

Operating System Concepts

- OS Concepts
- Linux commands
- Shell scripts
- Linux System call Programming

Learning OS

- step 1: End user
 - Linux commands
- step 2: Administrator
 - Install OS (Linux)
 - Configuration - Users, Networking, Storage, ...
 - Shell scripts
- step 3: Programmer
 - Linux System call programming
- step 4: Designer/Internals
 - UNIX & Linux internals

What is OS?

- Interface between end user and computer hardware.
- Interface between Programs and computer hardware.
- Control program that controls execution of all other programs.
- Resource manager/allocator that manage all hardware resources.
- Bootable CD/DVD = Core OS + Applications + Utilities
- Core OS = Kernel -- Performs all basic functions of OS.

OS Functions

- CPU scheduling
- Process Management
- Memory Management
- File & IO Management
- Hardware abstraction
- User interfacing
- Security & Protection
- Networking

Process Management

Program

- Set of instructions given to the computer --> Executable file.
- Program --> Sectioned binary --> "objdump" & "readelf".

- Exe header --> Magic number, Address of entry-point function, Information about all sections. (objdump -h program.out)
- Text --> Machine level code (objdump -S program.out)
- Data --> Global and Static variables (Initialized)
- BSS --> Global and Static variables (Uninitialized)
- RoData --> String constants
- Symbol Table --> Information about the symbols (Name, Size, section, Flags, Address) (objdump -t program.out)
- Program (Executable File) Format
 - Windows -- PE
 - Linux -- ELF
- Program are stored on disk (storage).

Process

- Program under execution
- Process execute in RAM.
- Process control block contains information about the process (required for the execution of process).
 - Process id
 - Exit status
 - 0 - Indicate successful execution
 - Non-zero - Indicate failure
 - Scheduling information (State, Priority, Sched algorithm, Time, ...)
 - Memory information (Base & Limit, Segment table, or Page table)
 - File information (Open files, Current directory, ...)
 - IPC information (Signals, ...)
 - Execution context (Values of CPU registers)
 - Kernel stack
- PCB is also called as process descriptor (PD), uarea (UNIX), or task_struct (Linux).
- In Linux size of task_struct is approx 4KB

Process

- Process is program in execution.
- Process has multiple sections i.e. text, data, rodata, heap, stack. ... into user space and its metadata is stored into kernel space in form of PCB struct.
- PCB contains
 - id, exit status,
 - scheduling info (state, priority, time left, scheduling policy, ...),
 - files info (current directory, root directory, open file descriptor table, ...),
 - memory information (base & limit, segment table, or page table),
 - ipc information (signals, ...),
 - execution context, kernel stack, ...

File Management

File

- File is collection of data/information on storage device.
 - File = Contents (Data) + Information (Metadata)
 - The data is stored in zero or more Data blocks (in FS), while metadata is stored in the FCB (in filesystem).
- FCB is called as "inode" on UNIX/Linux. It contains
 - type: UNIX/Linux has 7 types of files
 - -: regular, d: directory, l: symbolic link, p: pipe, s: socket, c: char device, b: block device
 - size: number of bytes
 - links: number of hard links
 - mode (permissions): (u) rwx, (g) rwx, (o) rwx
 - user & group
 - time-stamps: modification, creation, access.
 - info about data blocks
- terminal> ls -l
 - type, mode, links, user, group, size, timestamp, name.
- terminal> stat filepath

File System

- Files are stored on storage device. Arrangement of files in storage device is called as "File System".
- e.g. FAT, NTFS, EXT2/3/4, ReiserFS, XFS, HFS, etc.
- File System logically divide partition into 4 sections.
 - Boot block/Boot sector
 - Contains programs/info required for booting of OS
 - Typically contains bootstrap program and bootloader program
 - Super block/Volume control block
 - Contains information of whole partition.
 - Capacity, Label.
 - terminal> df -h
 - Total number of data blocks/inodes.
 - Number of used/free data blocks/inodes.
 - Information of free data blocks/inodes.
 - Inode List/Master file table
 - Inodes (FCB) for each file
 - Data blocks
 - Stores data of the file.
 - Each file have zero or more data blocks.
 - Size of data blocks can be configured while creating file system
- File system is created by the format utility while formatting the partition.
 - Windows: format.exe
 - Linux: mkfs
 - terminal> sudo mkfs -t ext3 /dev/sdb1
 - terminal> sudo mkfs -t vfat /dev/sdb1
 - -t fs_type e.g. ext3, ext4, vfat, ntfs, ...

- partition e.g. /dev/sdb1
- Disk/partition naming conventions
 - Windows:
 - Disks are named as disk0, disk1, ...
 - partitions are named as drives i.e. C:, D:, E:, ...
 - Linux:
 - Disks are named as /dev/sda, /dev/sdb, /dev/sdc, etc.
 - Partitions per disk are named as
 - sda partitions: sda1, sda2, sda3, ...
 - sdb partitions: sdb1, ...

Linux File Structure

- Linux follows "/" (root) file system.
- "/" is a starting point of Linux file system.
- All your data is stored in this partition.
- / contains boot, bin, sbin, etc, root, home, dev, proc, mnt, media, opt
- In Linux everything is a file.
- Mainly there are two types of files in Linux
 - File
 - Directory (Folder)
- Linux Directories
 - boot - files related to booting
 - vmlinuz - kernel Image
 - grub - boot loader
 - config - kernel configuration
 - initrd/initramfs - initial root file system
 - bin - user commands in binary format
 - sbin - all admin/system commands in binary format
 - etc - configuration files
 - root - home directory of root user
 - home - it contains sub directories for each user with its name
 - devendra -> /home/devendra
 - sunbeam -> /home/sunbeam
 - osboxes -> /home/osboxes
 - dev - it contains all device related files
 - lib - shared program libraries required by kernel
 - mnt - it is temporary mount point
 - media - it is mount point for media eg cdrom
 - opt - stores optional files of large softwares
 - proc - virtual file system - it contains information about system or processes
 - sys - entries of each block devices, subdirectories for each physical bus type supported, every device class registered with the kernel, global device hierarchy of all devices
 - tmp - temporary files that may be lost on system shutdown
 - usr - read only directory that stores small programs and files accessible to all users

User interfacing

- UI of OS is a program (Shell) that interface between End user and Kernel.
- Shell -- Command interpreter
 - End user --> Command --> Shell --> Kernel
- User interfacing (Shell)
 - Graphical User Interface (GUI)
 - Command Line Interface (CLI)

Example shells

- Windows
 - GUI shell: explorer.exe
 - CLI shell: cmd.exe, powershell.exe
- DOS
 - CLI shell: command.com
- Unix/Linux
 - CLI shell: bsh, "bash", ksh, csh, zsh, ...
 - ls /bin/*sh
 - echo \$SHELL
 - shell of current user can be changed using "chsh" command.
- GUI shell/standards
 - GNOME: GNU Network Object Model Environment (e.g. Ubuntu, Redhat, CentOS, ...)
 - KDE: Kommon Desktop Environment (e.g. Kubuntu, SuSE, ...)

Path

- It is a unique location of any file in the file system.
- It is represented by character strings with few delimiters ("/", "\", ":")
- Types of path
 - There are two types of paths in linux
 - Absolute path
 - Path which starts with "/" is called as absolute path.
 - E.g. /home/devendra/MyData/Demos/demo01.sh
 - Relative path
 - Path with respect to current directory is called as relative path
 - E.g. MyData/Assignments/assign02.pdf

