Sonarqube

→ Sonarqube is an open-source platform for continuous code quality management. It provides a range of static code analysis tools and metrics to access & improve the quality of code in software development projects.

→ Sonarqube analyzes source code, detects code smells, identifies bugs, and measures code complexity & test coverage.

* **Key features of Sonarqube include:**

① **Static Code Analysis:**

→ sonarqube performs static code analysis to identify coding issues, including code smells, bugs, vulnerabilities, and security vulnerabilities. It analyze code across multiple programming.

② **Code Quality Metrics:**

→ sonarqube measures and tracks code quality overtime. It calculates metrics such as code duplication, code coverage, cyclomatic complexity maintainability, and more.

→ These metrics help identify areas of improvement & track the progress of code quality initiatives.

③ **Issue Tracking:**

→ sonarqube identifies code issues and creats a comprehensive list of actionable findings.

→ It assigns severity levels to each issue & provides detailed explanations and recommendations for remediation.

→ Issue can be tracked, assigned, & managed with in the sonarqube interface.

④ Integration with CI/CD pipelines:

sonarqube can be integrated into the CI/CD pipeline to automatically analyze code quality during the development process. It can be used to fail builds and raise alerts if code quality threshold are not met ensuring that high quality code is delivered.

⑤ Quality Gates:

→ sonarqube allows the definition of quality gates, which are a set of predefined criteria that code must meet to pass quality checks.

→ Quality Gates can be config used to enforce specific standards and prevent code with critical issues from being merged into the codebase.

⑥ Language Support:

→ sonarqube supports a wide range of programming languages including Java/C/C++, C#, Javascript, python, Ruby, Typescript, and more

Versions:

Jenkins Version: 2.4.14

Terraform Version: 1.64

Vscode: 1.79.0

Ansible: 2.14.6

Docker: 23.0.6

kubernetes: 1.27

④ Integration with CI/CD pipelines:

Sonarqube can be integrated into the CI/CD pipeline to automatically analyze code quality during the development process. It can be used to fail builds (or) raise alerts if code quality threshold are not met ensuring that high quality code is delivered.

⑤ Quality Gates:

→ Sonarqube allows the definition of quality gates, which are a set of predefined criteria that code must meet to pass quality checks.

→ Quality Gates can be config used to enforce specific standards and prevent code with critical issues from being merged into the codebase.

⑥ Language Support:

→ Sonarqube supports a wide range of programming languages including Java/C/C++, C#, Javascript, Python, Ruby, Typescript, and more

ersions:

