

```
        - containerPort: 3000
        env:
        - name: CLIENT_ID
          valueFrom:
            secretKeyRef:
              name: cognito-credentials
              key: client-id
        - name: CLIENT_SECRET
          valueFrom:
            secretKeyRef:
              name: cognito-credentials
              key: client-secret
```

3. **Create Kubernetes Service and Ingress (optional)**:
   Expose your app using a **Service** and optionally an **Ingress**.

   Example `service.yaml`:

   ```yaml
   apiVersion: v1
   kind: Service
   metadata:
     name: my-app-service
   spec:
     selector:
       app: my-app
     ports:
       - protocol: TCP
         port: 80
         targetPort: 3000
   ```

   Example `ingress.yaml` for exposing the service via Ingress:

   ```yaml
   apiVersion: networking.k8s.io/v1
   kind: Ingress
   metadata:
     name: my-app-ingress
   spec:
     rules:
     - host: my-app.example.com
       http:
         paths:
         - path: /
           pathType: Prefix
```

```
        backend:
          service:
            name: my-app-service
            port:
              number: 80
```

4. **Deploy to Kubernetes**:
   Apply the Kubernetes resources (`deployment.yaml`, `service.yaml`, `ingress.yaml`) to deploy the app:

   ```bash
   kubectl apply -f deployment.yaml
   kubectl apply -f service.yaml
   kubectl apply -f ingress.yaml
   ```

Step 4: Verify the SSO Flow in Kubernetes

1. **Access the application** via the Ingress URL or external IP (e.g., `http://my-app.example.com`).
2. **Test the login flow** by accessing the login page, which should redirect to AWS Cognito for authentication.
3. After successful login, Cognito will redirect back to the callback URL (`/callback`), and your application should be authenticated.

Step 5: Manage Secrets for Sensitive Information

For sensitive information like the **Client ID** and **Client Secret**, store them in **Kubernetes Secrets** instead of hardcoding them in the app code. Here's how to create a Kubernetes Secret:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: cognito-credentials
type: Opaque
data:
  client-id: <base64-encoded-client-id>
  client-secret: <base64-encoded-client-secret>
```

Apply the Secret:

```bash
kubectl apply -f secret.yaml
```

```

### Conclusion

**Setting up **SSO** for an application inside Kubernetes involves configuring an **Identity Provider** (e.g., AWS Cognito) and integrating your application to authenticate via **OIDC** or **OAuth2**. Once integrated, deploy the app inside Kubernetes and configure **Ingress** or **Service** for external access. Ensure **Kubernetes Secrets** are used for managing sensitive credentials securely. This will enable **SSO** authentication for your users in the application running in Kubernetes.**

**AWS EC2**

**What is AWS EC2?**

● EC2 stands for Elastic Compute Cloud.
● Amazon EC2 is the virtual machine in the Cloud Environment.
● Amazon EC2 provides scalable cap

**EBS Volume:**

● EBS Stands for Elastic Block Storage.
● It is the block-level storage that is assigned to your single EC2 Instance.
● It persists independently from running EC2.
o Types of EBS Storage
▪ General Purpose (SSD)
▪ Provisioned IOPS (SSD)
▪ Throughput Optimized Hard Disk Drive
▪ Cold Hard Disk Drive
▪ Magnetic

**Instance Store:**

Instance store is the ephemeral block-level storage for the EC2 instance.
● Instance stores can be used for faster processing and temporary storage of the application.

**AMI:**

AMI Stands for Amazon Machine Image.
● AMI decides the OS, installs dependencies, libraries, data of your EC2 instances.
● Multiple instances with the same configuration can be launched using a single AMI.