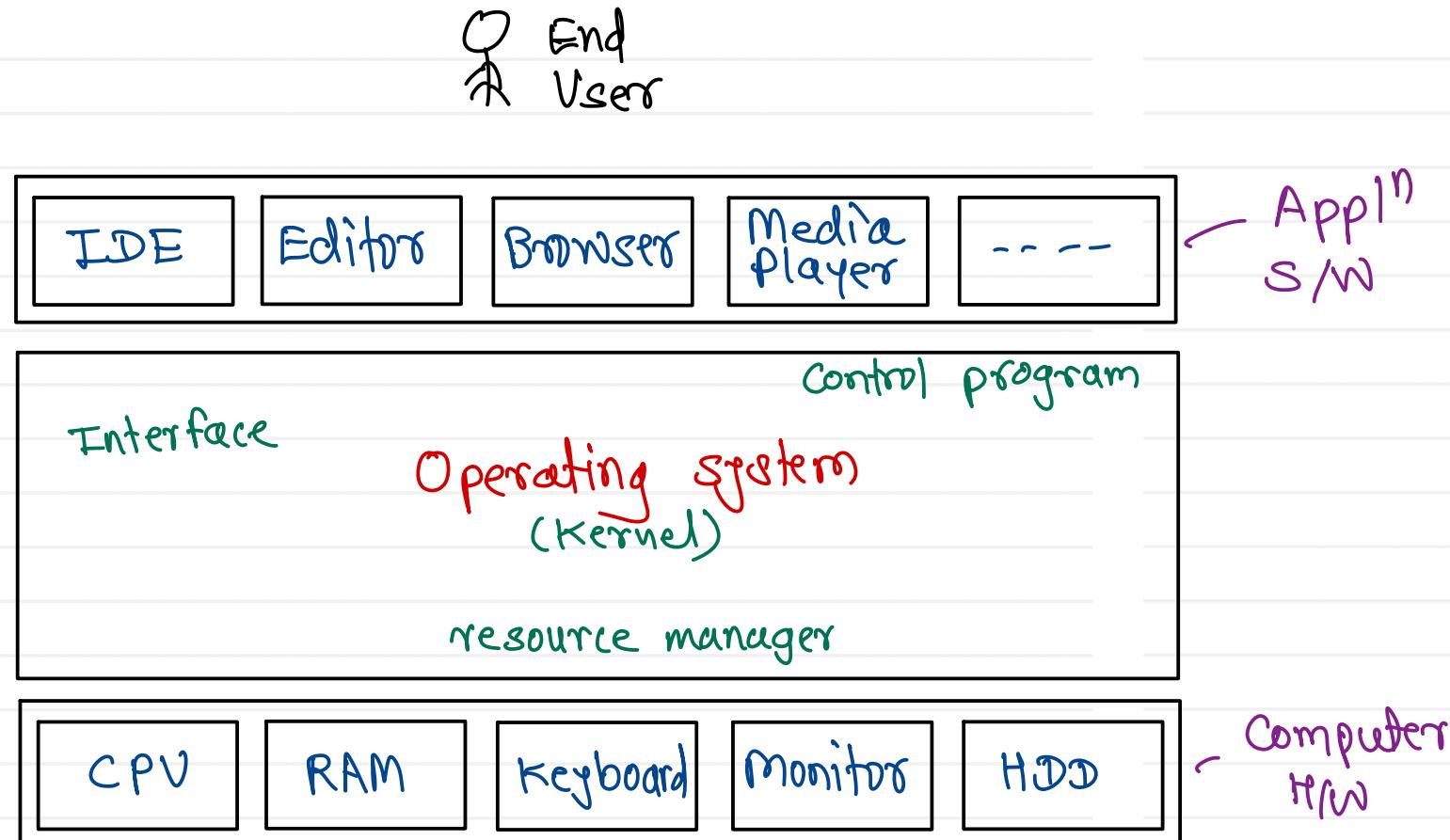


**Sunbeam Institute of Information Technology
Pune and Karad**

Module - Concepts of Operating System

Trainer - Devendra Dhande
Email – devendra.dhande@sunbeaminfo.com

Operating system



- interface between end user & hardware
 - interface between application softwares & hardware
 - control program which controls execution of all programs running in system
 - resource allocator/ mgr which manages limited hardware resources
- CD/DVD/ISO - Core OS + Application Software + System Utilities (Kernel)



Linux kernel architecture

Program1 Program2 Program3

System call API

System calls

Process Management

CPU scheduling

Memory Management

File & IO Management

Device Drivers

Hardware Abstraction

CPU

RAM

Keyboard

Monitor

HDD

Kernel

- Networking
- User Interfacing
- Security & protection

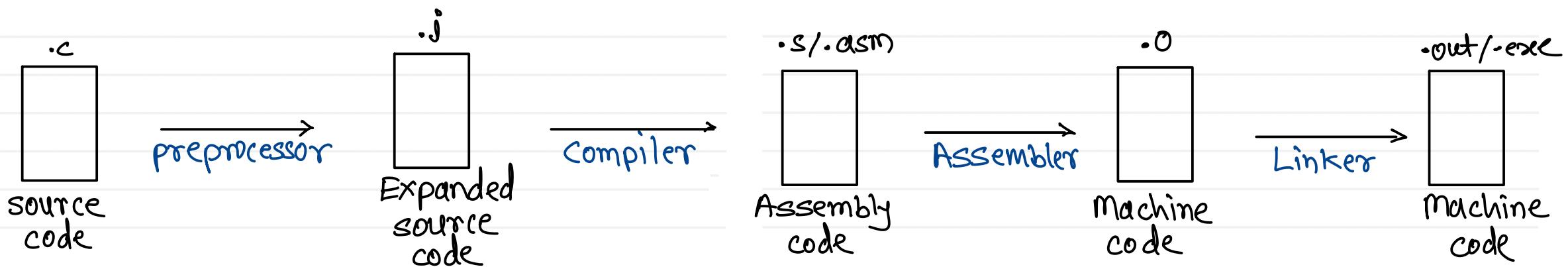
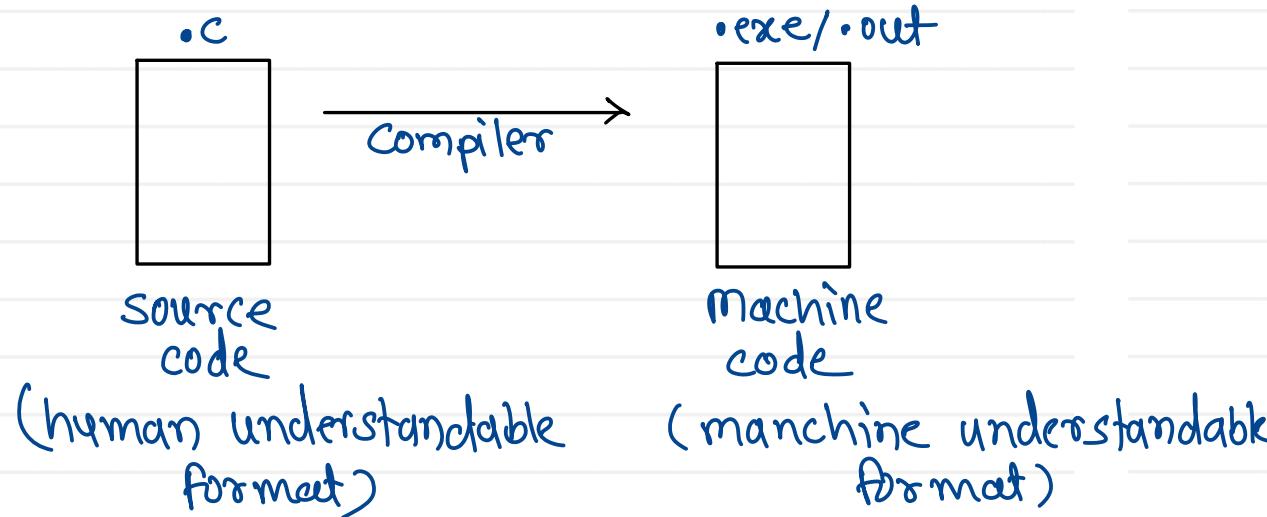
↑
optional
functions



Program compilation steps

Process : Program in execution

Program : set of instructions to machine (CPU)



Toolchain : (gcc)

- set of tools which work on source code one by one to convert it into machine code.

1. Preprocessor (CPP)
2. Compiler (cc)
3. Assembler (as)
4. Linker (ld)
5. Debugger (gdb)
6. Utilities (make, objdump)



Program

Program

• .exe / .out

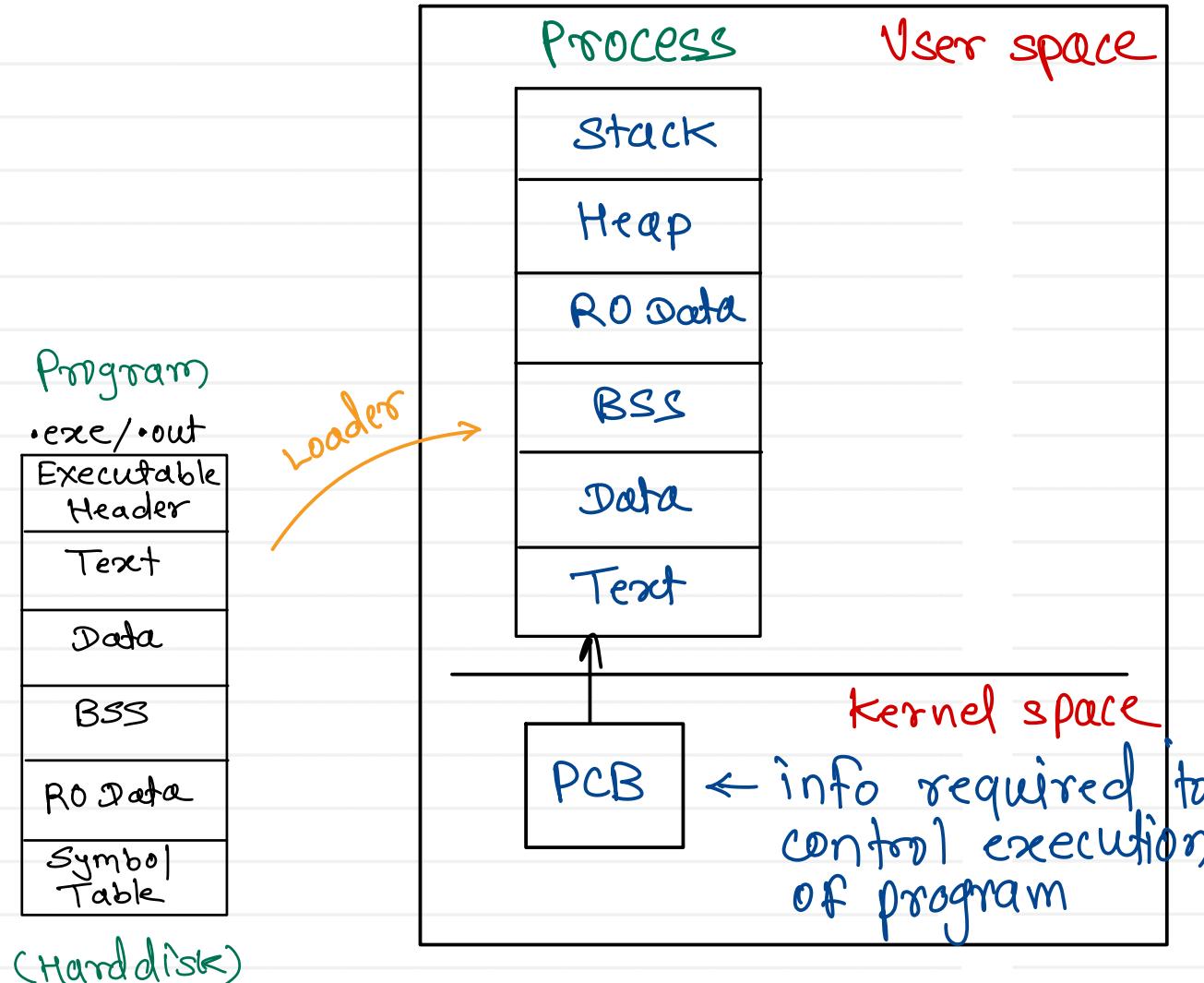
Executable Header
Text
Data
BSS
RO Data
Symbol table