on: 2.4.14

Ansible

* **Need for Ansible:**

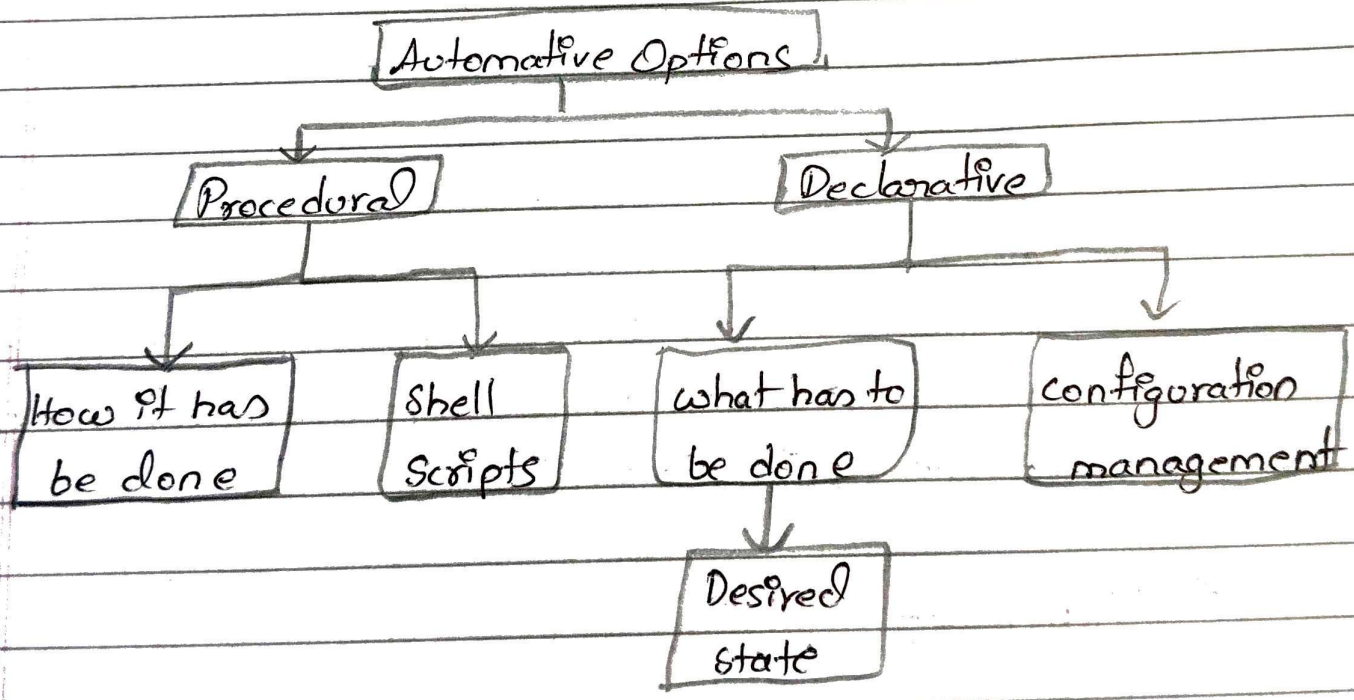
=> Problem statement

* Organization QA Policy
  - for every change submitted by developer run
    - unit tests
    - functional tests
    - performance tests

=> To solve this no. of deployments we have to automate.

To automate we have two ways

1. Procedural
2. Declarative



Configuration management can be acheived by many tools. some of them are:

1) puppet
2) chef
3) Ansible
4) Salt
5) Powershell DSC

* Architecture of Configuration Management (CM):

Components of configuration management architecture

1) **CM server:**
   → This has necessary tools to automate the deployment.
   → This seerver understands the desire state.
2) **Nodes:**
   → These are the seervers on which your application run
   → This is where as we are supposed to deploy the application

* Types of CM:

① Pull Based CM
② Push Based CM

=> Pull based CM: Nodes need to know about server which requires agent to be installed on nodes.

Ex: chef, puppet

=> Push Based CM:

- CM servers should be aware of the nodes.
- Servers logs into the node & executes the work.
  - → credentials & permissions are required

Ex: Ansible, Salt

* Ansible:

=> Ansible is an open source s/w developed in python. Ansible expects python to be installed on nodes.

=> What needs to provided to Ansible

① **Desired State:**
It is written in YAML and it is called as "Playbook"

② **List of Nodes:**
The nodes where the playbook has to be executed, this is referred as "inventory"

③ **Credentials:**
Credentials can be passed as part of playbook (or) inventory / command line

* 1. Setup of Ansible:

Ansible control node

→ This component takes the inputs mentioned above to execute playbooks on the nodes

→ This node is a linux VM / physical machine and we install ansible

→ Credentials: username
password
key Based Approach

* Ansible deals with credentials:

i) possible credentials

→ username and password

→ username and key

ii) Ansible when executing playbook logins to the node using credentials provided and takes help of the python installed on the node to get the job done.

iii) Login into the node

→ SSH (This is used for linux & MAC)

→ Winrm (This is used for windows)

* Setup of Ansible:

Ansible contro node

→ This component takes the inputs mentioned above to execute playbooks on the nodes

→ This node is a linux VM / physical machine and we install ansible

→ Credentials : username
password
key Based Approach

* Ansible deals with credentials:

i) possible credentials
→ username and password
→ username and key

ii) Ansible when executing playbook logins to the node using credentials provided and takes help of the python installed on the node to get the job done.

iii) Login into the node
→ SSH (This is used for linux & MAC)
→ Winrm (This is used for windocos)

* Username and password Authentication:

Linux machines allow us to login using ssh protocol and configurations of ssh are present in

```
sudo nano /etc/ssh/sshd-config
# change password authentication to 'YES'
sudo systemctl restart sshd
sudo visudo
jenkins ALL=(ALL:ALL) NOPASSWD:ALL
```

* Ansible setup with key Based Authentication b/w ACN & Nodes

1) create two linux VM's
2) Create a user on both VMs with password
3) Ensure the user has such permissions
4) Now create an ssh key pair on Ansible control node

```
ssh-keygen
ssh-copy-id <username>@<public_ip>
```

* Ansible playbook:

-> Ansible playbook is a powerful automation tool used for managing and configuring computer systems.
-> It is a declarative language that allows you to describe the desired state of your systems & define the tasks needed to achieve that state.
-> Ansible playbooks are written in YAML (Yet Another Markup language) format, which makes them human readable & easy to understand.