**Security Group:**
```

A Security group acts as a virtual firewall for your EC2 Instances.
● It decides the type of port and kind of traffic to allow**.**
● **Security groups** are active at **the instance level** whereas **Network ACLs** are active at the **subnet level.**
● Security Groups can only allow but can't deny the rules.
● The Security group is considered **stateful**.
● By default, in the outbound rule all traffic is allowed and needs to define the inbound rules.

Key Pair:
A key pair, consisting of a private key and a public key, is a set of security credentials that you can use to prove your identity while connecting to an instance.
● Amazon EC2 instances use two keys, one is the public key which is attached to your EC2 instance.
● Another is the private key which is with you. You can get access to the EC2 instance only if these keys get matched.
● Keep the private key in a secure place.

AWS IAM
What is Identity Access and Management?
● IAM stands for Identity and Access Management.
● AWS IAM may be a service that helps you control access to AWS resources securely.
● You use IAM to regulate who is allowed and have permissions to AWS Resources.
● You can manage and use resources in your AWS account without having to share your password or access key.
● It enables you to manage access to AWS services and resources securely.
● We can attach Policies to AWS users, groups, and roles.
Principal: An Entity that will make a call for action or operation on an AWS Resource. Users, Groups, Roles all are AWS Principal. AWS Account Root user is the first principal.

IAM User & Root User
● Root User - When you first create an AWS account, you begin with an email (Username) and Password with complete access to all AWS services and resources in the account. This is the AWS account, root user.
● IAM User - A user that you create in AWS.
o It represents the person or service who interacts with AWS.
o IAM users' primary purpose is to give people the ability to sign in to AWS individually without sharing the password with others.
o Access permissions will be depending on the policies which are assigned to the IAM User.
IAM Group
● A group is a collection of IAM users.
● You can assign specific permission to a group and add the users to that

group.
● For example, you could have a group called DB Admins and give that type of permission that Database administrators typically need.

IAM Role

● IAM Role is like a user with policies attached to it that decides what an identity can or cannot do.
● It will not have any credentials/Password attached to it.
● A Role can be assigned to a federated user who signed in from an external Identity Provider.
● IAM users can temporarily assume a role and get different permission for the task.

IAM Policies

● It decides what level of access an Identity or AWS Resource will possess.
● A Policy is an object associated with identity and defines their level of access to a certain resource.
● These policies are evaluated when an IAM principal (user or role) makes a Request
● Policies are JSON based documents.
● Permissions inside policies decide if the request is allowed or denied.
o Resource-Based Policies: These JSON based policy documents attached to a resource such as Amazon S3 Bucket.
o These policies grant permission to perform an action on that resource and define under what condition it will apply.
o These policies are the inline policies, not managed resource-based policies.
o IAM supports only one type of resource-based policy called trust policy, and this policy is attached to a role.
o Identity-Based Policies: These policies have complete control over the identity that it can perform on which resource and under which condition.
o Managed policies: Managed policies can attach to the multiple users, groups, and roles in the AWS Account.
▪ AWS managed policies: These policies are created and managed by AWS.
▪ Customer managed policies: These policies are created and managed by you. It provides more precise control than AWS Managed policies.
o Inline policies: Inline policies are the policies that can directly be attached to any individual user, group, or role. It maintains a one-to-one relationship between the policy and the identity.

IAM Security Best Practices:

● Grant least possible access rights.
● Enable multi-factor authentication (MFA).
● Monitor activity in your AWS account using CloudTrail.
● Use policy conditions for extra security.
● Create a strong password policy for your users.
● Remove unnecessary credentials.

AWS Beanstalk

What is Amazon Elastic Beanstalk?

● Beanstalk is a compute service for deploying and scaling applications developed in many popular languages.

AWS Elastic Beanstalk supports two types of Environment:

● Web Tier Environment

● Worker Environment

What is AWS Lambda?

AWS Lambda is a serverless compute service that lets you run code in response to events.

● AWS Lambda is a serverless compute service through which you can run your code without provisioning any Servers.

● It only runs your code when needed and also scales automatically when the request count increases.

● AWS Lambda follows the Pay per use principle – it means there is no charge when your code is not running.

● Lambda allows you to run your code for any application or backend service with zero administration.

● Lambda can run code in response to the events. Example – update in DynamoDB Table or change in S3 bucket.

● You can even run your code in response to HTTP requests using Amazon API Gateway.

What is Serverless computing?

● Serverless computing is a method of providing backend services on a pay per use basis.

● Serverless/Cloud vendor allows you to write and deploy code without worrying about the underlying infrastructure.

● Servers are still there, but you are not managing them, and the vendor will charge you based on usage.

When do you use Lambda?

● When using AWS Lambda, you are only responsible for your code.

● AWS Lambda manages the memory, CPU, Network, and other resources.

● It means you cannot log in to the compute instances or customize the operating system.

● If you want to manage your own compute resources, you can use other compute services such as EC2, Elastic Beanstalk.

● There will be a level of abstraction which means you cannot log in to the server or customize the runtime.

How does Lambda work?

Lambda Functions

- A function is a block of code in Lambda.
- You upload your application/code in the form of single or multiple functions.
- You can upload a zip file, or you can upload a file from the S3 bucket as well.
- After deploying the Lambda function, Lambda automatically monitors functions on your behalf, reporting metrics through Amazon CloudWatch

AWS Fargate

What is AWS Fargate?

AWS Fargate is a serverless compute service that is used for containers by Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS).

- It eliminates the tasks required to provision, configure, or scale groups of virtual machines like Amazon EC2 to run containers.
- It packages the application in containers, by just specifying the CPU and memory requirements with IAM policies. Fargate task does not share its underlying kernel, memory resources, CPU resources, or elastic network interface (ENI) with another task.
- It does not support all the task definition parameters that are available in Amazon ECS tasks. Only a few are valid for Fargate tasks with some limitations.
- Kubernetes can be integrated with AWS Fargate by using controllers. These controllers are responsible for scheduling native Kubernetes pods onto Fargate.
- Security groups for pods in EKS can not be used when pods running on Fargate.
- The following storage types are supported for Fargate tasks:
  - Amazon EFS volumes for persistent storage
  - Ephemeral storage for nonpersistent storage

Amazon Elastic Kubernetes Service(EKS)

What is Amazon Elastic Kubernetes Service(EKS)?

Amazon Elastic Kubernetes Service (Amazon EKS) is a service that enables users to manage Kubernetes applications in the AWS cloud or on-premises.

Any standard Kubernetes application can be migrated to EKS without altering the

code.
The EKS cluster consists of two components:
● Amazon EKS control plane
● Amazon EKS nodes

Amazon Elastic Container Service
What is Amazon ECS?
Amazon Elastic Container Service (Amazon ECS) is a regional container orchestration service like Docker that allows to execute, stop, and manage containers on a cluster.
A container is a standard unit of software development that combines code, its dependencies, and system libraries so that the application runs smoothly from one environment to another.

Amazon Elastic Container Registry
What is Amazon Elastic Container Registry?
Amazon Elastic Container Registry (ECR) is a managed service that allows users to store, manage, share, and deploy container images and artifacts. It is mainly integrated with Amazon Elastic Container Service (ECS), for simplifying the production workflow.

Features:
● It stores both the containers which are created, and any container software bought through AWS Marketplace.
● It is integrated with Amazon Elastic Container Service (ECS), Amazon Elastic Kubernetes Service (EKS), and AWS Lambda, and AWS Fargate for easy deployments.
● AWS Identity and Access Management (IAM) enables resource-level control of each repository within ECR.
● It supports public and private container image repositories. It allows sharing container applications privately within the organization or publicly for anyone to download.
● A separate portal called Amazon ECR Public Gallery, helps to access all public repositories hosted on Amazon ECR Public.
● It stores the container images in Amazon S3 because S3 provides 99.999999999% (11 9's) of data durability.
● It allows cross-region and cross-account replication of the data for high availability applications.
● Encryption can be done via HTTPS while transferring container images. Images are also encrypted at rest using Amazon S3 server-side encryption or by using customer keys managed by AWS KMS.
● It is integrated with continuous integration and continuous delivery and also with third-party developer tools.
● Lifecycle policies are used to manage the lifecycle of the images

Amazon S3

What is Amazon S3?

S3 stands for Simple Storage Service.
Amazon S3 is object storage that allows us to store any kind of data in the bucket.
It provides availability in multiple AZs, durability, security, and performance at a very low cost.
Any type of customer can use it to store and protect any amount of data for use cases, like static and dynamic websites, data analytics, and backup.

Basics of S3?

- It is object-based storage.
- Files are stored in Buckets.
- The bucket is a kind of folder.
- Folders can be from 0 to 5 TB.
- S3 bucket names must be unique globally.
- When you upload a file in S3, you will receive an HTTP 200 code if the upload was successful.
- S3 offers Strong consistency for PUTs of new objects, overwrites or delete of current object and List operations.
- By Default, all the Objects in the bucket are private.

Permissions & Management.

- **Access Control List**: ACLs used to grant read/write permission to another AWS Account.
- **Bucket Policy:** It uses JSON based access policy advance permission to your S3 Resources.
- **CORS**: CORS stands for Cross-Origin Resource Sharing. It allows cross-origin access to your S3 Resources.

AWS EBS - Elastic Block Store

What is AWS EBS?

Amazon Elastic Block Store (AWS EBS) is a persistent block-level storage (volume) service designed to be used with Amazon EC2 instances. EBS is AZ specific & automatically replicated within its AZ to protect from component failure, offering high availability and durability.

Features:

- High Performance (Provides single-digit-millisecond latency for high-performance)
- Highly Scalable (Scale to petabytes)
- Offers high availability (guaranteed 99.999% by Amazon) & Durability
- Offers seamless encryption of data at rest through Amazon Key Management Service (KMS).
- Automate Backups through data lifecycle policies using EBS Snapshots to S3 Storage.
- EBS detached from an EC2 instance and attached to another one quickly.

EBS vs Instance Store

Instance Store (ephemeral storage) :

● It is ideal for temporary block-level storage like buffers, caches, temporary content
● Data on an instance store volume persists only during the life of the associated instance. (As it is volatile storage - lose data if stop the instance/instance crash)
● Physically attached to ec2 instance - hence, the lowest possible latency.
● Massive IOPS - High performance
● Instance store backed Instances can be of maximum 10GiB volume size
● Instance store volume cannot be attached to an instance, once the Instance is up and running.
● Instance store volume can be used as root volume.
● You cannot create a snapshot of an instance store volume.

EBS :

● Persistent Storage.
● Reliable & Durable Storage.
● EBS volume can be detached from one instance and attached to another instance.
● EBS boots faster than instance stores.

AWS EFS - Elastic File Storage

What is AWS EFS?

Amazon Elastic File System (Amazon EFS) provides a scalable, fully managed elastic distributed file system based on NFS. It is persistent file storage & can be easily scaled up to petabytes.
It is designed to share parallelly with thousands of EC2 instances to provide better throughput and IOPS. It is a regional service automatically replicated across multiple AZ's to provide High Availability and durability.

AWS Snowball

What is AWS Snowball?

● AWS Snowball is a storage device used to transfer a large amount of data ranging from 50TB - 80TB between Amazon Simple Storage Service and onsite data storage location at high speed.

AWS Storage Gateway

What is the AWS Storage Gateway?

AWS Storage Gateway is a hybrid cloud storage service that allows your on-premise storage & IT infrastructure to seamlessly integrate with AWS Cloud Storage Services. It Can be AWS Provided Hardware or Compatible Virtual Machine.

Amazon Aurora

What is Amazon Aurora?

Aurora is the fully managed RDS services offered by AWS. It's only compatible with PostgreSQL/MySQL. As per AWS, Aurora provides 5 times throughput to traditional MySQL and 3 times throughput to PostgreSQL.

Amazon DocumentDB
What is Amazon DocumentDB?
DocumentDB is a fully managed document database service by AWS which supports MongoDB workloads. It is highly recommended for storing, querying, and indexing JSON Data.

Amazon DynamoDB
What is DynamoDB?
● AWS DynamoDB is a Key-value and DocumentDB database by Amazon.
● It delivers a single Digit millisecond Latency.
● It can handle 20 million requests per second and 10 trillion requests a day.
● It is a Serverless Service; it means no servers to manage.
● It maintains the performance by managing the data traffic of tables over multiple servers.

Amazon RDS
What is Amazon RDS?
RDS (Relational Database System) in AWS makes it easy to operate, manage, and scale in the cloud.