(sectioned binary)

- Magic number (2 or 4 bytes)
 - identity to file format
- Windows - .exe - Portable Executable - MZ
- Linux - .out - Executable Linking Format - ELF
- info about program
- type of program (CLI / GUI / library)
- info about remaining sections of program (start, end, size)
- address of entry point function
- instructions of program in machine code format
- initialized static & global variables `int num1 = 10;`
- uninitialized static & global variables `int num2;`
- read only data (string constants)
- info about symbols
 - variables - name, type, add^r, section, size
 - symbols
 - functions - return type, name, add^r, no type args



Process



Linux :

PCB - Process descriptor
struct task_struct
(sched.h)

1. pid , ppid
2. exit status
3. mem info (base, limit, segment/page table)
4. sched info (algo, prio, state ...)
5. file info (opened files)
6. IPC info (signals)
7. execution context
8. kernel stack

File and File system

File : collection of data / information

File : data + metadata

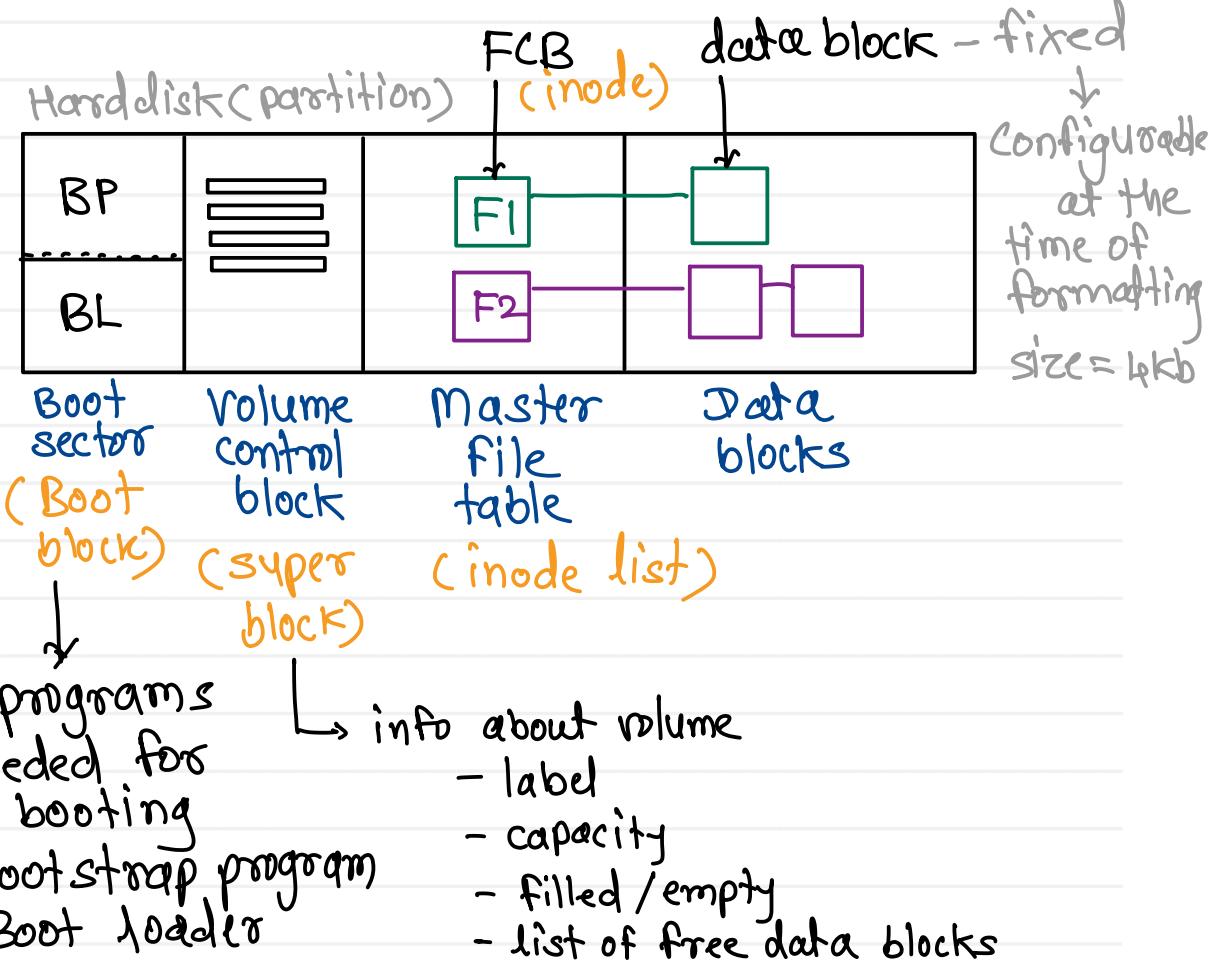
↓
(actual content) (info about file)

data block

File Control block

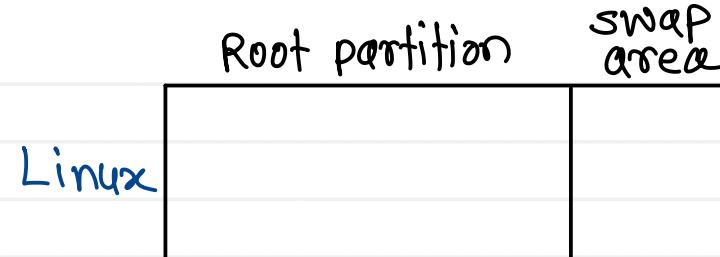
- name , path (FCB)
- size
- type
- user/owner , group
- timestamps
 - create, access , modify
- permissions
 - read, write, execute
 - user/ group / others
- link count
- info about data blocks

File system : organizing files on partitions



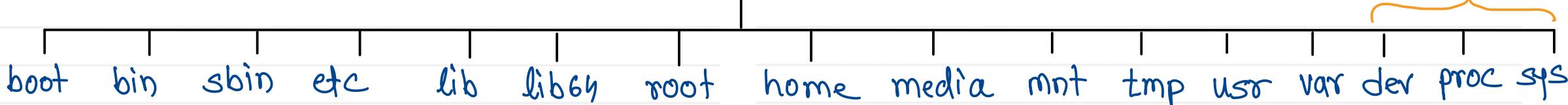


Linux file structure



- Linux follows root file structure
- 'root' is denoted by '/'

↑
directory



Absolute path:

- always starts with '/'

/home/sunbeam/Java/Assignments/assign01.pdf

Relative path:

- always starts w.r.t present working directory

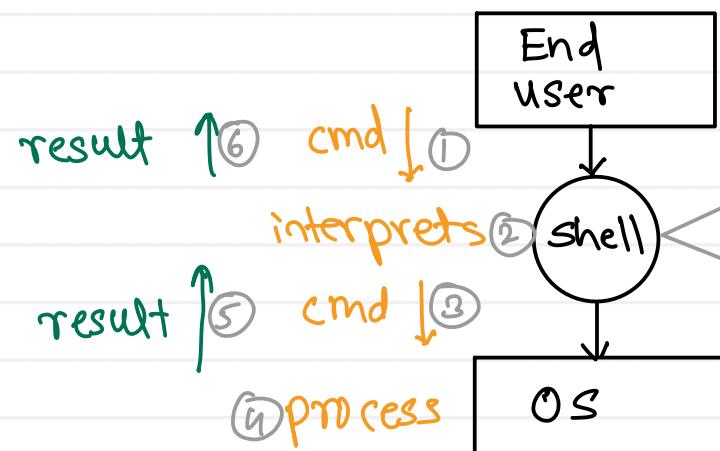
Java/Assignments/assign01.pdf

file → file
folder → directory
Administrator → super user
admin → root
pseudo file systems



User Interfacing

- user interfacing is achieved through one program which is called as "shell".
- Shell is an intermediate between end user & operating system.
- shell is a command interpreter



- in Ubuntu default shell is 'bash'

BASH - Bourne Again Shell

echo \$SHELL - will show default shell

chsh - to change default shell

CLI based shell (Command Line Interface)

Windows - cmd.exe, powershell.exe

Linux - bsh(sh), bash, csh, ksh, zsh

GUI based shell (Graphical User Interface)

Windows - explorer.exe

Linux - KDE (Kommand Desktop Env)

- GNOME

(GNU Network Object Model Env)



Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

Linux Admin

- In Linux, Admin is called as "super-user".
- Admin's login name is "root".
- Most of modern Linux, disable "root" login (for security).
- To execute commands with admin privileges use "sudo" (if approved by system admin).
 - cmd> sudo apt update
 - cmd> sudo apt install vim gcc python3 python3-pip
 - cmd> sudo snap install --classic code

Directory commands

- pwd -- print present working directory (current directory)
- cd -- change directory (syntax> cd dirpath)
- ls - list directory contents (syntax> ls dirpath)
- mkdir -- make directory (syntax> mkdir dirpath)
- rmdir -- remove empty directory (syntax> rmdir dirpath)
- cd
 - cd ~ - change working directory to home directory
 - cd - - change working directory to old working directory
 - cd .. - change working directory to parent directory

File commands

- cat
 - cat > filepath <-- create new file
 - cat filepath <-- display file contents
- rm
 - rm filepath <-- delete given file
 - rm -r dirpath <-- delete dir with all contents
- mv
 - mv filepath destdirpath <-- move given file into given dest directory
 - mv dirpath destdirpath <-- move given dir into given dest directory
 - mv oldname newname <-- rename given file
- cp
 - cp filename newfilename <-- copy file with new name/path.
 - cp filepath destdirpath <-- copy file into given dest dir with same name.
 - cp -r dirpath destdirpath <-- copy file into given dest dir with same name.