* Tasks:

-> Tasks are the fundamental units of work that define the actions to be performed on remote systems.
-> Each task in an ansible playbook describes a specific action or operation that should be executed.

* Module:

-> A module is a standalone unit of code that performs a specific action on a remote system.
-> Modules responsible for carrying out tasks in ansible playbooks and provide a way to interact with the managed hosts.

- apt
- yum
- file
- user
- group
- copy
- systemd
- get-url
- unarchive
- stat
- package
- template
- debug
- set_fact
- shell
- command
- apt-repository
- yum_repository
- apt_add_key
- setup
- git

* **Variables :**

→ In ansible, variables are used to store and manipulate data within playbooks (or) role.

→ They provide way to dynamically control the behaviour of tasks and templates.

→ Variables can be defined at various levels, such a

i) Group Variables

ii) Hosts Variables (or)

iii) Task-level variables

* **Handlers:**

→ Handlers are tasks that are triggered by other tasks, typically when there is a change in the system state.

→ They are used to respond to specific events and perform actions such as restarting a service or reloading a configuration file

→ Mostly used for service:

Start, restart, enable, stop, daemon-reload

* **Loops:-**

→ loops are used to repeat a set of tasks multiple times, iterating over a list or dictionary of items.

→ They provide a way to perform repetitive operations and apply the same tasks to multiple items without duplicating code.

* **Inventory:**

→ Inventory is a file (or) collection of files that defines the hosts and group of hosts that ansible can manage.

→ It provides a way to organize and categorize the target systems on which ansible can execute tasks and playbooks.

→ The inventory file is typically a text file written in a INI-style format (or) YAML format.

* Ansible facts :-

→ Ansible can collect information about the node where the playbook is under execution using facts.

→ To collect facts ansible uses setup module

```
ansible -i hosts -m setup -a "filter=*distribution*" all
```

* Adhoc Command :

→ Adhoc command is a way of running ansible module by constructing command line using /usr/bin/ansible

→ We use adhoc commands for activities which do not require automation

```
ansible -i hosts -m setup -a "filter=distribution all"
```

* Conditionals :

→ In a playbook, you may want to execute different tasks, or have different goals depending on the value of a fact (data about the remote system), a variable, or the result of a previous task, this is done by using conditions.

```
when: ansible_facts[distribution] == "ubuntu"
```

* Ansible templates :

→ Ansible uses jinja2 for templating. In ansible to use the variables. we use expression {{ variable-name }} which is derived from templates language jinja2

→ we can use jinja2 in files so that the content can be dynamic

* Ansible Roles:-

→ Ansible roles help in creating re usable ansible assets.

→ Roles let you automatically load related vars, files, tasks, handlers, and other artifacts based on a known file structure. After your group your content in roles, you easily reuse them and share them with other users.

→ By default, Ansible looks for roles in the following locations:

1) In collections, if you are using them
2) In a directory called roles/, relative to the playbook file
3) In the configured roles_path. The default search path is ~/.ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles
4) In the directory where the playbook file is located.

* Ansible Collections:

→ In ansible the reusable assets are roles, modules.

→ In ansible we can create custom modules as well by writing python code.

→ Ansible collections are collection of custom ansible modules and roles.

* Ansible Special Variables:

These variables cannot to set directly by the user; Ansible will always override them to reflect internal state.

Ansible Roles:-

Ansible roles help in creating re-usable ansible assets. Roles let you automatically load related vars, files, tasks, handlers, and other artifacts based on a known file structure. After your group your content in roles, you easily reuse them and share them with other users.

By default, Ansible looks for roles in the following locations:

1) In collections, if you are using them
2) In a directory called roles/, relative to the playbook file
3) In the configured roles_path. The default searchpath is ~/.ansible/roles:/usr/share/ansible/roles:/etc/ansible/roles

* **Ansible Vault:**

① create a vault-encrypted file: use the 'ansible-vault create' command to create a new encrypted file or convert an existing file into an encrypted file.
Ex: to create a new encrypted file named 'secrets.yml', you can run the following command

```
ansible-vault create secrets.yml
```

② Edit the encrypted file: Once you have created the encrypted file, you can edit its contents using the 'ansible-vault edit' command.

```
ansible-vault edit secrets.yml
```

③ Include the encrypted file in playbook: In your playbook, you can include the encrypted file using the 'vars_files' directive.