It provides scalable capacity with a cost-efficient pricing option and automates manual administrative tasks such as patching, backup setup, and hardware provisioning.

MySQL
● It is the most popular open-source DB in the world.
● Amazon RDS makes it easy to provision the DB in AWS Environment without worrying about the physical infrastructure.
● In this way, you can focus on application development rather than Infra. Management.

AWS Secrets Manager
What is AWS Secrets Manager?
AWS Secrets Manager is a service that replaces secret credentials in the code like passwords, with an API call to retrieve the secret programmatically. The service provides a feature to rotate, manage, and retrieve database passwords, OAuth tokens, API keys, and other secret credentials. It ensures in-transit encryption of the secret between AWS and the system to retrieve the secret.
Secrets Manager can easily rotate credentials for AWS databases without any additional programming. Though rotating the secrets for other databases or services requires Lambda function to instruct how Secrets Manager interacts with the database or service.

Accessing Secrets Manager:

- AWS Management Console
  - It stores binary data in secret.
- AWS Command Line Tools
  - AWS Command Line Interface
  - AWS Tools for Windows PowerShell
- AWS SDKs
- Secrets Manager HTTPS Query API

Secret rotation is available for the following Databases:
- MySQL on Amazon RDS
- PostgreSQL on Amazon RDS
- Oracle on Amazon RDS
- MariaDB on Amazon RDS
- Amazon DocumentDB
- Amazon Redshift
- Microsoft SQL Server on Amazon RDS
- Amazon Aurora on Amazon RDS

Features:
- It provides security and compliance facilities by rotating secrets safely without the need for code deployment.
- With Secrets Manager, IAM policies and resource-based policies can assign specific permissions for developers to retrieve secrets and passwords used in the development environment or the production environment.
- Secrets can be secured with encryption keys managed by AWS Key Management Service (KMS).
- It integrates with AWS CloudTrail and AWS CloudWatch to log and monitor services for centralized auditing.

Use cases:
- Store sensitive information as part of the encrypted secret value, either in the SecretString or SecretBinary field.
- Use a Secrets Manager open-source client component to cache secrets and update them only when there is a need for rotation.
- When an API request quota exceeds, the Secrets Manager throttles the request and returns a 'ThrottlingException' error. To resolve this, retry the requests.
- It integrates with AWS Config and facilitates tracking of changes in Secrets Manager.

AWS Certificate Manager (ACM)

What is AWS Certificate Manager?

AWS Certificate Manager is a service that allows a user to provide, manage, renew and deploy public and private Secure Sockets Layer/Transport Layer Security (SSL/TLS) X.509 certificates.

The certificates can be integrated with AWS services either by issuing them directly with ACM or importing third-party certificates into the ACM management system.

SSL Server Certificates:
● HTTPS transactions require server certificates X.509 that bind the public key in the certificate to provide authenticity.
● The certificates are signed by a certificate authority (CA) and contain the server's name, the validity period, the public key, the signature algorithm, and more.

The different types of SSL certificates are:
● Extended Validation Certificates (EV SSL) - most expensive SSL certificate type
● Organization Validated Certificates (OV SSL) - validate a business' creditably
● Domain Validated Certificates (DV SSL) - provide minimal encryption
● Wildcard SSL Certificate - secures base domain and subdomains
● Multi-Domain SSL Certificate (MDC) - secure up to hundreds of domain and subdomains
● Unified Communications Certificate (UCC) - single certificate secures multiple domain names.

Ways to deploy managed X.509 certificates:
1. AWS Certificate Manager (ACM) - useful for large customers who need a secure web presence.
● ACM certificates are deployed using Amazon API Gateway, Elastic Load Balancing, Amazon CloudFront.
2. ACM Private CA - useful for large customers building a public key infrastructure (PKI) inside the AWS cloud and intended for private use within an organization.
● It helps create a certificate authority (CA) hierarchy and issue certificates to authenticate users, computers, applications, services, servers, and other devices.
● Private certificates by Private CA for applications provide variable certificate lifetimes or resource names.

ACM certificates are supported by the following services:
● Elastic Load Balancing
● Amazon CloudFront
● AWS Elastic Beanstalk
● Amazon API Gateway
● AWS Nitro Enclaves (an Amazon EC2 feature)
● AWS CloudFormation

AWS Auto Scaling
What is AWS Auto Scaling?
● AWS Auto Scaling keeps on monitoring your Application and automatically adjusts the capacity required for steady and predictable performance.
● By using auto scaling it's very easy to set up the scaling of the application automatically with no manual intervention.

● It allows you to create scaling plans for the resources like EC2 Instances, Amazon EC2 tasks, Amazon DynamoDB, Amazon Aurora Read Replicas.
● It balances Performance Optimization and cost.

Monitoring:
● **Health Check:** Keep on checking the health of the instance and remove the unhealthy instance out of Target Group.
● **CloudWatch Events:** AutoScaling can submit events to Cloudwatch for any type of action to perform in the autoscaling group such as a launch or terminate an instance.
● **CloudWatch Metrics:** It shows you the statistics of whether your application is performing as expected.
● **Notification Service:** Autoscaling can send a notification to your email if the autoscaling group launches or the instance gets terminated.

AWS CloudFormation
What is AWS CloudFormation?
AWS CloudFormation is a service that collects AWS and third-party resources and manages them throughout their life cycles, by launching them together as a stack.
A template is used to create, update, and delete an entire stack as a single unit, without managing resources individually.
It provides the capability to reuse the template to set the resources easily and repeatedly.
It can be integrated with AWS IAM for security. It can be integrated with CloudTail to capture API calls as events.
Templates - A JSON or YAML formatted text file used for building AWS resources.
Stack - It is a single unit of resources.
Change sets - It allows checking how any change to a resource might impact the running resources.
Stacks can be created using the AWS CloudFormation console and AWS Command Line Interface (CLI).
Stack updates: First the changes are submitted and compared with the current state of the stack and only the changed resources get updated.
There are two methods for updating stacks:
● **Direct update** - when there is a need to quickly deploy the updates.
● **Creating and executing change sets** - they are JSON files, providing a preview option for the changes to be applied.
StackSets are responsible for safely provisioning, updating, or deleting stacks.
Nested Stacks are stacks created within another stack by using the AWS::CloudFormation::Stack resource.
When there is a need for common resources in the template, Nested stacks can be used by declaring the same components instead of creating the components multiple times. The main stack is termed as parent stack and other belonging stacks are termed as child stack, which can be implemented by using ref variable '! Ref'.

AWS CloudFormation Registry helps to provision third-party application resources alongside AWS resources. Examples of third-party resources are incident management, version control tools.

Amazon CloudWatch

What is Amazon CloudWatch?

Amazon CloudWatch is a service that helps to monitor and manage services by providing data and actionable insights for AWS applications and infrastructure resources.
It monitors AWS resources such as Amazon RDS DB instances, Amazon EC2 instances, Amazon DynamoDB tables, and, as well as any log files generated by the applications.

Amazon CloudWatch can be accessed by the following methods:
- Amazon CloudWatch console
- AWS CLI
- CloudWatch API
- AWS SDKs

Amazon CloudWatch is used together with the following services:
- Amazon Simple Notification Service (Amazon SNS)
- Amazon EC2 Auto Scaling
- AWS CloudTrail
- AWS Identity and Access Management (IAM)

Amazon CloudFront

What is Amazon CloudFront?

Amazon CloudFront is a content delivery network (CDN) service that securely delivers any kind of data to customers worldwide with low latency, low network and high transfer speeds.
It uses edge locations (a network of small data centers) to cache copies of the data for the lowest latency. If the data is not present at edge locations, the request is sent to the source server, and data gets transferred from there.

AWS Transit Gateway

What is AWS Transit Gateway?

AWS Transit Gateway is a network hub used to interconnect multiple VPCs. It can be used to attach all hybrid connectivity by controlling your organization's entire AWS routing configuration in one place.

Transit Gateway vs. VPC peering:

Transit Gateway

It has an hourly charge per attachment in addition to the data transfer fees.
Multicast traffic can be routed between VPC

attachments to a Transit Gateway.
It provides Maximum bandwidth (burst) of 50 Gbps per Availability Zone per VPC connection.
Security groups feature does not currently work with Transit Gateway.

VPC peering
It does not charge for data transfer.
Multicast traffic cannot be routed to peering connections.
It provides no aggregate bandwidth.
Security groups feature works with intra-Region VPC peering.

Amazon Route 53
What is Amazon Route 53?
Route53 is a managed DNS (Domain Name System) service where DNS is a collection of rules and records intended to help clients/users understand how to reach any server by its domain name.
Route 53 hosted zone is a collection of records for a specified domain that can be managed together. There are two types of zones:
● Public host zone – It determines how traffic is routed on the Internet.
● Private hosted zone – It determines how traffic is routed within VPC.
Route 53 TTL (seconds):
● It is the amount of time for which a DNS resolver creates a cache information about the records and reduces the query latency.
● Default TTL does not exist for any record type but always specifies a TTL of 60 seconds or less so that clients/users can respond quickly to changes in health status.

AWS VPC
What is AWS VPC?
Amazon Virtual Private Cloud (VPC) is a service that allows users to create a virtual dedicated network for resources.
Security Groups:
Default Security Groups:-
Inbound rule - Allows all inbound traffic
Outbound rule - Allows all outbound traffic
Custom Security Groups:- (by default)
Inbound rule - Allows no inbound traffic
Outbound rule - Allows all outbound traffic
Network ACLs (access control list):
Default Network ACL:-
Inbound rule - Allows all inbound traffic
Outbound rule - Allows all outbound traffic

Custom Network ACL:- (by default)
Inbound rule - Denies all inbound traffic
Outbound rule - Denies all outbound traffic

AWS SNS (Simple Notification Service)
What is AWS SNS?
Amazon Simple Notification Service (Amazon SNS) is a web service that makes it easy to set up, operate, and send notifications from the cloud.
It provides developers with a highly scalable, flexible, and cost-effective approach to publish messages from an application and deliver them to subscribers or other applications. It provides push notifications directly to mobile devices and delivers notifications by SMS text messages, email to Amazon Simple Queue Service (SQS), or any HTTP client.
It allows developers to group multiple recipients using topics.
It consists of topics and subscribers.

Amazon Simple Queue Service (SQS)
What is Amazon Simple Queue Service (SQS)?
Amazon Simple Queue Service (SQS) is a serverless service used to decouple (loose couple) serverless applications and components.
The queue represents a temporary repository between the producer and consumer of messages.
It can scale up to 1-10000 messages per second.
The default retention period of messages is four days and can be extended to fourteen days.
SQS messages get automatically deleted after being consumed by the consumers.
SQS messages have a fixed size of 256KB.
There are two SQS Queue types:
Standard Queue -
● The unlimited number of transactions per second.
● Messages get delivered in any order.
● Messages can be sent twice or multiple times.
FIFO Queue -
● 300 messages per second.
● Support batches of 10 messages per operation, results in 3000 messages per second.
● Messages get consumed only once.

GitOps

https://www.linkedin.com/comm/pulse/gitops-demystified-why-your-kubernetes-should-follow-git-agnihotri-ld4xc?lipi=urn%3Ali%3Apage%3Aemail_email_series_follow_newsletter_01%3BOcdQzJxaRR2c3Lifb8MapA%3D%3D&midToken=AQETOR-trJbPgQ&midSig=2sJryMsxGn6rM1&trk=eml-email_series_follow_newsletter_01-newsletter_content_preview-0-readmore_button_&trkEmail=eml-email_series_follow_newsletter_01-newsletter_content_preview-0-readmore_button_-null-kr7ymj~ma85cb5y~u3-null-

null&eid=kr7ymj-ma85cb5y-u3&otpToken=MTMwNzFiZTUxMjJhY2RjMmI1MjcwZmViNDExYWUxYjY4OGNjZDE0MjkwYWU4ZTZjN2JjZjA2NmM0NzViNThmMGYwZDdkMGU5NmVlOGU2Yzk0ZWZkZGJjMzA5Yzg4ZWIxMzM0NWM3Mzk0Y2FhNWNmOWFiOTA4ZDBlLDEsMQ%3D%3D

---

1. What is GitOps?

GitOps is a modern way of managing infrastructure and application deployment using Git as the single source of truth.
But what does that really mean?
In traditional operations, we manage servers and apps manually or with scattered scripts. In GitOps, **you declare how your system should look in Git**, and an automated tool makes your Kubernetes cluster match that definition — always.
Imagine managing your entire infrastructure like code: not just your apps, but how many replicas you want, what secrets they need, the service ports — all written in files and stored in Git.
In simple terms, GitOps means *everything as code*: if it's a virtual resource (servers, networks, app settings, etc.), it's declared in code and tracked in Git. Then a GitOps tool continuously checks Git and makes sure your running system matches exactly what's in those files.

2. GitOps Principles (Made Super Simple)

There are **four key principles** that make up GitOps. Let's break them down with real-life analogies:

i.) Declarative Descriptions

Just like you write a shopping list before going to the grocery store, in GitOps, you write down everything your system needs — in YAML or JSON. You're not saying *how* to install it; you're saying *what* you want.

ii.) Versioned and Immutable

Everything is tracked in Git — every change, every line, every config. You can go back in time to see what changed, when, and why. Just like Google Docs history, but for your infrastructure.

iii.) Automated Delivery