Linux commands

- cd
 - cd ~ - change working directory to home directory
 - cd - - change working directory to old working directory
 - cd .. - change working directory to parent directory
- ls

- ls - list the contents of present working directory
- ls path - list the contents of given path
- ls -l - list the contents in detail format
 - type and permissions
 - Types of files
 - Regular file (-)
 - Directory file (d)
 - Link file (l)
 - pipe file (p)
 - socket file (s)
 - character special file (c)
 - block special file (b)
 - Permissions of files
 - r - read, w - write, x - execute
 - (rwx)user/owner, (rwx)group, (rwx)others
 - link count
 - user/owner
 - group
 - size
 - timestamp
 - name
- ls -a - list all contents along with hidden
- ls -A - list all contents along with hidden except . and ..
- ls -i - list contents with inode number
 - inode number is unique number given to every file
- ls -s - list content with size (number of blocks)
- ls -S - list content in descending order of their sizes
- touch
 - if file does not exist, empty file is created
 - if file exist, timestamp of that file is changed
- stat
 - stat file - display information of file
 - stat file1 file2 - display information of file1 and file2
 - stat -c "format" file - display file information in given format
- head
 - head file - display first 10 lines
 - head -5 file - display first 5 lines
- tail
 - tail file - display last 10 lines
 - tail -4 file - display last 4 lines
- sort

- sort file - sort the content by alphabetically
- sort -n file - sort the content by their value
 - sort command do not modify file content
- uniq
 - uniq file - display contents uniquely (truncate duplicate)
 - truncate duplicate content if it is consecutive
- rev filepath
 - Print each line reversed.
 - File contents are not modified.
- tac filepath
 - Print all lines in reverse order. The first line printed at last, while last line printed first.
- stat path
 - Display info about file or directory.
- alias
 - alias list="ls -l"
 - list will be alias/nick name to ls -l
 - list will give output same as ls -l
- unalias
 - unalias list
- which
 - which command
 - display the location of command executable.
- whereis
 - whereis command
 - display the location of command executable and also manual page location.

Redirection

- for every command input is taken from terminal, output is printed on terminal and error is also printed on terminal
- Standard streams (by default for every process, three files are opened)
 - stdin
 - stdout
 - stderr
- There are three types of redirections
 - input redirection

- input will be taken from file instead of stdin
- to do input redirection '<' symbol is used
- command < file
- output redirection
 - output will be written into file instead of stdout
 - to do output redirection '>' or '>>' symbol is used
 - command > file
 - older content of file will be over written
 - command >> file
 - content will be appended into file at the end
- error redirection
 - error will be written into file instead of stderr
 - to do output redirection '2>' or '2>>' symbol is used
 - command 2> file
 - older content of file will be over written
 - command 2>> file
 - content will be appended into file at the end

Pipe

- Using pipe, we can redirect output of any command to the input of any other command.
- Two processes are connected using pipe operator (|).
- Two processes runs simultaneously and are automatically rescheduled as data flows between them.
- If you don't use pipes, you must use several steps to do single task.
- command1 | command2
 - output of command1 will be given as input to command 2
- E.g.
 - who | wc

Shell meta characters

- '*' - zero or more occurrences of any character
- '?' - one occurrence of any character

Regular Expressions

- Find a pattern in text file(s).
- Regular expressions are patterns used to match character combinations in strings.
- A regular expression pattern is composed of simple characters, or a combination of simple and special characters e.g. /abc/, /ab*c/
- Pattern is given using regex wild-card characters.
 - Basic wild-card characters
 - \$ - find at the end of line.
 - ^ - find at the start of line.
 - [] - any single char in give range or set of chars

- [^] - any single char not in give range or set of chars
- . - any single character
- ▪ ▪ zero or more occurrences of previous character
- Extended wild-card characters
 - ? - zero or one occurrence of previous character
 - ▪ ▪ one or more occurrences of previous character
 - {n} - n occurrences of previous character
 - {,n} - max n occurrences of previous character
 - {m,} - min m occurrences of previous character
 - {m,n} - min m and max n occurrences of previous character
 - () - grouping (chars)
 - () - find one of the group of characters

grep

- Regex commands
 - grep - GNU Regular Expression Parser - Basic wild-card
 - egrep - Extended Grep - Basic + Extended wild-card
 - fgrep - Fixed Grep - No wild-card
- Command syntax
 - grep "pattern" filepath
 - grep [options] "pattern" filepath
 - -c : count number of occurrences
 - -v : invert the find output
 - -i : case insensitive search
 - -w : search whole words only
 - -R : search recursively in a directory
 - -n : show line number.

VI Editor

- sudo apt-get install vim
- VI editor works in two modes
 - command mode
 - insert mode
- press i - to go into insert mode
- press Esc - to go into command node
- VI editor commands:
 - :w - write/save into file
 - :q - quit vi editor
 - :wq - save and quit

- :y - to copy current line
 - :ny - to copy nth line
 - yy - to copy current line
 - nyy - copy n lines from current line
 - :m,ny - copy fomr mth line to nth line
 - :d - to cut current line
 - :nd - to cut nth line
 - dd - to cut current line
 - ndd - cut n lines from current line
 - :m,nd - cut fomr mth line to nth line
 - press p - to paste copied line on next line of current line
 - yw - copy from current position upto next word
 - yiw - copy current word
 - y\$ - copy from cursor position upto end of line
 - y^ - copy from cursor position upto start of line
 - vim -o - to open multiple files horizontally
 - vim -O - to open multiple files vertically
 - ctrl + ww - to go into next file
 - /pattern - to search the pattern
 - n - to go on next occurrence
 - 😊/pattern1/pattern2/ - find and replace only first occurrence of current line
 - 😊/pattern1/pattern2/g - find and replace all occurrences of current line
 - :%s/pattern1/pattern2/g - find and replace all occurrences of file
- To do setting in vi editor
 - create file into home directory as below:

```
vim ~/.vimrc
```

- add below content into it

```
set number
set autoindent
set tabstop=4
set nowrap
```

SUNBEAM



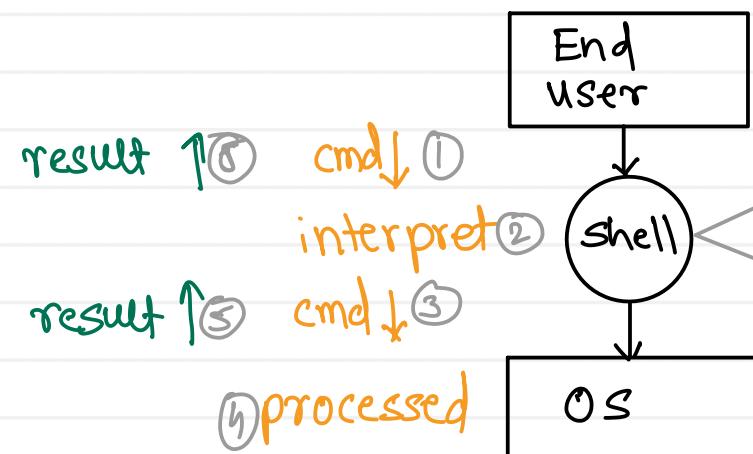
**Sunbeam Institute of Information Technology
Pune and Karad**

Module - Concepts of Operating System

Trainer - Devendra Dhande
Email – devendra.dhande@sunbeaminfo.com

User Interfacing

- user interfacing is provided by implementing one program which is called as "shell"
- shell is intermediate between end user & operating system
- shell is a command interpreter



- in Ubuntu default shell is 'bash'

BASH - Bourne Again Shell

echo \$SHELL - will show default shell

chsh - to change default shell

CLI based shell (Command Line Interface)

Windows - cmd.exe, powershell.exe

Linux - bsh(sh), bash, csh, ksh, zsh

GUI based shell (Graphical User Interface)

Windows - explorer.exe

Linux - KDE (Kommand Desktop Env)

- GNOME

(GNU Network Object Model Env)



File - types, permissions, groups

type	count of hard link	size (bytes)	name
<code>-rwxrwxr-x</code>	1 sunbeam	16753	sample.txt
permissions	user/owner	group	time stamp

File types :