④ Running the playbook with vault: when running the playbook, you need to provide the vault password to decrypt the encrypted file. you can do this using the '--ask-vault-pass' option.

```
ansible-playbook -playbook.yml --ask-vault-pass
```

* Ansible Vault:

① create a vault-encrypted file: use the 'ansible vault create' command to create a new encrypted file or convert an existing file into an encrypted file.
Ex: to create a new encrypted file named 'secrets.yml', you can run the following command

```
ansible-vault create secrets.yml
```

② Edit the encrypted file: Once you have created the encrypted file, you can edit its contents using the 'ansible-vault edit' command.

```
ansible-vault edit secrets.yml
```

③ Include the encrypted file in playbook: In your playbook, you can include the encrypted file using the 'vars_files' directive.

④ Running the playbook with vault: when running the playbook, you need to provide the vault password to decrypt the encrypted file. you can do this using the '--ask-vault-pass' option.

```
ansible-playbook playbook.yml --ask-vault-pass
```

Docker

* Microservices:
This is all about breaking a monolith app to multiple smaller services.

* Generations of running applications:
① Run directly on physical servers
(cost is high and also one server, one application)
② Hypervisor
(creating multiple VM in physical server, and also hypervisor licensing costs are involved
③ Container
(created by container engine(docker). It has lean OS and application runs inside the container & smaller changes, replacing services in easier(new versions))

* Docker:
-> A platform for developing, shipping, and running application in light weight, portable containers.
-> Docker is an open-source platform that enables developers to automate the deployment, scaling, & management of application using containerization.
-> It provides an easy and efficient way to package applications along with their dependencies and configurations, into lightweight, portable & isolated containers.

Docker

docker system -a prune
(It remove all related to docker)

* Microservices :

This is all about breaking a monolith app to multiple smaller services.

* Generations of running application :

① Run directly on physical servers
(cost is high and also one server, one application)

② Hypervisor
(creating multiple VM in physical server, and also hypervisor licensing costs are involved

③ Container
(created by container engine(docker). It has lean OS and application runs inside the container & smaller changes, replacing services in easier(new versions))

* Containers:

→ A light weight, standalone, and executable s/w package that includes everything needed to run a piece of s/w includes the code, runtime, libraries, and system tools.

→ Containers are lightweight and standalone units that encapsulate an application & its dependencies, including libraries, binaries, and configuration files

* Docker Image:

→ A docker image is a read-only template used to create containers. It contains the application code, runtime libraries, and any other dependensices required to run the application.

* Dockerfile:

→ A dockerfile is a text file that contains a set of instructions used to build a Docker image.

→ It provides a set of declarative syntax for specifing the steps required to create an image, including defining the base image, adding files, installing dependencies, & configuring the runtime environment.

* **Image layer:**

→ A image layer refers to a single immutable layer within a docker image.

→ Docker images are built using a layered filesystem, where each layer represents a specific set of changes or instructions applies to the base images.

→ These layers are stacked on top of each other, with each layer modifying (or) adding to previous layer.

* **Container options:-**

→ We can perform the following options (Container life cycle)

1) Create Container
2) Delete Container
3) Start Container
4) Stop Container
5) Pause Container
6) unpause container.

→ Every container, when created gets a unique

i) Containerid

ii) Container name

→ **docker Stats** ⇒ To know the CPU/RAM utilization

→ **docker system -a prune** ⇒ It remove all related to Docker.

* Docker Volume:

→ Docker volumes are a feature in docker that allow for persistent data storage and sharing between containers and the host machine.

→ Volume provides a away to store a manage data separately from the container lifecycle, ensuring that the data persists even if the container is stopped, deleted or replaced.

① Bind mount:

In bind mount is a type of volume that allow you to directly mount a specific directory or file from the host machine into container.

Syntax: '-/host/path:/container/path'

② tmpfs mount:

A tmpfs mount is a type of volume that is stored in memory instead of on the host machine file system. It allows you to create temporary file system within a container that is stored in memory and does not persist across container restarts (or) on the host machine.