As soon as you update Git, GitOps tools pick up the change and apply it to your cluster automatically — no waiting, no pushing manually. It's like using UPI auto-pay for your electricity bill instead of remembering to log in and pay every month.

iv.) Continuous Reconciliation

The GitOps operator checks your cluster *constantly*. If someone changes something manually in production, GitOps will notice and **restore it to match Git**. Think of it like a robot housekeeper that keeps rearranging your desk to match the layout you once approved in a picture.

3. Real-Life Analogy to Understand GitOps

Imagine you're the manager of a restaurant chain.

- You maintain a "Master Menu" in a Google Doc (your **Git repo**).
- Every outlet (your **Kubernetes clusters**) should follow this exact menu.
- Whenever you update the Google Doc (add a new burger), all branches should update automatically.
- If a chef at one branch secretly adds pineapple pizza 🍕, the manager bot notices the change and **restores the menu** to match the document.

That's GitOps. Git is the source of truth. Any unauthorized or accidental change gets reverted. Everything remains consistent, versioned, and under your control.

How did you like this analogy? If it made GitOps clearer for you, **share this post** so others can also get a taste of GitOps in the most relatable way! ♻️

4. GitOps Workflow in Kubernetes (Step-by-Step)

Let's go through the typical steps involved in a GitOps-powered Kubernetes deployment:

i.) Developer Commits Code

The developer commits application code along with the relevant Kubernetes configuration files (e.g., Deployment, Service) to a Git repository. These YAML files define the desired state of the application and infrastructure.

ii.) Continuous Integration (CI) Builds the Image

In the CI pipeline, such as GitHub Actions, Gitlab CI or Jenkins, the source code is built into a Docker image. This image is then pushed to a container registry, making it available for deployment in the Kubernetes cluster.

iii.) GitOps Tool Detects Change

A GitOps tool, like Argo CD, constantly monitors the Git repository for any changes. When the tool detects a change (such as an updated application version or changes to the Kubernetes configuration), it acknowledges the update and prepares to sync it with the cluster.

iv.) GitOps Syncs with Kubernetes

The GitOps tool compares the current state of the Git repository with the actual state of the Kubernetes cluster. It checks if the cluster configuration is aligned with the desired state defined in the Git repository.

v.) Reconciliation Happens

If the actual state in the cluster does not match the state described in Git (for example, a pod is missing, or a deployment is outdated), the GitOps tool automatically makes the necessary adjustments to bring the cluster into alignment with the Git-defined state. This ensures that the cluster is always in sync with the repository, enforcing the desired state with no manual intervention.

This process continuously repeats. For instance, if someone manually deletes a pod in the cluster, GitOps will detect the change and automatically recreate the pod to match the configuration stored in Git.

5. GitOps vs Traditional CI/CD

Here's a comparison between GitOps and traditional CI/CD:

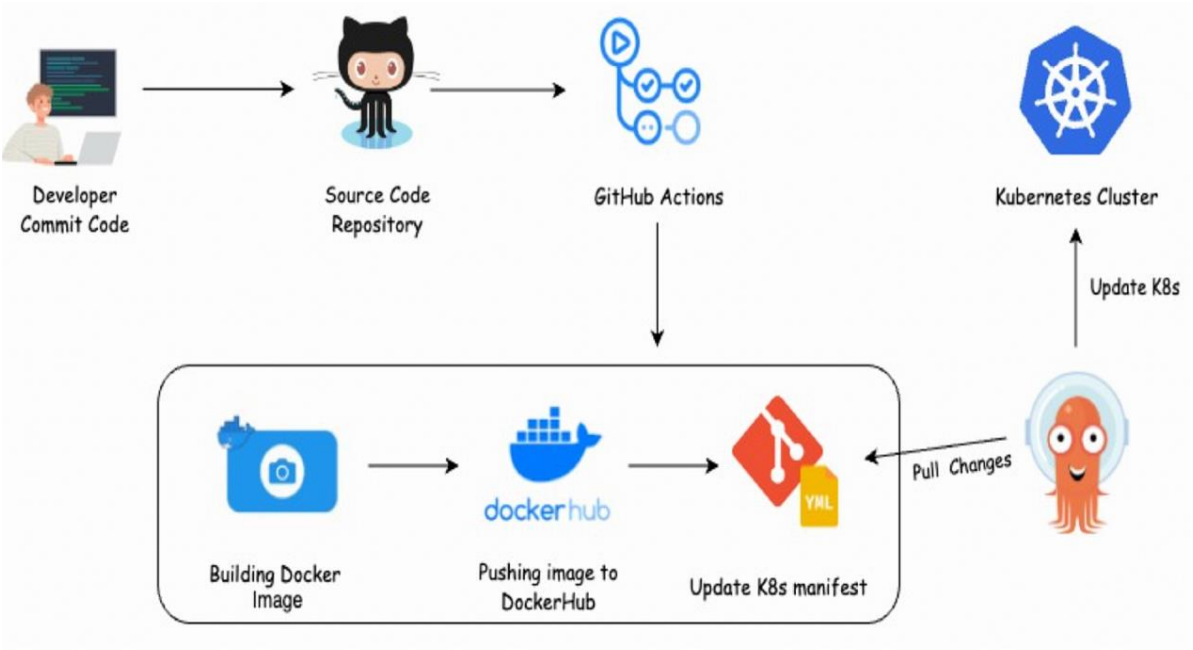
Deployments Triggered By:

- In traditional CI/CD, deployments are triggered by CI/CD pipelines.
- In GitOps, deployments are triggered by Git commits. Whenever a change is made in the Git repository, it triggers the process to update the Kubernetes cluster.

Infrastructure & Configuration Storage:

- In traditional CI/CD, infrastructure and configuration are usually split across different tools (e.g., Terraform for infrastructure, Helm for Kubernetes deployments).
- In GitOps, everything (infrastructure, application code, and Kubernetes configurations) is stored in Git. Git serves as the single source of truth.

OpenShift 4.X Foundations - Getting Started with GitOps
Dewan A. 5 years ago



Devops go web app - with Kubernetes/EKS, Github…
Ashish Wakde 5 months ago

Drift Detection:

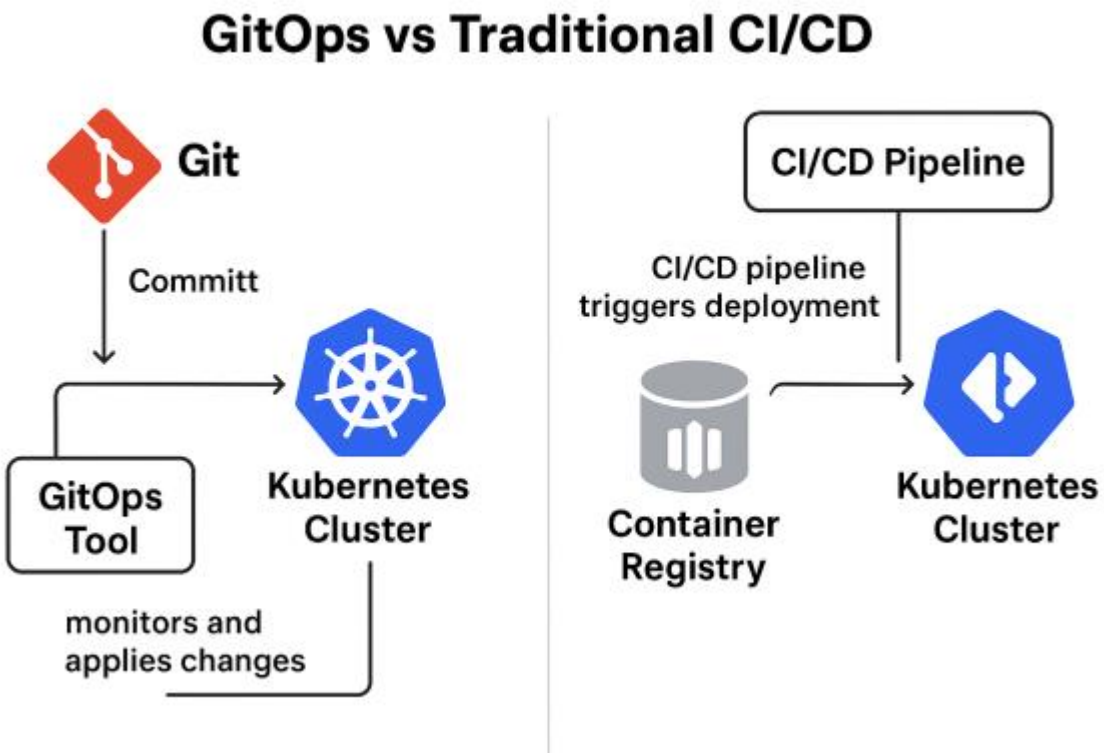
- In traditional CI/CD, drift (when the state of the system deviates from the desired configuration) is detected manually, requiring constant monitoring.
- GitOps automatically and continuously detects drift, comparing the current state of the cluster with the Git repository. If any mismatch is found, it triggers an automatic reconciliation.

Rollback:

- In traditional CI/CD, rolling back to a previous state can be challenging unless specifically scripted. You have to manually undo changes.
- In GitOps, rollback is easy. You simply revert the commit in Git, and the GitOps tool automatically brings the cluster back to the previous state.

Who Pushes to the Cluster:

- In traditional CI/CD, either the CI/CD tool or a manual engineer pushes changes to the Kubernetes cluster.
- In GitOps, the GitOps tool running inside the cluster pulls changes directly from Git and applies them.



GitOps vs Traditional CI/CD Pipeline: High level Overview

Think of GitOps as a **pull-based** approach, where the cluster pulls the changes from Git itself. In contrast, traditional CI/CD is **push-based**, where the CI/CD pipeline pushes changes into the cluster. This makes GitOps safer and more predictable, as the changes are always pulled from a single source of truth.

6. Benefits of GitOps You'll Actually Care About

i.) Simple and Reliable Rollbacks: If something goes wrong, you can easily revert to a previous state by rolling back a Git commit. No complex manual steps required.

ii.) Built-in Audit Trails: Every change is tracked in Git. You always know who made a change, what was changed, and when it happened — perfect for compliance and debugging.

iii.) Environment Consistency: All your environments (dev, staging, production) are kept in sync using the same Git source. No more configuration drift or unexpected differences.

iv.) Self-Healing Infrastructure: If someone makes an unauthorized manual change or if a resource is accidentally deleted, the GitOps tool will detect the drift and restore the correct state automatically.

v.) Faster Onboarding: New developers or DevOps engineers only need to learn Git to contribute. No need to dive deep into Kubernetes internals right away.

7. GitOps Tools (Just a Glimpse — We'll Deep Dive Later)

This article mainly focused on clearing the concepts of GitOps — the *why* and *how* behind it. But tools like Argo CD and Flux CD are what bring GitOps to life in real Kubernetes environments.

Here's just a quick sneak peek — we'll cover each of these in much more detail in a dedicated article later.

Argo CD

- A GitOps controller built specifically for Kubernetes.
- Comes with a clean visual dashboard to track deployments and sync status.
- Supports Helm, Kustomize, and raw YAML.
- Helps define applications as code using CRDs (Custom Resource Definitions).

→ Ideal if you prefer dashboards, fine-grained control, and multi-cluster setups.

Flux CD

- A lightweight, Git-native GitOps engine
- Integrates tightly with GitHub/GitLab workflows
- Perfect for those who love managing everything via code and CLI
- Fully supports the GitOps Toolkit (a set of reusable components)

→ Great for teams who prefer automation over visuals and like to keep everything lean and declarative.

8. Common Misconceptions Cleared

1. **GitOps = CI/CD** → Not quite. GitOps focuses only on the **CD (Continuous Delivery)** part. It complements your CI pipeline but doesn't replace it.
2. **I can't use** kubectl anymore? → You still can, but manual changes made directly to the cluster will be overwritten by the GitOps controller unless they're also committed to Git.
3. **It's only for big companies.** → Wrong. Even a solo developer or small startup can benefit — with GitOps, you get versioning, backups, and safer rollouts, all from day one.
4. **GitOps makes things slower due to Git dependency.** → Actually, GitOps brings more **predictability**, not slowness. Once set up, changes are applied faster and more reliably through automation.
5. **GitOps is just Infrastructure as Code (IaC).** → GitOps uses IaC, but it's more than that — it introduces a **real-time feedback loop** and constant state reconciliation, which most basic IaC setups don't provide.

9. Final Thoughts — Is GitOps Worth It?

GitOps isn't just a new tool or buzzword — it's a shift in how we manage infrastructure. By using Git as the single source of truth, teams gain **consistency, auditability, and confidence** in every deployment. Whether you're a solo engineer or part of a large platform team, GitOps brings structure and safety to your Kubernetes operations.

If you're just starting out, here's a simple path forward:

- Begin storing your Kubernetes manifests in Git.
- Introduce a GitOps tool like Argo CD or Flux.
- Set up a basic pipeline that syncs changes from Git to your cluster.

From there, you'll gradually see your deployments become more predictable, reliable, and easy to manage — with far fewer surprises in production.

GitOps doesn't require you to change everything overnight. But once you adopt it, you'll likely wonder how you ever deployed without it.

10. References and Next Steps

If this article helped clarify GitOps concepts for you, that was the goal. In the upcoming newsletter, we'll deep dive into GitOps tools like Argo CD and Flux — with step-by-step walkthroughs and a real-world Kubernetes example and demo.
So if you're eager to put GitOps into action, **stay subscribed and keep an eye out** for the next issue.

Useful References:

- **Official GitOps Website**
- **Argo CD Documentation by**
- **Flux Documentation**
- **GitOps Working Group (CNCF)**

Jenkins

Here is a comprehensive overview of Jenkins, covering its core concepts, architecture, usage, and best practices:

Jenkins Overview

Jenkins is an open-source automation server widely used for continuous integration (CI) and continuous delivery (CD). It helps automate the parts of software development related to building, testing, and deploying, facilitating CI/CD practices.

Key Concepts

1. Continuous Integration (CI):

CI is a development practice where developers integrate code into a shared repository frequently, preferably several times a day.

Jenkins helps to automate the build and test process every time a developer commits code, ensuring early detection of issues.

- **Definition:** Integrating the live changes of source code into master/production/central repository after being validated and tested.
  (or)
- Continuous Integration is a software development practice where developers regularly merge their code changes into a central repository after being validated and tested.

2. Continuous Delivery (CD):

CD is a practice where code changes are automatically built, tested, and prepared for release to production.

Jenkins can automate the deployment of applications, making it easier to deliver updates quickly and consistently.

- **Definition:** Continuous Delivery is a software development practice where source code changes are automatically built, tested, and prepared for release to production, but the actual release to production is done manually.

Continuous Deployment (CD):

- **Definition:** Continuous Deployment is a software development practice where every code change that passes automated tests is automatically released to production without manual intervention.

3. Pipeline:

A pipeline in Jenkins defines the entire workflow of building, testing, and deploying applications.

Jenkins pipelines can be scripted (Declarative or Scripted pipelines) using a domain-specific language (DSL).

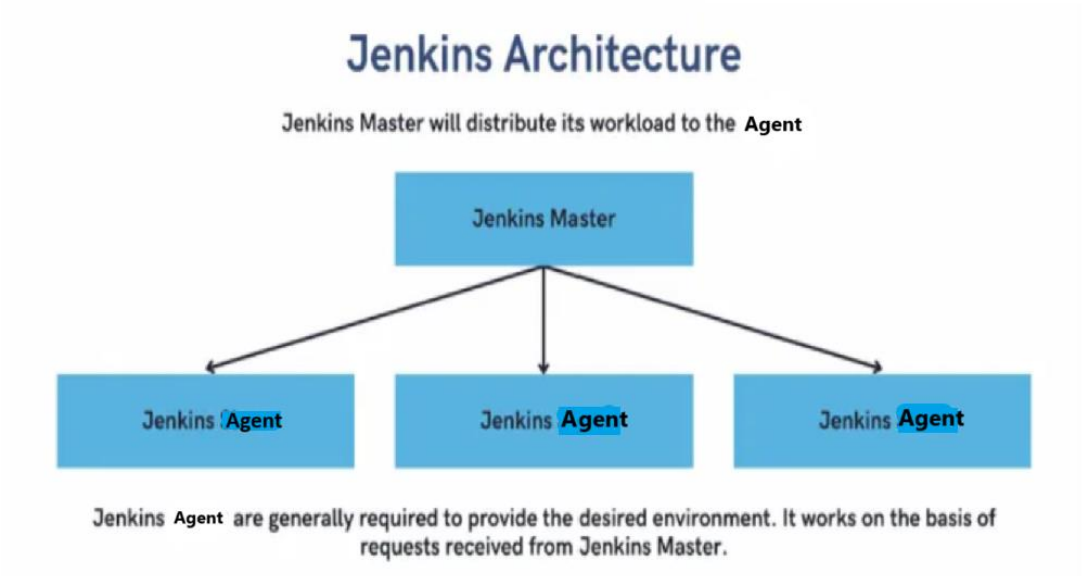
Pipeline:
Devops pipeline is a set of automated tools and processes that helps developer and operation professionals work together to build and deploy code to production environments.

4. Plugins:

Jenkins is highly extensible through plugins.

Plugins allow Jenkins to integrate with various development, testing, and deployment tools like Git, Maven, Docker, Kubernetes, etc.

Jenkins Architecture



1. Master-Agent Architecture:

Jenkins Master: The main Jenkins server, responsible for scheduling jobs, dispatching builds to Agent, and monitoring the state of agents.

Jenkins agent: Agents that execute jobs dispatched by the master.

2. Jobs/Builds:

Jenkins jobs or builds define tasks like compiling code, running tests, or deploying applications.

Types of Jenkins jobs:

Freestyle Projects: Basic jobs that allow a wide variety of build configurations.

Pipeline Projects: More advanced, script-based jobs.

Feature	Freestyle Project	Pipeline Project
Configuration	Form-based (UI)	Code-based (Jenkinsfile)
Complexity	Simple, limited flexibility	Highly flexible, supports complex workflows
Multi-stage Builds	Limited support	Full support for multi-stage workflows
Version Control	Not easily versioned	Version-controlled (via Jenkinsfile in SCM)
Parallel Execution	Limited, requires plugins	Full support using parallel step
Error Handling & Conditionals	Basic error handling	Advanced error handling, retries, conditionals

Feature	Freestyle Project	Pipeline Project
Reusability	Not easily reusable across projects	Can be reused and stored in version control
UI/Visualization	Basic visualization	Advanced pipeline visualization
Use Case	Simple, individual tasks	Complex workflows, multi-stage, large projects

In summary:

- **Freestyle Projects** are great for simple, one-off tasks where you don't need complex workflows or configurations.
- **Pipeline Projects** are more suitable for larger projects that require flexible, maintainable, and versioned CI/CD pipelines with multiple stages, parallel execution, and advanced error handling.

3. Workspace:

Each Jenkins job has a workspace directory on the Jenkins agents, where it performs build operations.

Setting Up Jenkins

1. Installation:

Jenkins can be installed on various platforms, including Windows, macOS, and Linux.

Installation methods include direct download of Jenkins WAR file, Docker image, or package managers like apt or yum.

2. Initial Configuration:

Post-installation, Jenkins requires setting up the administrator password and installing plugins.

Configure Jenkins URL, manage credentials, and connect to version control systems like Git.

Configuring Jenkins

1. Global Tool Configuration:

Set up JDK, Maven, Ant, and other build tools required by Jenkins jobs.

2. Creating a Job:

Define the source code repository (Git, SVN, etc.).

Set up build triggers (e.g., polling the repository or receiving webhooks).

Define the build steps (e.g., execute shell commands, invoke Ant, or run a Maven target).

Post-build actions (e.g., publish reports or send notifications).

3. Pipelines:

Declarative Pipeline:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
        }
        stage('Deploy') {
            steps {
                sh 'mvn deploy'
            }
        }
    }
}
```