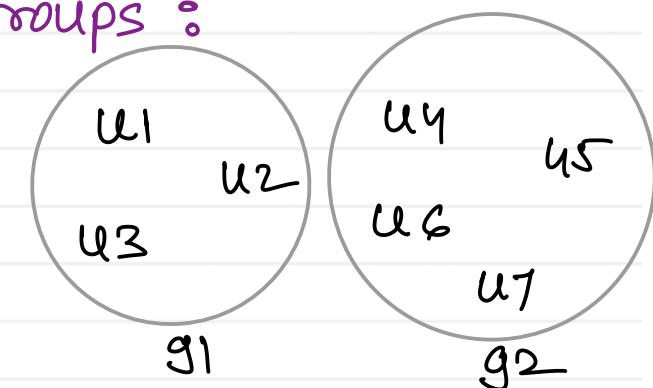
1. Regular file (-)
2. Directory file (d)
3. Link file (l)
4. Pipe file (p)
5. Socket file (s)
6. Character special file (c)
7. Block special file (b)

Permissions :

r-read
w-write
x-execute

rwx	rwx	r-x
user/owner	group	others

Groups :



file : sample.txt

User/owner : u1 : rwx
group : g1 (u2,u3) : rwx
Others : u4, u5, u6, u7 : r-x





chmod command

chmod $\pm r/w/x$ fileName

r - read, w - write, x - execute

+ : add

- : remove

chmod +r sample.txt

chmod -x sample.txt

chmod u/g/o $\pm r/w/x$ fileName

u - user/owner

g - group

o - other

chmod u-r sample.txt

chmod g+w sample.txt

chmod o+r sample.txt

human formed: rwx rwx r-x

Numeric format: 111 111 101 (775)

Existing permissions: rw- r-- r--

add execute to user & remove write from group

New permissions: rwx r-- r--

111 100 100

(744)

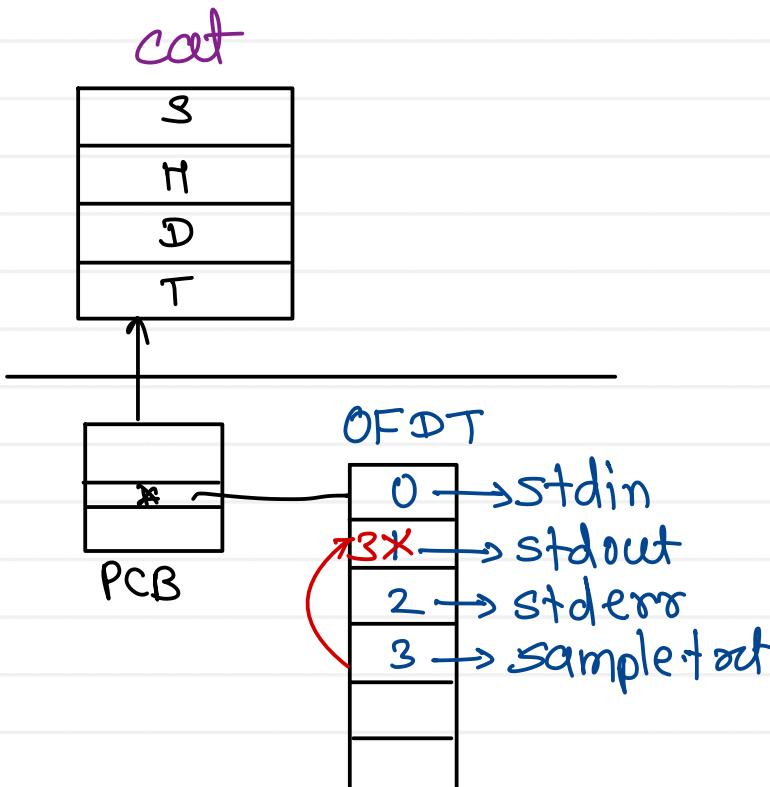
chmod 744 sample.txt

chmod mode fileName

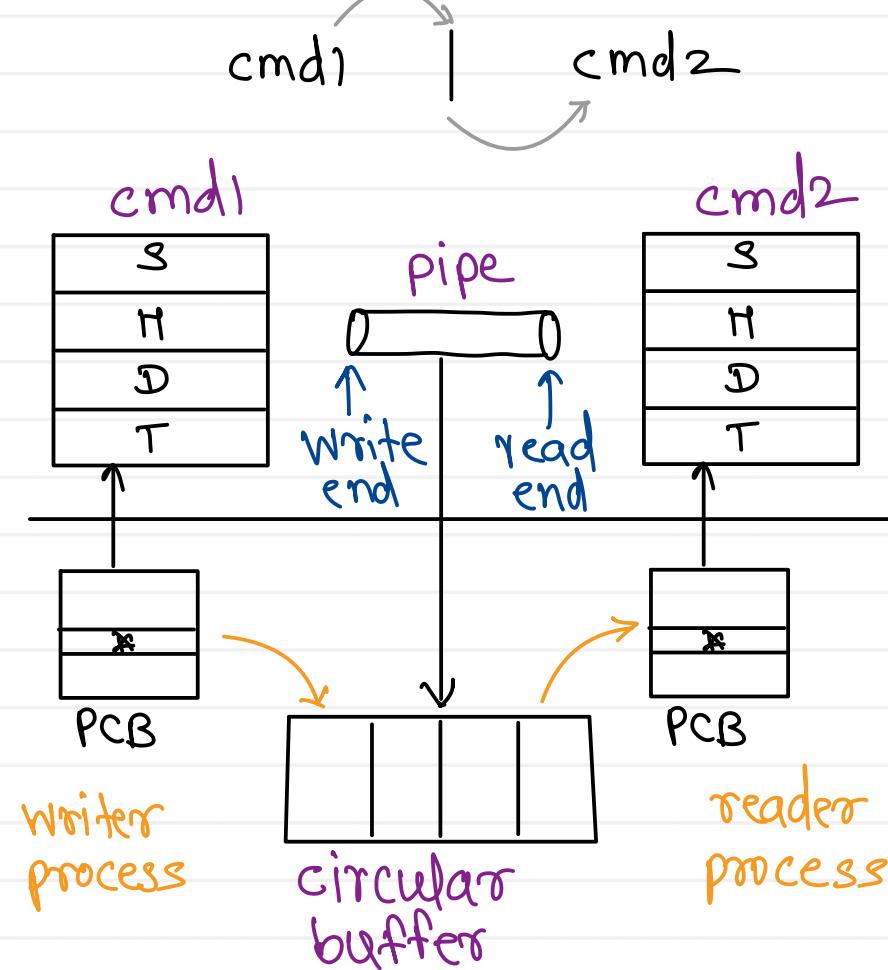


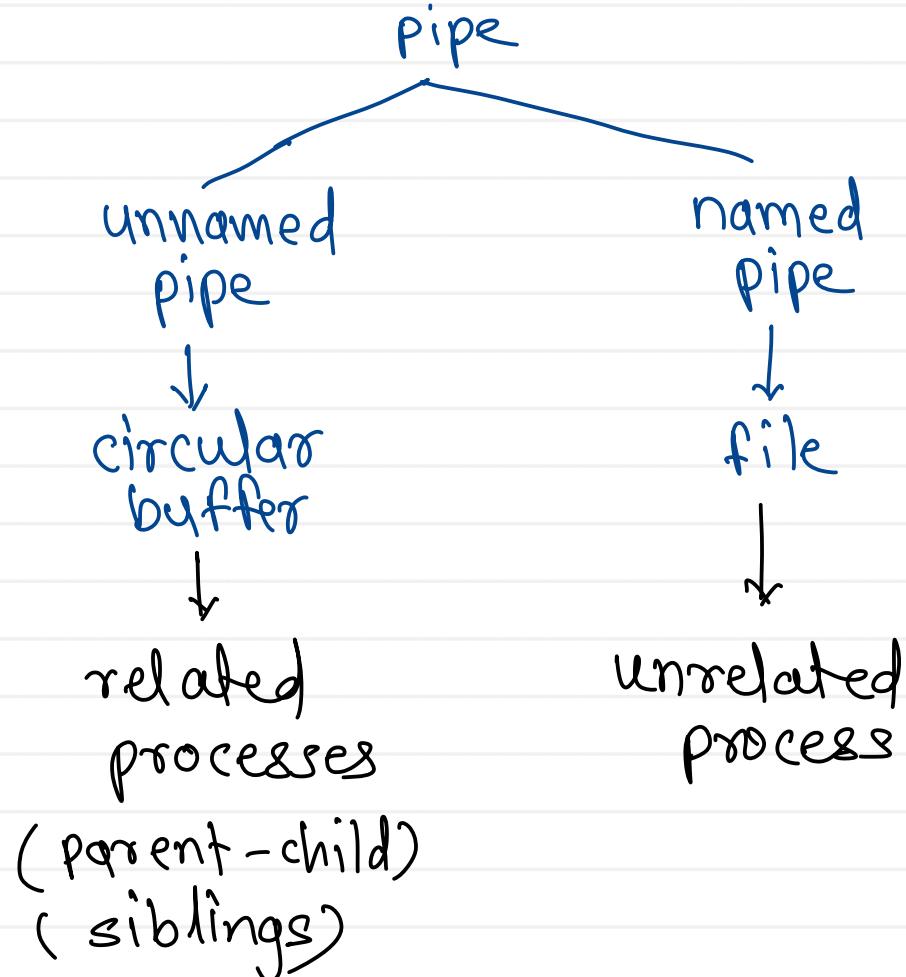
Redirection and Pipe

`cat > sample.txt`



`sort numbers.txt | uniq`







Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

Classification of OS

- OS can be categorized based on the target system (computers).
 - Mainframe systems
 - Desktop systems
 - Multi-processor (Parallel) systems
 - Distributed systems
 - Hand-held systems
 - Real-time systems

Resident Monitor

- Early (oldest) OS resides in memory and monitor execution of the programs. If it fails, error is reported.
- OS provides hardware interfacing that can be reused by all the programs.