③ Volumes:

These are storage spaces generally from docker host which is managed by docker. It has a sub-command 'dockervolume' Stored in host file system ged by docker (/var/lib/docker/volumes/onlinux)

* **Docker Volume:**

→ Docker volumes are a feature in docker that allow for persistent data storage and sharing between containers and the host machine.

→ Volume provides a away to store a manage data separately from the container lifecycle, ensuring that the data persists even if the container is stopped, deleted or replaced.

① **Bind mount:**

In bind mount is a type of volume that allow you to directly mount a specific directory or file from the host machine into container

Syntax : `'-/host/path :/container/path'`

② **tmpfs mount:**

A tmpfs mount is a type of volume that is stored in memory instead of on the host machine file system. It allows you to create temporary file system within a container that is stored in memory and doesnot persist across container restarts (or) on the host machine.

③ **Volumes:**

These are storage spaces generally from docker host which is managed by docker. It has a sub-command "dockervolume" stored in host file system managed by docker `/var/lib/docker/volumes/` on linux

* Docker Networking :-

→ It refers to the networking capabilities and features provided by docker to enable communication and connectivity b/w containers, as well as b/w containers and the outside world.

→ Docker provides various networking options that allow containers to securely communicate with each other & with other network resources.

① Default Networking :

→ By default, docker creates a virtual network called the "Bridge" network.

→ containers that are started without explicitly specifying a network will be connected to the bridge network.

→ container within the same bridge network can communicate with each other using IP address.

② User-defined Networks :

→ Docker allow you to create user-defined networks to isolate containers & control their communication.

→ It provide a secure and isolated environment for containers.

→ You can create networks with their own IP address & connectivity rules.

i) **Bridge**:
The default network driver that allows containers to communicate on the same host

ii) **Overlay**:
It Enables communication b/w containers across multiple hosts in a swarm cluster

iii) **Host**:
Connects containers directly to the host's network Starts by passing network isolation.

iv) **Macvlan**:
Assign a MAC address to each container, making them appear as separate physical devices on the n/w.

* **Container-to-Container Communication**:

Docker provides DNS-based service discovery for container-to-container communication within the same network.

Containers can communicate with each other using container names as DNS names.

* **Port Mapping**:

Docker allows you to expose container ports to the host machine (or) bind them to specific IP address. This enables accessing containerized services from outside the docker environment.

i) **Bridge**:
The default network driver that allows containers to communicate on the same host

ii) **Overlay**:
It Enables communication b/w containers across multiple hosts in a swarm cluster.

iii) **Host**:
Connects containers directly to the host's network
Starts by passing network isolation.

iv) **Macvlan**:
Assign a MAC address to each container, making them appear as separate physical devices on the n/w.

* **Container-to-Container Communication**:

Docker provides DNS-based service discovery for container-to-container communication within the same network.

Containers can communicate with each other using container names as DNS names.

* **Port Mapping**:

Docker allows you to expose container ports to the host machine (or) bind them to specific IP address. This enables accessing containerized services from outside the Docker environment.

* Network Security:

→ Docker networking includes features for securing container communication, such as container-to-container authentication & encryption using TLS (Transport Layer Security)

* Integration with External Networks:

→ Docker supports integration with external networks such as the host network (or) existing network infrastructure allowing containers to communicate with resources outside the docker environment.

* **Docker Swarm:**

→ Docker swarm is a cluster management & orchestration features are embedded inside docker engine.

→ Docker swarm consists of multiple docker hosts which run in swarm mode.

→ It has multiple docker hosts.
Two roles "Managers" & "Workers" exist in docker swarm.

- **Manager** is responsible for membership & delegation.
- **Worker** is responsible for running swarm services.

→ Each docker host can be a manager, a worker or both.

→ In docker swarm desired state is mentioned.

→ For instance if you are running one container in swarm on a particular node(worker) & that node goes down, then swarm schedules this nodes task on other node to maintain the state.

→ Task is a running container which is part of swarm service managed by swarm manager

Nodes:

It is instance of docker engine participating in swarm. There are two kinds of nodes:

① Manager Nodes:

you communicate to manager node to deploy applications in the form of service definitions. Manager nodes dispatch of work called as tasks to the worker node.

② Worker nodes:

→ They receive & execute the tasks dispatched from manager nodes. An agent runs on the worker node & reports on the tasks assigned to it.