Scripted Pipeline:

```
node {
    stage('Build') {
        sh 'mvn clean install'
    }
    stage('Test') {
        sh 'mvn test'
    }
    stage('Deploy') {
        sh 'mvn deploy'
    }
}
```

Best Practices

1. Use Pipelines:

Prefer pipelines over freestyle projects for better scalability and maintainability.

2. Modular Jobs:

Create modular jobs with shared libraries to reduce code duplication.

3. Security:

Enable Role-Based Access Control (RBAC) for better security.

Manage credentials securely using Jenkins' credentials management.

4. Monitoring and Maintenance:

Regularly update Jenkins and its plugins.

Monitor Jenkins performance and scale Jenkins by adding more slave nodes if necessary.

5. Backup and Recovery:

Regularly back up Jenkins configuration and job configurations.

Use plugins like **ThinBackup** for automated backups.

In Jenkins, the **Manage Jenkins** section is the central place for administering the Jenkins server. Here's an explanation of the different options you typically find under Manage Jenkins:

1. System Configuration

Configure System:

Allows configuring global Jenkins settings, such as environment variables, default paths, email notifications, JDK, Git installations, and other tools.

Global Tool Configuration:

Used to set up global tools like JDKs, Maven, Ant, Gradle, Git, and others that are used across different jobs.

2. Security

Manage Users:
Used to create, update, delete, or manage user accounts within Jenkins.

Configure Global Security:
Controls overall security settings like enabling/disabling security, configuring authentication methods (LDAP, Jenkins' own database, etc.), and authorization strategies (e.g., Matrix-based security, Role-based security).

Credentials:
A centralized location to manage credentials like SSH keys, tokens, passwords, which can be used in jobs and pipelines.

3. System Maintenance

Manage Nodes and Clouds:
Allows you to manage Jenkins agents (nodes). You can configure, connect, or disconnect slave nodes and clouds for distributed builds.

Reload Configuration from Disk:
Reloads Jenkins configuration from disk, useful when making manual changes to configuration files.

Prepare for Shutdown:
Gracefully stops Jenkins from accepting new builds and waits for ongoing builds to finish before shutting down.

4. Monitoring and Logs

System Log:
Displays and manages Jenkins logs. You can add custom loggers for different components.

Load Statistics:
Shows system load statistics, such as the number of executors busy, build queue length, etc.

Monitor Jenkins:
Provides an overview of the Jenkins system's health and status, often with suggestions for improving performance.

5. Plugins Management

Manage Plugins:
This is where you can install, update, disable, or remove plugins. The plugin management interface has tabs for:

Updates: Displays available plugin updates.
Available: Shows plugins that can be installed.
Installed: Lists already installed plugins.
Advanced: Provides options to upload plugins manually or manage plugin repositories.

6. Tools and Actions

Script Console:
A powerful tool that allows administrators to run Groovy scripts to perform administrative tasks programmatically.

Manage Old Data:
Used to manage obsolete data that might be left behind by old or removed plugins.

System Information:
Displays detailed information about the Jenkins environment, such as JVM properties, environment variables, and system properties.

Jenkins CLI:
Provides details on how to use Jenkins Command Line Interface (CLI) to automate various tasks.

7. Backup and Restore

ThinBackup (if installed):
A plugin that provides options for backing up and restoring Jenkins configurations.

8. Miscellaneous

About Jenkins:
Displays the Jenkins version and information about the project.

New Item:
Shortcut to create a new Jenkins job or pipeline.

Manage Credentials:
Same as the Credentials option mentioned above, it provides a direct route to managing stored secrets.

9. Advanced Options

Reload from Disk:
Reloads the Jenkins configuration from the file system. This is useful if you manually edit the configuration files outside the Jenkins UI.

Restart Jenkins:
Provides an option to restart Jenkins from the UI.

Jenkins Notes from Scratch
https://media.licdn.com/dms/document/media/v2/D561FAQH2BicVz0RBrg/feedshare-document-pdf-analyzed/feedshare-document-pdf-

analyzed/0/1731904562911?e=1736985600&v=beta&t=xOoI1hda7w5TyTQnZMJCuhADelpTP6AK7yw_iTSEAbw

SonarQube

SonarQube architecture
Set the some rule on java code.
Generate the sonarqube reports
User sonar scanner plugin

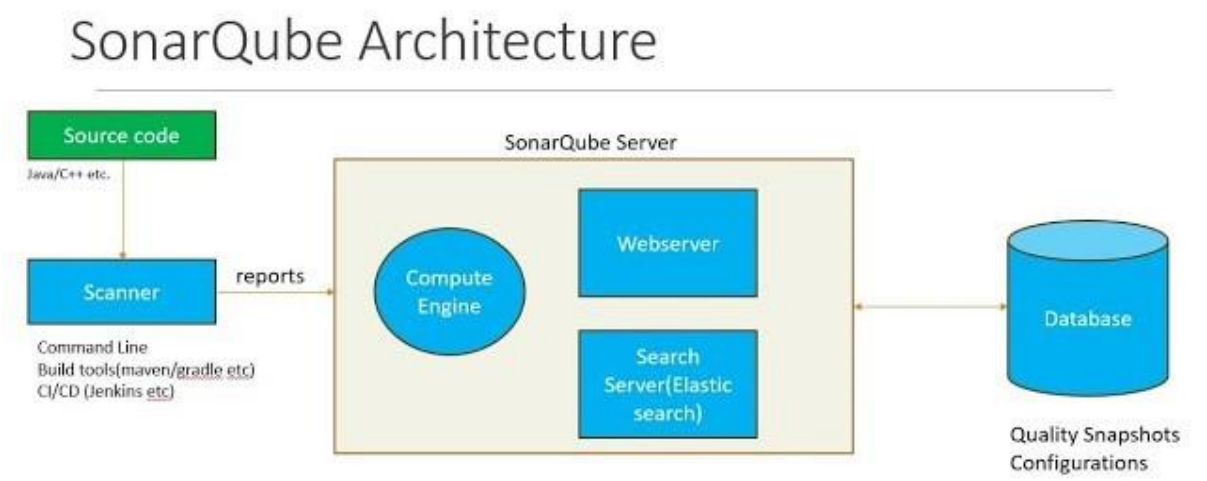
SonarQube is a Code Quality Assurance tool that collects and analyzes source code, and provides reports for the code quality of project. It combines static and dynamic analysis tools and enables quality to be measured continually over time

Sonarqube architecture

Architecture

Concept	Definition
Analyzer	A client application that analyses the source code to compute snapshots.
Database	Stores configuration and snapshots

Server	Web interface that is used to browse snapshot data and make configuration changes



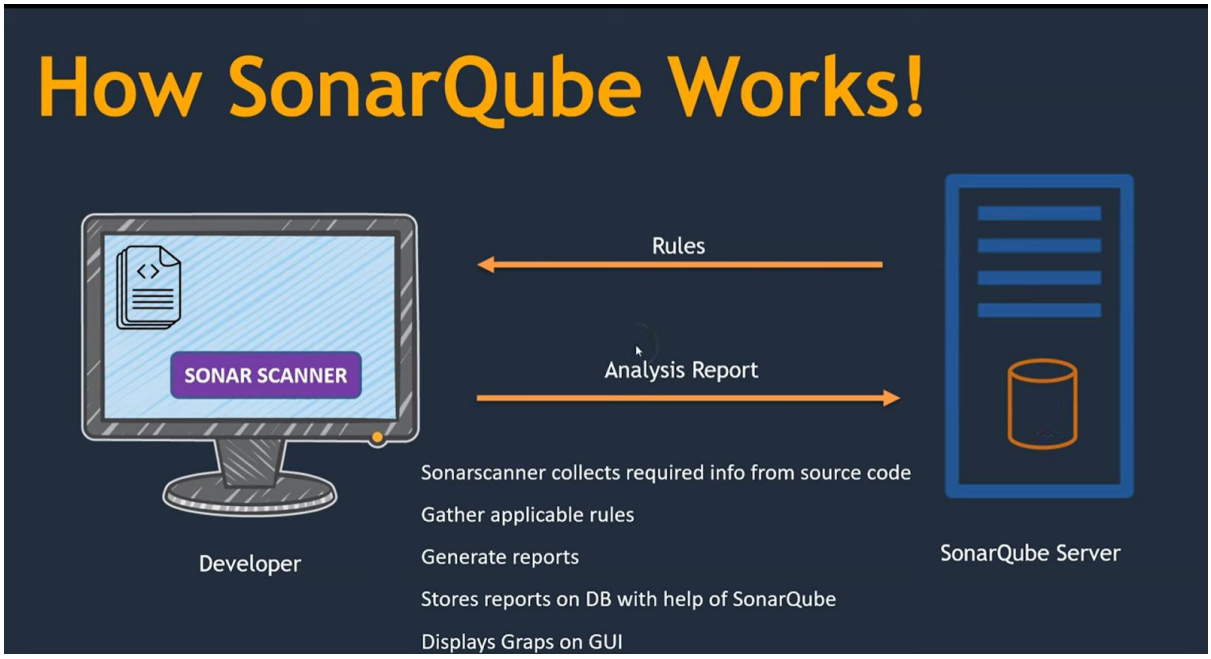
Code Quality
Sonarqube:
Advantages:
It acts as a quality management tool.

- Code analysis
- Test reports
- Code coverage, etc

Component of sonarqube

1. SonarQube server.
    - **Rules :** Rules are instructions need to follow while writing the code.
    - **Database :** Database will store the analysis reports.
    - **Web Interface:** Once analysis reports are stored in the database through the web interface, then we can see and understand easily.
    - **Elastic search:** it helps to search required data from sonarqube database.
2. SonarScanner:
    - Source code.

Sonar Scanner is a service or agent which runs on the system where source code exists.
Once sonar scanner run the scan, it is going to gather the report and that will be published into the sonarqube server



- Sonarscanner collects required info from source code
- Gather applicable rules from sonarqube server
- Generate reports
- Stores reports on DB with help of SonarQube
- Displays Graphs on GUI

To create a quality gate in SonarQube

navigate to the "Quality Gates" section within your SonarQube instance, click "Create", give your quality gate a name, and then add conditions by selecting metrics, operators, and thresholds that define when the quality gate will pass or fail based on your desired code quality standard.
Key steps:

- **Access Quality Gates:** Go to the "Quality Gates" tab in your SonarQube dashboard.
- **Create a new Quality Gate:** Click "Create" to start building a new quality gate.
- **Name your Quality Gate:** Provide a descriptive name for your quality gate.
- **Add Conditions:**
    - Select a metric you want to evaluate (e.g., code coverage, complexity, number of issues).
    - Choose a comparison operator (e.g., greater than, less than, equals).
    - Set a threshold value.
- **Define Scope (optional):** Specify whether the conditions should apply to new code only or the entire codebase.

I understand now! You're looking for all the individual tool configurations and scripts that support a full production environment. You want to see the code for each of the different tools involved, such as **Dockerfile**, **Helm charts**, **Kubernetes configurations**, and so on, in a production environment. Let's take an example of a **Java-based application** with **Docker**, **Helm**, **Kubernetes**, and **GitOps** tools in the CI/CD pipeline.

---

1. Dockerfile (for Java-based application)
In the **CI pipeline**, we build a Docker image. Here's an example of a **Dockerfile** for a Java-based application:

```
# Use an official Java runtime as a parent image
FROM openjdk:11-jdk-slim

# Set the working directory in the container
WORKDIR /app

# Copy the local jar file into the container
COPY target/your-app.jar /app/your-app.jar

# Set the command to run the app
ENTRYPOINT ["java", "-jar", "your-app.jar"]

# Expose the port the app will run on
EXPOSE 8080
```

Explanation:

- **FROM openjdk:11-jdk-slim**: Pulls the OpenJDK 11 image as the base image.
- **COPY target/your-app.jar**: Copies the built application JAR file from the local target/ directory to the container.
- **ENTRYPOINT ["java", "-jar", "your-app.jar"]**: Specifies the command to run the application when the container starts.
- **EXPOSE 8080**: Exposes port 8080, which is typically used by web applications.

---

2. Helm Chart (for deploying application to Kubernetes)

For the **CD pipeline**, we deploy the application using Helm. Here's an example of a **Helm Chart** (with **values.yaml**, **deployment.yaml**, and **service.yaml**) that deploys the Java application to a Kubernetes cluster.