Batch Systems

- The batch/group of similar programs is loaded in the computer, from which OS loads one program in the memory and execute it. The programs are executed one after another.
- In this case, if any process is performing IO, CPU will wait for that process and hence not utilized efficiently.

Multi-Programming

- In multi-programming systems, multiple programs can be loaded in the memory.
- The number of programs that can be loaded in the memory at the same time, is called as "degree of multi-programming".
- In these systems, if one of the processes is performing IO, CPU can continue execution of another program. This will increase CPU utilization.
- Each process will spend some time for CPU computation (CPU burst) and some time for IO (IO burst).
 - If CPU burst > IO burst, then process is called as "CPU bound".
 - If IO burst > CPU burst, then process is called as "IO bound".
- To efficiently utilize CPU, a good mix of CPU bound and IO bound processes should be loaded into memory. This task is performed by a unit of OS called as "Job scheduler" OR "Long term scheduler".
- If multiple programs are loaded into the RAM by job scheduler, then one of process need to be executed (dispatched) on the CPU. This selection is done by another unit of OS called as "CPU scheduler" OR "Short term scheduler".

Multi-tasking OR time-sharing

- CPU time is shared among multiple processes in the main memory is called as "multi-tasking".
- In such system, a small amount of CPU time is given to each process repeatedly, so that response time for any process < 1 sec.
- With this mechanism, multiple tasks (ready for execution) can execute concurrently.
- There are two types of multi-tasking:
 - Process based multitasking: Multiple independent processes are executing concurrently. Processes running on multiple processors called as "multi-processing".

- Thread based multi-tasking OR multi-threading: Multiple parts/functions in a process are executing concurrently.

Multiprocessor systems

- The systems in which multiple processors are connected in a close circuit is called as "multiprocessor computer".
- The programs/OS take advantage of multiple processors in the computer are called as "Multiprocessing" programs/OS.
 - Windows Vista: First Windows OS designed for multi-processing.
 - Linux 2.5+: Linux started supporting multi-processing.
- Modern PC architectures are multi-core arch i.e. multiple CPUs on single chip.
- Since multiple tasks can be executed on these processors simultaneously, such systems are also called as "parallel systems".
- Parallel systems have more throughput (Number of tasks done in unit time).
- There are two types of multiprocessor systems:
 - Asymmetric Multi-processing
 - Symmetric Multi-processing

Asymmetric Multi-processing

- OS treats one of the processor as master processor and schedule task for it. The task is in turn divided into smaller tasks and get them done from other processors.

Symmetric Multi-processing

- OS considers all processors at same level and schedule tasks on each processor individually.
- All modern desktop systems are SMP.

Multi-user

- Multiple users can execute multiple tasks concurrently on the same systems. e.g. IBM 360, UNIX, Windows Servers, etc.
- Each user can access system via different terminal.
- There are many UNIX commands to track users and terminals.
 - tty, who, who am i, whoami, w

Desktop systems

- Personal computers -- desktop and laptops
- User convenience and Responsiveness
- Examples: Windows, Mac, Linux, few UNIX, ...

Handheld systems

- OS installed on handheld devices like mobiles, PDAs, iPODs, etc.
- Challenges:
 - Small screen size
 - Low end processors

- Less RAM size
- Battery powered
- Examples: Symbian, iOS, Linux, PalmOS, WindowsCE, etc.

Realtime systems

- The OS in which accuracy of results depends on accuracy of the computation as well as time duration in which results are produced, is called as "RTOS".
- If results are not produced within certain time (deadline), catastrophic effects may occur.
- These OS ensure that tasks will be completed in a definite time duration.
- Time from the arrival of interrupt till begin handling of the interrupt is called as "Interrupt Latency".
- RTOS have very small and fixed interrupt latencies.
- RTOS Examples: uC-OS, VxWorks, pSOS, RTLinux, FreeRTOS, etc.

Distributed systems

- Multiple computers connected together in a close network is called as "distributed system".
- Its advantages are high availability (24x7), high scalability (many clients, huge data), fault tolerance (any computer may fail).
- The requests are redirected to the computer having less load using "load balancing" techniques.
- The set of computers connected together for a certain task is called as "cluster". Examples: Linux.

Process Life Cycle

Process States

- New
 - New process PCB is created and added into job queue. PCB is initialized and process get ready for execution.
- Ready
 - The ready process is added into the ready queue. Scheduler pick a process for scheduling from ready queue and dispatch it on CPU.
- Running
 - The process runs on CPU. If process keeps running on CPU, the timer interrupt is used to forcibly put it into ready state and allocate CPU time to other process.
- Waiting
 - If running process request for IO device, the process waits for completion of the IO. The waiting state is also called as sleeping or blocked state.
- Terminated
 - If running process exits, it is terminated.

- Linux: TASK_RUNNING (R), TASK_INTERRUPTIBLE (S), TASK_UNINTERRUPTIBLE (D), TASK_STOPPED(T), TASK_ZOMBIE (Z), TASK_DEAD (X)

Types of Scheduling

Non-preemptive

- The current process gives up CPU voluntarily (for IO, terminate or yield).
- Then CPU scheduler picks next process for the execution.
- If each process yields CPU so that other process can get CPU for the execution, it is referred as "Co-operative scheduling".

Preemptive

- The current process may give up CPU voluntarily or paused forcibly (for high priority process or upon completion of its time quantum)

Scheduling criteria's

CPU utilization: Ideal - max

- On server systems, CPU utilization should be more than 90%.
- On desktop systems, CPU utilization should around 70%.

Throughput: Ideal - max

- The amount of work done in unit time.

Waiting time: Ideal - min

- Time spent by the process in the ready queue to get scheduled on the CPU.
- If waiting time is more (not getting CPU time for execution) -- Starvation.

Turn-around time: Ideal - CPU burst + IO burst

- Time from arrival of the process till completion of the process.
- CPU burst + IO burst + (CPU) Waiting time + IO Waiting time

Response time: Ideal - min

- Time from arrival of process (in ready queue) till allocated CPU for first time.

Scheduling Algorithms

FCFS

- Process added first in ready queue should be scheduled first.
- Non-preemptive scheduling
- Scheduler is invoked when process is terminated, blocked or gives up CPU is ready for execution.
- Convoy Effect: Larger processes slow down execution of other processes.

SJF

- Process with lowest burst time is scheduled first.
- Non-preemptive scheduling
- Minimum waiting time

SRTF - Shortest Remaining Time First

- Similar to SJF - but Preemptive scheduling
- Minimum waiting time

Priority

- Each process is associated with some priority level. Usually lower the number, higher is the priority.
- Preemptive scheduling or Non Preemptive scheduling
- Starvation
 - Problem may arise in priority scheduling.
 - Process not getting CPU time due to other high priority processes.
 - Process is in ready state (ready queue).
 - May be handled with aging -- dynamically increasing priority of the process.

Round-Robin

- Preemptive scheduling
- Process is assigned a time quantum/slice.
- Once time slice is completed/expired, then process is forcibly preempted and other process is scheduled.
- Min response time.

Thread concept

- Threads are used to execute multiple tasks concurrently in the same program/process.
- Thread is a light-weight process.
 - For each thread new control block and stack is created. Other sections (text, data, heap, ...) are shared with the parent process.
 - Inter-thread communication is much faster than inter-process communication.
 - Context switch between two threads in the same process is faster.
- Thread stack is used to create function activation records of the functions called/executed by the thread.

Process vs Thread

- In modern OS, process is a container holding resources required for execution, while thread is unit of execution/scheduling.
- Process holds resources like memory, open files, IPC (e.g. signal table, shared memory, pipe, etc.).
- PCB contains resources information like pid, exit status, open files, signals/ipc, memory info, etc.
- CPU time is allocated to the threads. Thread is unit of execution.
- TCB contains execution information like tid, scheduling info (priority, sched algo, time left, ...), Execution context, Kernel stack, etc.

- terminal> ps -e -o pid,nlwp,cmd
- terminal> ps -e -m -o pid,tid,nlwp

main thread

- For each process one thread is created by default called as main thread.
- The main thread executes entry-point function of the process.
- The main thread use the process stack.
- When main thread is terminated, the process is terminated.
- When a process is terminated, all threads in the process are terminated.