Services:

→ Service is the definition of the task to be executed.
→ Typically it would be the application to be deployed.
→ Two kinds of service models are available:
  ① Replicated Services Model
  ② Global Services Model

Task:

→ Carries a docker container and the commands to run inside the container.
→ It is the atomic scheduling unit of swarm. Once task is assigned to node, it cannot move to another node.
→ It can only run on the assigned node or fail.

Docker Swarm Setup:

Install Docker on all machines.

* Rolling Updates:

-> At rollout time you can apply service updates to nodes incrementally.

-> The swarm manager lets you control the delay b/w service deployment to different sets of nodes.

-> If anything goes wrong, you can rollback to a previous version of the service.

* Scaling:

-> For each service, you can declare the no. of tasks you want to run.

-> When you scale up or down, The swarm manager automatically adapts by adding or removing tasks to maintain the desired state.

* Swarm mode CLI commands:

1. Swarm init
2. Swarm join
3. Service create
4. Service inspect
5. Service ls
6. Service rm
7. Service scale
8. Service ps
9. Service update

* **Rolling Updates:**

→ At rollout time you can apply service updates to nodes incrementally.

→ The swarm manager lets you control the delay b/w service deployment to different sets of nodes.

→ If anything goes wrong, you can rollback to a previous version of the service.

* **Scaling:-**

→ For each service, you can declare the no. of tasks you want to run.

→ When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.

* **Swarm mode CLI commands:**

1. Swarm init
2. Swarm join
3. Service create
4. Service inspect
5. Service ls
6. Service rm
7. Service scale
8. Service ps
9. Service update

* Docker Compose :-

→ Docker compose is a tool that allows you to define and manage multi-container docker applications.

→ It provides a simple way to define the services, networks and volumes required for an application using a declarative YAML file.

→ With Docker Compose, you can describe the components of your application stack & their configuration in a single file.

→ compose works in all environment : production, staging, development, testing, as well as CI workflows.

① Services :

A service is a containerized component of ur app It can be a web server, database, cache, or any other service, that makes up your application stack.

② Networking :

Docker compose automatically creates a n/w for your application stack, allowing services to communicate with each other using their service names as hostnames.

③ Volumes :

Docker compose allows you to define volumes to persist data generated by your services. Volumes can be mounted from the host machine.

④ Environment Variables:

Docker compose supports specifying environment variables for your services.

* Docker Registries:

→ Registry is used to store the docker images

→ Docker images are OCI (Open Container Initiative) complaint.

→ There are two types of registries:

① **public registry**:- Open to all for usage

Ex: DockerHub

② **Private Registry**:- Registry for internal usage within Organisation

Ex: DockerHub (Private)

Azure Container Registry (ACR)

AWS Elastic Container Registry (ECR)

JFrog / Artifactory

Kubernetes (K8s)	Docker Swarm
① Scalability (K8s is designed to handle large-scale deployments & can scale thousands of nodes & containers)	① Scalability (Docker swarm can also scale application horizontally, but it generally consider more suitable for smaller scale deployments)
② Networking & Service discovery (K8s provides a more advanced networking model with its built-in n/w'ing solutions called "kube-proxy". It offers features like service discovery, load balancing, & n/w policies)	② Docker swarm uses a overlay n/w for inter-node communication and service discovery. It offers built in DNS based service discovery & basic load balancing)

Imp:

* docker stats (To know the CPU/RAM Utilization)
* list all the process "ps" (or) "ps aux"
* get the ip address "ipaddr" (or) "ifconfig"
* Explore storage "df -h" & "lsblk"
* Find Username & hostname "whoami" & "hostname"

Kubernetes (K8s)	Docker Swarm
① Scalability (K8s is designed to handle large-scale deployments & can scale thousands of nodes & containers)	① Scalability (Docker swarm can also scale application horizontally, but it generally consider more suitable for smaller scale deployments)
② Networking & Service discovery (K8s provides a more advanced networking model with its built-in n/w'ing solutions called "kube-proxy". It offers features like service discovery, load balancing, & n/w policies)	② Docker swarm uses a overlay n/w for inter-node communication and service discovery. It offers built in DNS based service discovery & basic load balancing)

Imp:

* docker stats (To know the CPU/RAM Utilization)
* list all the process "ps" (or) "ps aux"
* get the ip address "ipaddr" (or) "ifconfig"
* Explore storage "df -h" & "lsblk"
* Find Username & hostname "whoami" & "hostname"