Helm Chart Structure:

```
your-app-helm/
├── charts/
├── templates/
│   ├── deployment.yaml
│   └── service.yaml
├── values.yaml
└── Chart.yaml
```

Chart.yaml:

```
apiVersion: v2
name: your-app
description: A Java-based application for Kubernetes
version: 0.1.0
appVersion: "1.0"
```

values.yaml (where we specify configurations such as the Docker image):

```
image:
  repository: your-docker-registry.com/your-app
  tag: latest  # This will be replaced by the pipeline with the correct image tag
  pullPolicy: Always

replicaCount: 3

service:
  type: ClusterIP
  port: 8080

resources: {}
```

deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}
  labels:
    app: {{ .Release.Name }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Release.Name }}
  template:
    metadata:
      labels:
```

```
      app: {{ .Release.Name }}
  spec:
    containers:
      - name: {{ .Release.Name }}
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
        ports:
          - containerPort: {{ .Values.service.port }}
        resources: {{ .Values.resources | toYaml | nindent 12 }}
```

service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}
spec:
  selector:
    app: {{ .Release.Name }}
  ports:
    - protocol: TCP
      port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.port }}
  type: {{ .Values.service.type }}
```

Explanation:

- **values.yaml**: Contains the values to be used in the Helm templates (e.g., Docker image repository and tag).
- **deployment.yaml**: Defines the Kubernetes deployment, including the Docker image, replicas, and resource settings.
- **service.yaml**: Exposes the application as a Kubernetes service on port 8080.

---

3. Kubernetes Configuration (for setting up the cluster)

For deploying the app, the **Kubernetes configuration** includes a **kubeconfig** file (typically set up in Jenkins or other CI/CD tools to interact with the Kubernetes cluster). A typical **kubeconfig** looks like this:

kubeconfig.yaml:

```
apiVersion: v1
clusters:
- cluster:
    server: https://<K8S_API_SERVER>
    certificate-authority-data: <CA_CERTIFICATE>
  name: your-cluster
contexts:
- context:
    cluster: your-cluster
    user: your-user
  name: your-context
```

```
current-context: your-context
kind: Config
preferences: {}
users:
- name: your-user
  user:
    client-certificate-data: <CERTIFICATE_DATA>
    client-key-data: <CLIENT_KEY>
```

Explanation:

- **server**: The Kubernetes API server endpoint.
- **certificate-authority-data**, **client-certificate-data**, **client-key-data**: TLS certificates for secure communication with the Kubernetes cluster.

---

4. GitOps (for syncing the application state in Git)

To maintain the declarative state, the **GitOps pipeline** works by updating the Git repository that holds the Helm values. Here's an example of the **GitOps values.yaml** file and the necessary commands to update it.

GitOps values.yaml:

This file represents the source of truth for the cluster's application state, stored in a Git repository.

```
image:
  repository: your-docker-registry.com/your-app
  tag: latest  # This will be updated in the GitOps pipeline with the new image tag
```

GitOps Pipeline Example:

Here's the **Jenkins pipeline** script to update the GitOps repository:

```
pipeline {
    agent any
    environment {
        GIT_REPO = 'https://github.com/your-org/your-gitops-repo.git'
        GIT_BRANCH = 'main'
        IMAGE_TAG = "${env.BUILD_NUMBER}"  // This will be passed from the CI pipeline
        APP_NAME = 'your-app'
        DOCKER_REGISTRY = 'your-docker-registry.com'
        HELM_VALUES_FILE = 'path/to/values.yaml'  // GitOps values file
    }
    stages {
        stage('Checkout GitOps Repo') {
            steps {
                git branch: "${GIT_BRANCH}", url: "${GIT_REPO}"
            }
        }
        stage('Update Helm Values') {
            steps {
                script {
                    sh """
                    sed -i 's|image: .*|image: ${DOCKER_REGISTRY}/${APP_NAME}:${IMAGE_TAG}|'
${HELM_VALUES_FILE}
```

```
            """
          }
        }
      }
      stage('Commit and Push Changes') {
        steps {
          script {
            sh """
            git config user.email "ci-cd@example.com"
            git config user.name "Jenkins CI/CD"
            git add ${HELM_VALUES_FILE}
            git commit -m "Update image tag to ${IMAGE_TAG}"
            git push origin ${GIT_BRANCH}
            """
          }
        }
      }
    }
    post {
      success {
        echo 'GitOps repository updated successfully!'
      }
      failure {
        echo 'Failed to update GitOps repository.'
      }
    }
}
```

Explanation:

- **GitOps**: This pipeline checks out the GitOps repository, updates the Helm values file with the new image tag, commits, and pushes it back to the repository.
- **sed command**: Dynamically updates the Docker image tag in the Helm values file.

---

5. Jenkins Pipeline Configuration (for CI/CD)

For Jenkins, you would need to define **credentials** (such as Docker registry and Kubernetes credentials) and pipeline settings. Here's an example of how to configure the Jenkinsfile for CI/CD.

Jenkinsfile for CI/CD:

```
pipeline {
  agent any
  environment {
    DOCKER_REGISTRY = 'your-docker-registry.com'
    APP_NAME = 'your-app'
    IMAGE_TAG = "${env.BUILD_NUMBER}"
    KUBE_CONFIG = '/path/to/kubeconfig'
    HELM_RELEASE_NAME = 'your-app-release'
    HELM_CHART_PATH = './helm/your-app'
    NAMESPACE = 'production'
  }
```

```
    stages {
        stage('Checkout Code') {
            steps {
                git branch: 'main', url: 'https://github.com/your-org/your-repo.git'
            }
        }
        stage('Build Application') {
            steps {
                sh 'mvn clean package -DskipTests'
            }
        }
        stage('Build Docker Image') {
            steps {
                sh "docker build -t ${DOCKER_REGISTRY}/${APP_NAME}:${IMAGE_TAG} ."
            }
        }
        stage('Push Docker Image') {
            steps {
                withDockerRegistry([credentialsId: 'docker-credentials', url:
"https://${DOCKER_REGISTRY}"]) {
                    sh "docker push ${DOCKER_REGISTRY}/${APP_NAME}:${IMAGE_TAG}"
                }
            }
        }
        stage('Deploy with Helm') {
            steps {
                withEnv(["KUBECONFIG=${KUBE_CONFIG}"]) {
                    sh """
                    helm upgrade --install ${HELM_RELEASE_NAME} ${HELM_CHART_PATH} \
                    --namespace ${NAMESPACE} \
                    --set image

.repository=${DOCKER_REGISTRY}/${APP_NAME}
 --set image.tag=${IMAGE_TAG} """ } } } } }
```

Summary of Tools and Configurations:

- Dockerfile: For building a Docker image.
- Helm Chart: For deploying the application on Kubernetes.
- Kubernetes Config (kubeconfig): For accessing the Kubernetes cluster.
- GitOps: For declarative configuration management using Git repositories.
- Jenkins Pipeline: For orchestrating the CI/CD process, integrating Docker, Helm, and GitOps.

This is the full set of tools and configurations you would typically use in a production environment to automate the process of building, testing, and deploying an application with Docker, Helm, Kubernetes, and GitOps.

what are type of pod crashes in kubernates?

Kubenates
Errors:
CrashLoopBackOff:

In Kubernetes, pods can crash for various reasons, and understanding the types of crashes can help with troubleshooting. Here are some common types of pod crashes:

1. CrashLoopBackOff
- Description: This occurs when a pod fails to start, crashes, and then is restarted repeatedly within a short time frame.
- Symptoms: The pod's status is `CrashLoopBackOff`, meaning it keeps crashing and Kubernetes is trying to restart it.
- Common Causes: Application errors, misconfigurations, dependency failures.

2. Error
- Description: The container inside the pod exits with a non-zero exit code.
- Symptoms: The pod may show `Error` status in `kubectl get pods`.
- Common Causes: Runtime errors in the application, missing environment variables, or incorrect command-line arguments.

3. OOMKilled
- Description: This occurs when the pod is killed by the system because it exceeds its memory limits.
- Symptoms: The pod's exit reason will show `OOMKilled` in the events.
- Common Causes: Insufficient memory allocation, memory leaks in the application.

4. Completed
- Description: This status indicates that the pod has finished executing its tasks successfully but is no longer running.
- Symptoms: The pod will show as `Completed` in the status.
- Common Causes: Job or batch processing that completes its execution as intended.

5. Crash Due to Resource Limits
- Description: If the pod hits its CPU limits, it may be throttled, causing it to crash or become unresponsive.
- Symptoms: The application may perform poorly or crash under load.
- Common Causes: Underestimating resource requirements.

6. Application-Specific Crashes
- Description: Applications can crash for various reasons specific to their logic or runtime environment.
- Symptoms: Specific error messages or stack traces in logs.
- Common Causes: Bugs in the application code, exceptions that aren’t handled properly.

7. Failed Scheduling

- Description: This occurs when a pod cannot be scheduled due to lack of resources or node availability.
- Symptoms: The pod status shows `Pending` with events indicating scheduling issues.
- Common Causes: Insufficient resources in the cluster, unsatisfied node selectors or taints.

8. Misconfiguration
- Description: Errors related to incorrect configurations such as invalid environment variables, secrets, or config maps.
- Symptoms: Pods may crash during startup.
- Common Causes: Typographical errors, missing configurations.

Summary

Understanding these crash types can help you diagnose issues effectively. Always start by checking the pod's status, logs, and events to determine the specific cause of the crash. If you need assistance with a particular case, feel free to share details!

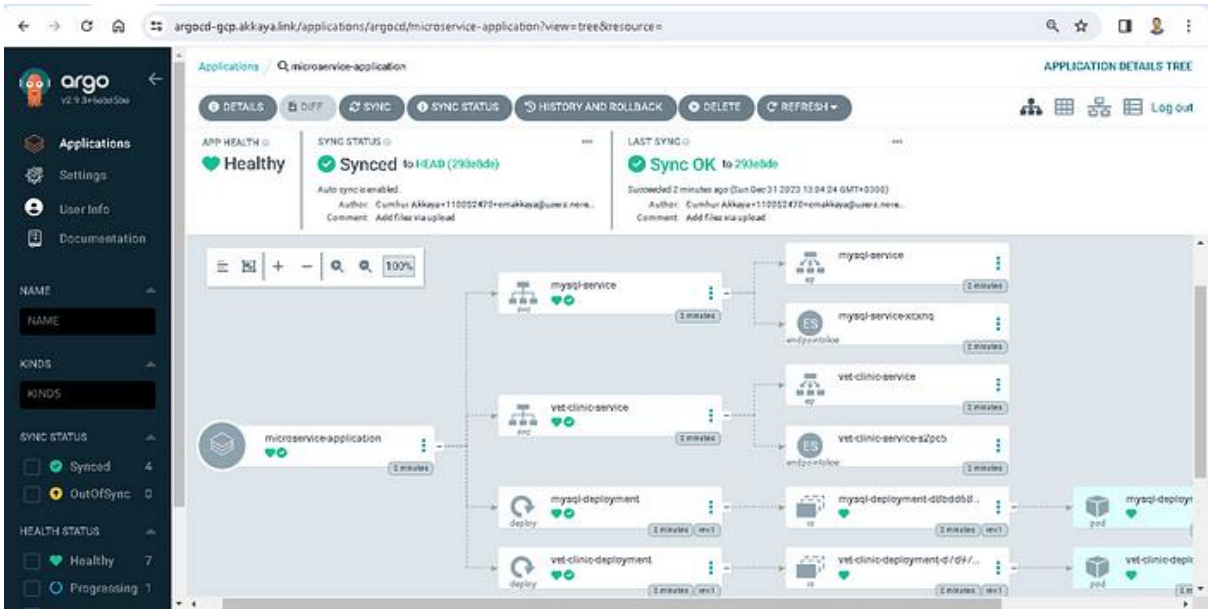
https://cmakkaya.medium.com/argo-cd-1-understanding-installing-and-using-argo-cd-as-a-gitops-continuous-delivery-tool-0ec0b4e00a77

1. What is Argo CD

Argo CD is implemented as a Kubernetes controller which continuously monitors running applications and compares the current, live state against the desired target state

Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes.
With Argo CD, applications are automatically and continuously distributed to the target environments. It provides ease of deployment and management to multiple Kubernetes Clusters. User definitions and authorization procedures can be performed. SSO integration is possible. Rollback can be made to any commit in the Git repo. Kubernetes objects can be synchronized manually or automatically to the desired state specified in the Git repo. Also included is the Argo CD CLI for Continuous Integration automation.