SUNBEAM

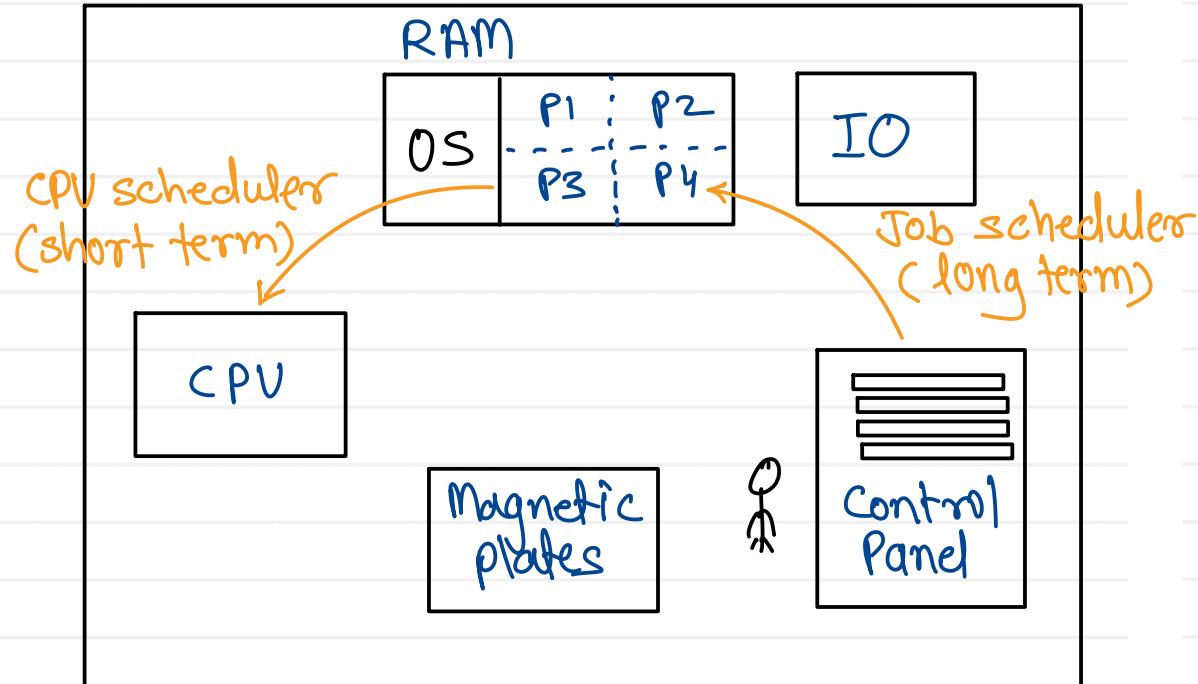


**Sunbeam Institute of Information Technology
Pune and Karad**

Module - Concepts of Operating System

Trainer - Devendra Dhande
Email – devendra.dhande@sunbeaminfo.com

Types of Operating system



1. Resident monitor
2. Batch system
3. Multiprogramming system:
 - multiple programs are loaded inside RAM (memory) at a time.

CPV burst : time spent by process on CPU
IO burst : time spent by process to perform IC

CPV burst > IO burst : CPV bound process

IO burst > CPV burst : IO bound process

Degree of multiprogramming :
No. of programs loaded into RAM

- mixture of CPV & IO bound processes was getting loaded inside RAM to balance CPV & IO load.



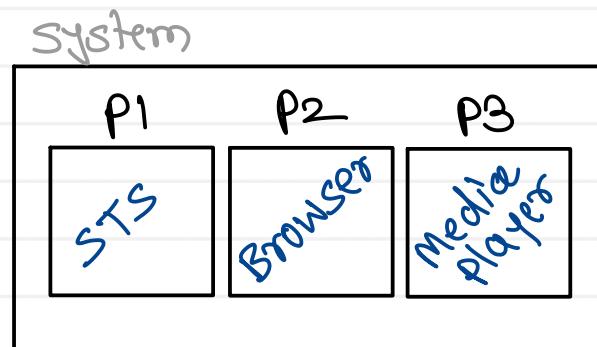
Types of Operating system

4. Time Sharing system (Multitasking system) :

- CPU time is shared in all the processes of RAM
- Response time < 1 sec

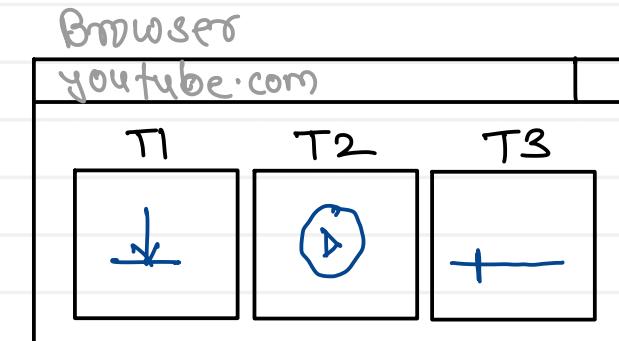
There are two types of Multitasking.

1. Process based multitasking



- system wide multitasking

2. Thread based Multitasking
(Multithreading)

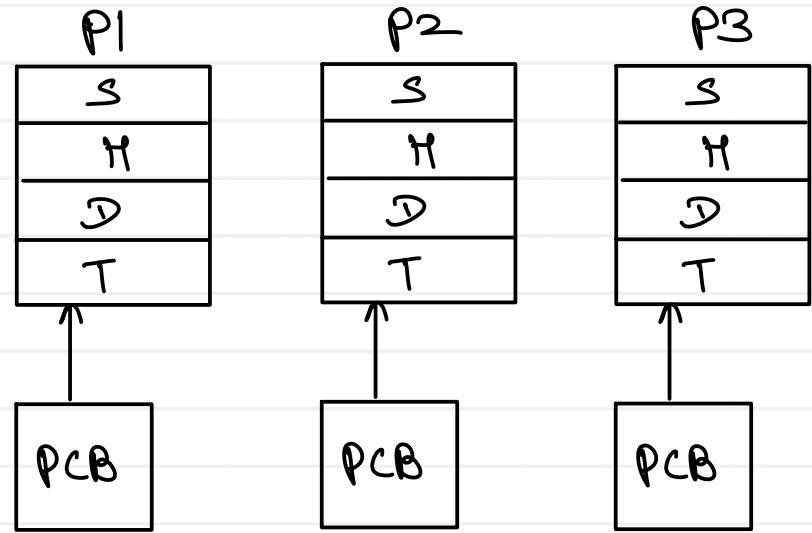


- multitasking within process

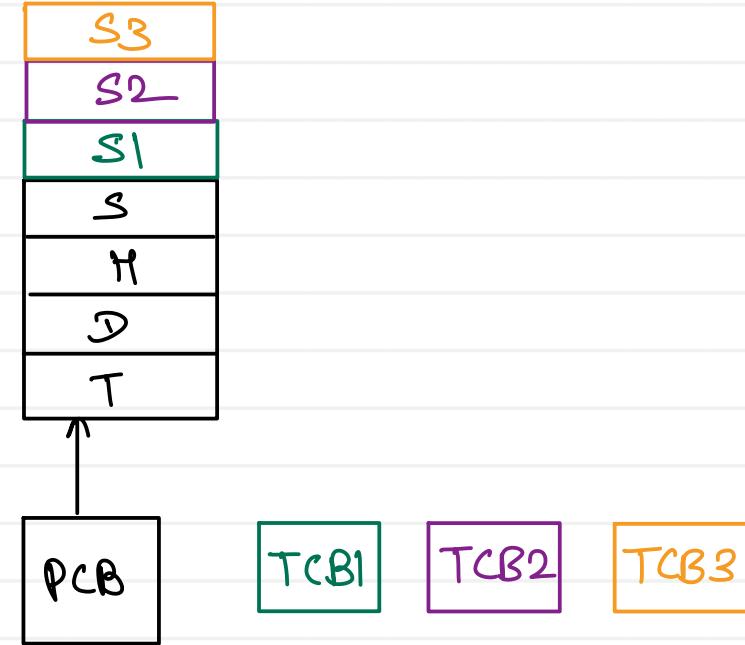


Process Vs Thread

- processes are created inside system



- Thread are always created inside process



- Process is container of resources
- thread is alive entity in that container.
- process never runs on CPU, it is a thread which runs on CPU.

- Thread is light weight process

Types of Operating system

5. Multiprocessing system :

- multiple processors (CPUs) are fitted on single chip. such chips are called as "multiprocessor"/ "multicore".
- OS starts scheduling processes for multiple cores.
- multiple instructions / processes can be processed parallelly.

i. Symmetric multiprocessing :

- OS treats every core equally & schedules separate process for every core.

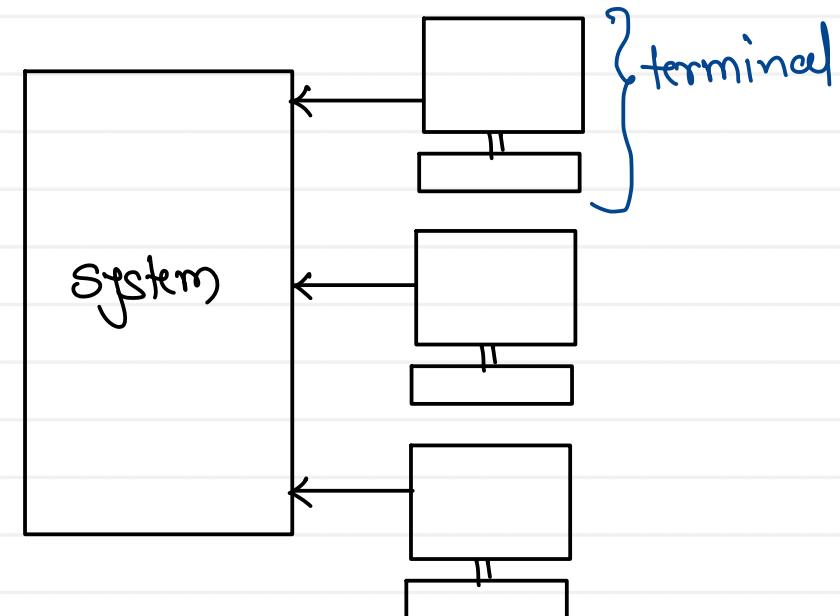
ii Asymmetric multiprocessing :

- OS schedules process for only one core & internally that core divides job into remaining cores