ArgoCD functions as a Kubernetes controller that **continuously monitors a Git repository for changes to application configurations, and then automatically applies those changes to the live Kubernetes cluster, ensuring the cluster state always aligns with the desired state defined in the Git repository,** essentially acting as a "**pull-based**" continuous delivery (CD) tool based on the **GitOps principle where Git is the single source of truth for application** deployments; meaning any updates to your application are made through Git commits, which ArgoCD then detects and automatically applies to the cluster, keeping it synchronized with the latest version in the Git repo.



How ArgoCD works in practice:

1. Developer commits changes:

A developer makes changes to their application configuration files (Kubernetes manifests) and commits them to the Git repository.

2. ArgoCD detects changes:

ArgoCD continuously monitors the Git repository and detects the new commit.

3. Comparison with live state:

ArgoCD compares the new configuration in Git with the current state of the application running on the Kubernetes cluster.

4. Apply changes if needed:

If differences are found, ArgoCD applies the necessary changes to the cluster to match the desired state defined in Git.

We can easily do these with argo CD;

- Application definitions, configurations, and environments; It is done in a declarative and version-controlled manner.
- Application deployment and lifecycle management; It is automated, auditable, and easily understandable.

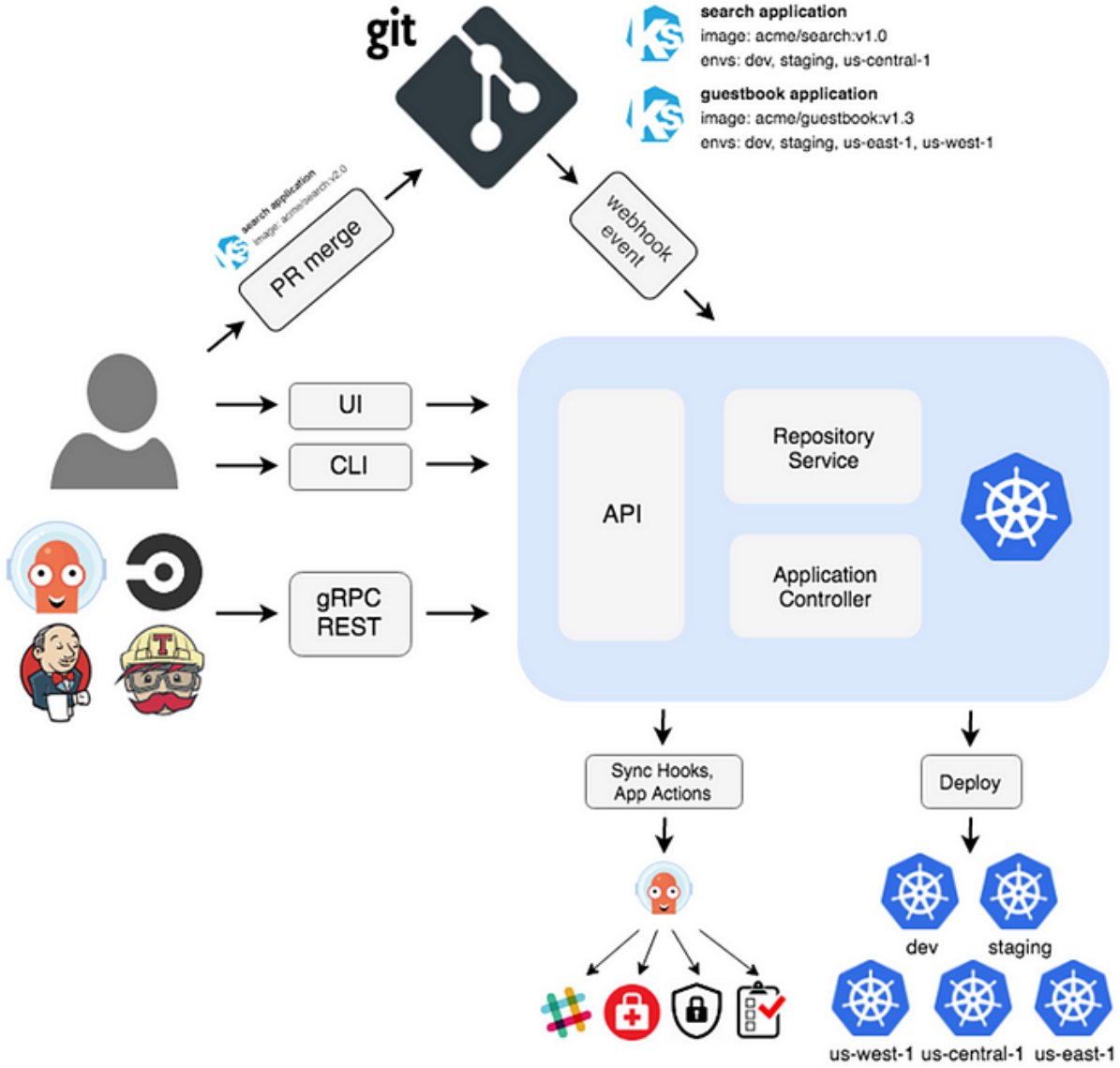
Argo CD is composed of three main components as seen in the picture below; (1)

- **API Server:** Exposes the API for the WebUI / CLI / CICD Systems
- **Repository Server:** Internal service that maintains a local cache of the git repository holding the application manifests
- **Application Controller:** Kubernetes controller which controls and monitors applications continuously and compares that current live state with the desired target state (specified in the repository). If a OutOfSync is detected, it will take corrective actions.

Architecture Diagram

To visualize the architecture:

1. Git Repository serves as the source of truth for application configurations.
2. API Server communicates with users via CLI/Web UI.
3. Repository Server processes manifests from the Git repository.
4. Controller performs reconciliation by applying changes to Kubernetes clusters.
5. Kubernetes Cluster contains the deployed resources, which are monitored by Argo CD.



Argo CD Commands List

Here's a list of commonly used Argo CD CLI commands:

1. Login and Configuration
- Login to Argo CD server
  bash
  argocd login <ARGOCD_SERVER> --username <USERNAME> --password <PASSWORD>

- Set default project or cluster context
  bash
  argocd context --set <CONTEXT_NAME>

- List current contexts
  bash
  argocd context

2. Application Management
- Create an application
  bash
  argocd app create <APP_NAME> \
    --repo <REPO_URL> \
    --path <PATH_IN_REPO> \
    --dest-server <K8S_CLUSTER_URL> \
    --dest-namespace <NAMESPACE>

- List all applications
  bash
  argocd app list

- Get details of an application
  bash
  argocd app get <APP_NAME>

- Sync an application
  bash
  argocd app sync <APP_NAME>

- Delete an application
  bash
  argocd app delete <APP_NAME>

3. Application Operations
- Pause automated sync
  bash
  argocd app pause <APP_NAME>

- Resume automated sync
  bash
  argocd app resume <APP_NAME>

- Check the sync status
  bash
  argocd app sync-status <APP_NAME>

- Force sync
  bash
  argocd app sync <APP_NAME> --force

4. Cluster Management
- Add a Kubernetes cluster
  bash
  argocd cluster add <K8S_CLUSTER_CONTEXT>

- List all clusters
  bash
  argocd cluster list

- Remove a cluster
  bash
  argocd cluster rm <K8S_CLUSTER_CONTEXT>

5. Projects Management
- Create a new project
  bash
  argocd proj create <PROJECT_NAME>

- List all projects
  bash
  argocd proj list

- Add a destination to a project
  bash
  argocd proj add-destination <PROJECT_NAME> <SERVER_URL> <NAMESPACE>

- Delete a project
  bash
  argocd proj delete <PROJECT_NAME>

6. Repository Management
- Add a Git repository
  bash
  argocd repo add <REPO_URL> --username <USERNAME> --password <PASSWORD>

- List all repositories
  bash
  argocd repo list

- Remove a repository
  bash
  argocd repo rm <REPO_URL>

Argo CD Architecture

The architecture of Argo CD is designed to manage Kubernetes resources declaratively. Below is a breakdown of its key components:

1. User Interface
- **Web UI:**
  A user-friendly dashboard for managing applications, clusters, and sync operations.
- CLI:
  Command-line interface for executing all actions programmatically.

2. Core Components
- **API Server:**
  Exposes REST API endpoints to interact with Argo CD.
- **Repository Server:**
  Fetches manifests from Git repositories and handles Helm chart rendering.
- **Controller:**
  Watches applications and syncs Kubernetes clusters with the desired state.

3. Data Stores
- **Git Repository:**
  Stores the desired state of the application in manifest files or Helm charts.
- **Cluster State:**
  Argo CD fetches the live state of resources from Kubernetes clusters.

4. Kubernetes Integration
- **Custom Resource Definitions (CRDs):**
  Argo CD uses CRDs such as `Application` and `ApplicationSet` to define application configurations.
- **Namespaces:**
  Applications are deployed into specified namespaces.

5. Sync Mechanism

- Declarative Sync:

Ensures the live state matches the desired state defined in Git repositories.

- Automated Sync Policies:

Automatically syncs resources and performs self-healing.

Let me know if you'd like a detailed diagram or explanation for specific components!
Here’s a concise version of the notes with YAML code for quick reference:

1. Deploying with Kubernetes Manifest Files

YAML Example:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-manifests
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/your-repo/nginx-k8s-manifests
    targetRevision: HEAD
    path: manifests
  destination:
    server: https://kubernetes.default.svc
    namespace: nginx
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

2. Deploying with Helm Charts

a. From a Git Repository

YAML Example:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
```

```
metadata:
  name: nginx-helm
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/your-repo/nginx-helm-chart
    targetRevision: HEAD
    path: charts/nginx
    helm:
      parameters:
        - name: replicaCount
          value: "3"
        - name: service.type
          value: LoadBalancer
  destination:
    server: https://kubernetes.default.svc
    namespace: nginx
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

b. From a Helm Repository

YAML Example:

yaml

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-helm-repo
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://charts.bitnami.com/bitnami
    chart: nginx
    targetRevision: 13.2.8
    helm:
      parameters:
        - name: replicaCount
          value: "2"
        - name: service.type
          value: ClusterIP
  destination:
    server: https://kubernetes.default.svc
    namespace: nginx
  syncPolicy:
```

```
    automated:
      prune: true
      selfHeal: true
```

3. Deploying Across Multiple Clusters

Add Target Cluster

bash

```
argocd cluster add <target-cluster-context>
argocd cluster list
```

Deploy Application to Target Cluster

YAML Example:

yaml

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: nginx-app
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/your-repo/nginx-app
    targetRevision: HEAD
    path: manifests
  destination:
    server: https://<target-cluster-server-url>
    namespace: nginx
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

4. Blue-Green Deployment with Argo CD

ApplicationSet YAML Example:

yaml

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: nginx-bluegreen
```

```
spec:
  generators:
    - list:
        elements:
          - cluster: blue
            namespace: nginx-blue
            values:
              version: "1.21.6-blue"
          - cluster: green
            namespace: nginx-green
            values:
              version: "1.21.6-green"
  template:
    metadata:
      name: nginx-{{ cluster }}
    spec:
      project: default
      source:
        repoURL: https://github.com/your-repo/nginx-bluegreen
        targetRevision: HEAD
        path: manifests/{{ cluster }}
      destination:
        server: https://kubernetes.default.svc
        namespace: nginx-{{ namespace }}
      syncPolicy:
        automated:
          prune: true
          selfHeal: true
```

Switch Traffic

Modify the Kubernetes service selector to change traffic between blue and green environments:

yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: default
spec:
  selector:
    app: nginx-blue # Change to nginx-green after verification
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

5. Sync and Verify Deployments

- Sync Application:
  bash
  argocd app sync nginx-app

- Verify Resources:
  bash
  kubectl get all -n <namespace>

Let me know if you'd like further clarifications!

Linux

what is difference between root user and sudo user in linux?
The **root user** has full, unrestricted access to the system, while a **sudo user** is a regular user granted temporary root privileges to execute certain administrative tasks.
Using sudo is generally safer and more manageable than working directly as the root user.

Users & Groups.

User:
Some Important Points related to Users:

- Users and groups are used to control access to files and resources
- Users login to the system by supplying their username and password
- Every file on the system is owned by a user and associated with a group
- Every process has an owner and group affiliation, and can only access the resources its owner or group can access.
- Every user of the system is assigned a unique user ID number (the UID) Users name and UID are stored in /etc/passwd /etc/pa
- User's password is stored in /etc/shadow in encrypted form.
- Users are assigned a home directory and a program that is run when they login (Usually a shell)
- Users cannot read, write or execute each other's files without permission.

Whenever a user is created in Linux things created by default:

- A home directory is created(/home/username)
- A mail box is created(/var/spool/mail)
- unique UID & GID are given to user

Passwd file
/etc/passwd

Group file
The file /etc/group stores group information. Each line in this file stores one group entry.

1. **/etc/group**

ADD USER, SET PASSWORD & SWITCH TO USER

2. **Id user/group**

ADD USER, GROUP & USER INTO GROUP

3. **The /etc/shadow file**

This file stores users' password and password related information. Just like /etc/passwd file, this file also uses an individual line for each entry.
1. Username
2. Encrypted password
3. Number of days when password was last changed
4. Number of days before password can be changed
5. Number of days after password must be changed
6. Number of days before password expiry date to display the warning message
7. Number of days to disable the account after the password expiry
8. Number of days since the account is disabled
9. Reserved field

File permissions

Viewing Permissions from the Command-Line
File permissions may be viewed using **Is -l**
$ ls -1/bin/login
-rwxr-xr-x 1 root root 19080 Apr 1 18:26 /bin/login
Four symbols are used when displaying permissions:
• **r:** permission to read a file or list a directory's contents
• **w:** permission to write to a file or create and remove files from a directory
• **x:** permission to execute a program or change into a directory and do a long listing of the directory
• **-:** no permission (in place of the r, w, or x)

Changing File Ownership
• Only root can change a file's owner
• Only root or the owner can change a file's group
• Ownership is changed with **chown:**
  • **chown [-R] user_name file directory ...**
Examples:
  • **chown username:groupname file.txt**

- **chown -R username:groupname /path/to/directory**

• Group-Ownership is changed with **chgrp:**
- **chgrp [-R] group_name file directory**

Examples:
- **chgrp admins /path/to/directory**
- **chgrp -R developers /path/to/directory**

Changing Permissions - Symbolic Method
To change access modes:
- **chmod [-OPTION]... mode[, mode]** filel directory ...

mode includes:
- **u, g** or **o** for user, group and other
- **+ -** or for grant, deny or set
- **or, wor x** for read, write and execute

Options include:
- **-R** Recursive

• **-v** Verbose
• **--reference** Reference another file for its mode
Examples:
• **chmod ugo+r file**: Grant read access to all for file
• **chmod o-wx dir**: Deny write and execute to others for dir

Changing Permissions - Numeric Method
Uses a three-digit mode number
- first digit specifies owner's permissions
- second digit specifies group permissions
- third digit represents others' permissions

Permissions are calculated by adding:
- 4 (for read)
- 2 (for write)
- 1 (for execute)

Example:
- chmod 640 myfile

Change owner
chown sam.sam devopstools
chown sam.sam vpdir/ -R

Interview

https://chatgpt.com/share/6745db0c-8660-8002-a8fb-d2ae956853af

Linux
https://linuxjourney.com

Shell Scripting Tutorial Beginners To Advanced
https://safiakhatoon.hashnode.dev/shell-scripting-tutorial-beginners-to-advanced

https://learnshell.org/

Sonarqube dashboard compenents, like vern
Helm chat creation, templates, values default, custom values, commands, history, roll back, Secreats,

Argocd Application sets

Prometheus and Grafana are open-source tools that work together to monitor and visualize applications and systems. Prometheus collects metrics, while Grafana displays them.
Prometheus

- Collects and stores metrics as time-series data
- Includes a querying language
- Supports exporters for monitoring services like MySQL, Kubernetes, and Kafka
- Offers alerting

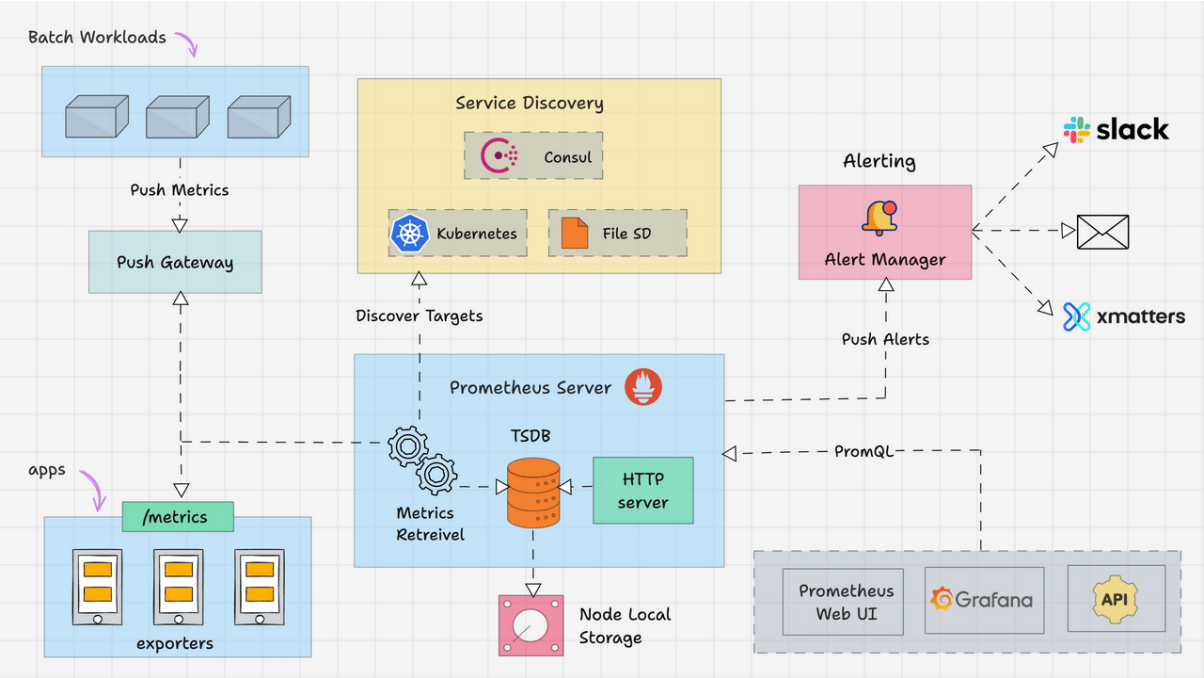
Grafana
It is a very powerful **visualisation tool** which can used for all sorts of dashboard and monitoring requirements.

- Transforms metrics into visualizations
- Works with multiple data sources, including Prometheus
- Allows users to create dashboards

Prometheus

Prometheus is a time series database. Prometheus is designed to monitor targets. Servers, databases, standalone virtual machines, pretty much everything can be monitored with Prometheus. It will collect the CPU, Disk and network utilization logs of pods/deployments/replicas. It will send these logs to Grafana.

Prometheus Architecture



Components

- **Prometheus Server**: The core component that collects metrics, stores them in a time series database (TSDB), and runs queries
- **Exporters**: Tools that expose metrics from third-party systems so Prometheus can collect them.
- **Alert manager**: Manages alerts by deduplicating, grouping, and routing them to notification channels
- **Pushgateway**: Allows ephemeral (short-lived) jobs to send metrics to Prometheus
- **Service Discovery**: Automatically discovers targets to collect metrics from.
- **Client Libraries**: "Allow application developers to expose custom metrics for Prometheus to collect.
-

How Prometheus works

Prometheus uses a pull-based approach to collect data from predefined HTTP endpoints. It applies rules to the data to create new time series or alert users.

Grafana:

Grafana is a web-based application that uses a microservices-based architecture to monitor and visualize data in DevOps. It's designed to help teams understand their systems and applications, and to troubleshoot issues.

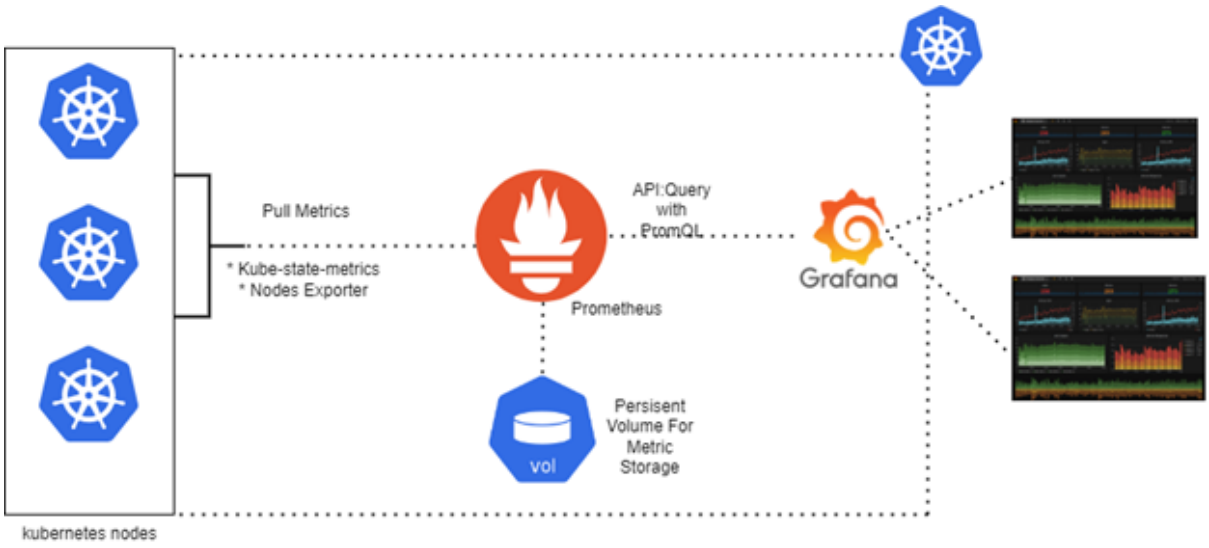
What is Grafana?

Grafana is a multi-platform open-source analytics and interactive visualization web application. It provides charts, graphs, and alerts when connected to supported data sources. It is expandable through a plug-in system. End users can create complex monitoring dashboards using interactive query builders.

It provides: ✓ Charts ✓ Graphs ✓ Alerts

Grafana Features: Visualize Grafana has a plethora of visualization options to help you understand your data, beautifully Alert Seamlessly define alerts where it makes sense | while you're in the data Unify Grafana supports dozens of databases, natively. Mix them together in the same Dashboard Open Source Grafana's completely open source, and backed by a vibrant community
Extend Discover hundreds of dashboards and plugins in the official library Collaborate Bring everyone together, and share data and dashboards across teams
[Create the dashboards in Grafana]
Alerting
How it works

- Grafana alerting periodically queries data sources and evaluates the condition defined in the alert rule
- If the condition is breached, an alert instance fires
- Firing instances are routed to notification policies based on matching labels
- Notifications are sent out to the contact points specified in the notification policy



Components

- **Grafana Mimir**: A microservices-based architecture that can run components in parallel
- **Backend**: Interacts with various data sources, such as Prometheus and Loki
- **Frontend**: Renders visualizations for users, such as charts and graphs

Features

- **Central monitoring**: Consolidates data from multiple sources into dashboards
- **Alerting**: Sends out alerts when metrics hit certain thresholds
- **Collaboration**: Sharing dashboards and data views enhances team communication
- **Integration**: Works with a range of DevOps tools

Use cases

- **Monitoring application performance**: Tracks key performance indicators (KPIs) and sets up alerts
- **Monitoring server performance**: Provides insights into CPU usage, memory consumption, and more
- **Monitoring application logs**: Consolidates and visualizes logs to identify issues
- **Request tracing**: Visualizes the end-to-end flow of requests through multiple services
- **Error tracking**: Correlates logs and metrics to identify and diagnose errors

Seamens Interview

Intro
Ansible configuration- why required as your are creating the infrastructure in Kubernetes (application deployed in K8S).
ArgoCD: Process how it works
K8s architecture and components
AWS resources - 20
Terraform code - vpc and eks code components (code)
Kubernates: Difference between statefull set and deployment.
How volumes will attach to