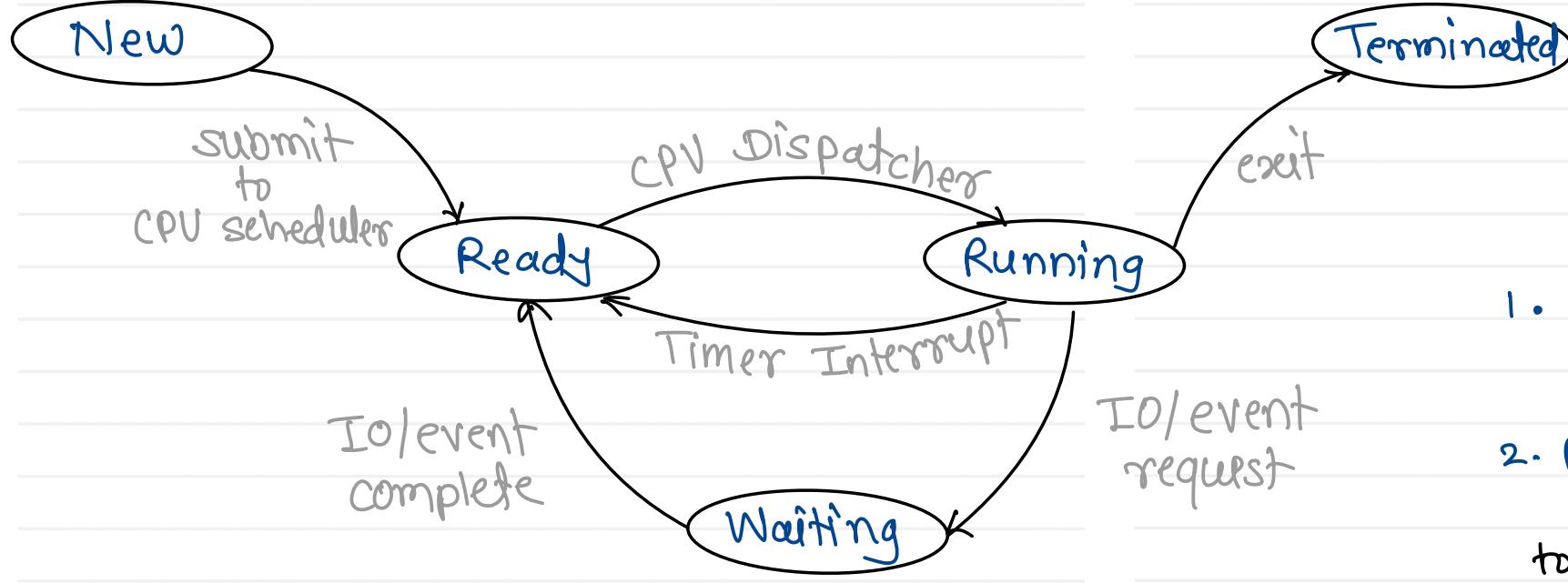
Windows Vista , Linux 2.6

6. Multiuser system :

multiple users can operate single system at a time.



Process life cycle



OS data structures:

1. Job queue / process list
 - all processes of system
2. Ready queue
 - processes which are ready to execute on CPU
3. Waiting queues
 - processes which are waiting for IO / event.
 - waiting queues are created for every IO device / event separately.

Context switching

Execution context :

- values of CPU registers

↑
small memory elements
available inside CPU

1. General purpose registers : (operands / results)

AH, AL, BH, BL, CH, CL, DH, DL

2. Segment registers : (base addresses of sections)

CS, DS, SS, ES

3. Offset registers : (displacement of variables)

BP, SP, SI, DI

4. Special registers : (address of next instruction)

PC / IR

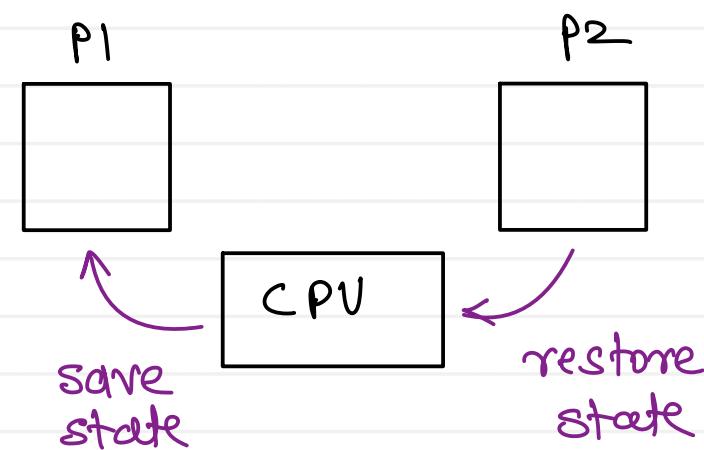
5. Flag register : (status of current operation)

Z, S, P, O

- CPU dispatcher loads , execution context from PCB of process on CPU

Context switching :

- changing the process of CPU
- execution context of running process is stored into its PCB & execution context of selected process is restored on CPU





Types of scheduling

CPU scheduler is called in below 4 conditions:

1. Running → Terminated } voluntarily
2. Running → Waiting
3. Running → Ready } forcefully
4. Waiting → Ready

CPU scheduling algorithms:

1. FCFS
2. SJF (SRTF)
3. Priority
4. RR

Types of scheduling :

1. Non pre emptive scheduling

- CPU access is given to next process voluntarily.

- cooperative scheduling.

2. Pre emptive scheduling

- CPU access is given to next process forcefully.

(case 1 & 2)

(all cases)



CPU scheduling criteria's

1. CPU Utilization : (Max)

- it determines how much time CPU is busy or idle.
e.g. CPU Utilization = 70 %

70 % CPU busy & 30 % CPU is idle
server OS - CPU Utilization = 90 %
desktop OS - CPU Utilization = 70 %

2. Throughput : (Max)

- amount of work done in unit time
- it measured as no. of processes completed in unit time

3. Waiting time : (Min)

- time spent by process in ready queue.

$$WT = TAT - CPU \text{ burst}$$

4. Response time : (Min)

- time from arrival in ready queue to first time executed on CPU

$$RT = ST - AT$$

5. Turn Around Time (TAT) : (Min)

- time spent by process inside RAM

$$TAT = CT - AT$$

$$TAT = \frac{CPU}{Waiting \text{ time}} + \frac{CPU}{burst} + \frac{IO}{Waiting \text{ time}} + \frac{IO}{burst}$$

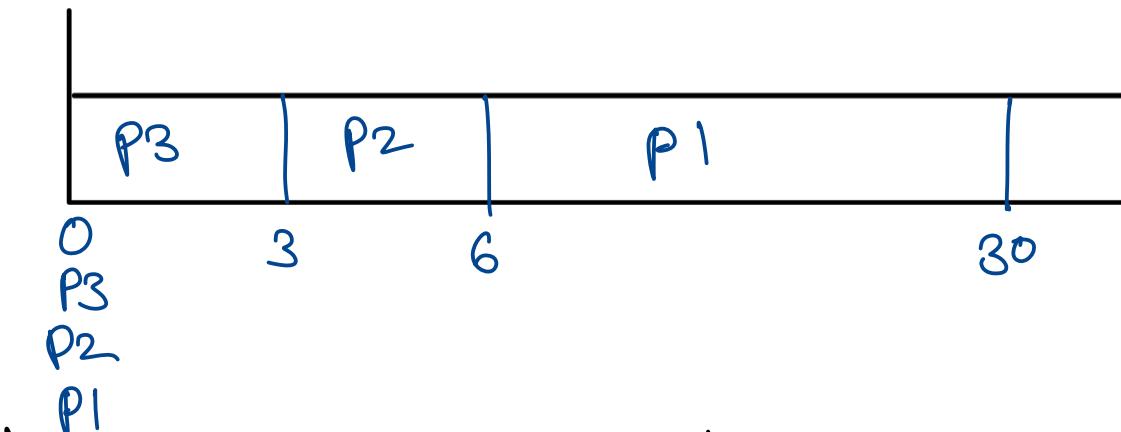
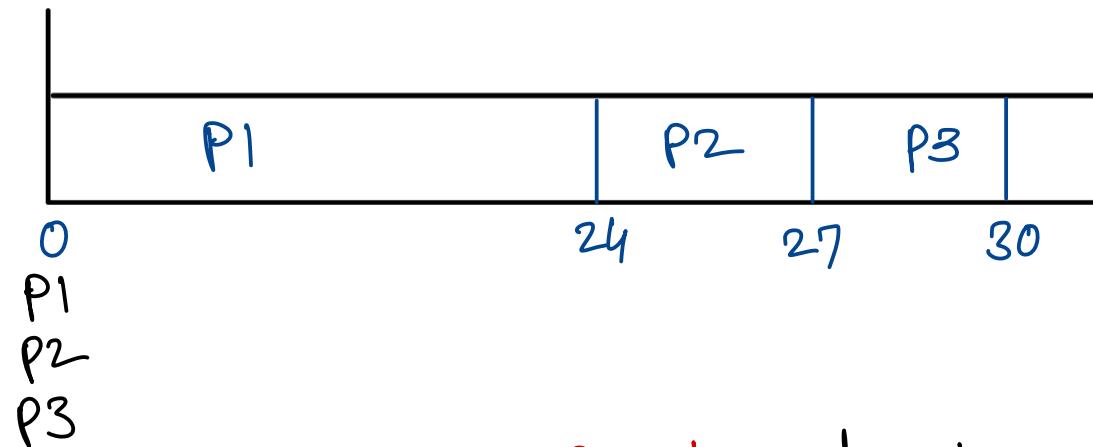
FCFS (First Come First Serve) (Non preemptive)

Process	Arrival	CPU Burst
P1	0	24
P2	0	3
P3	0	3

RT WT TAT
 0 0 24
 24 24 27
 27 27 30

Process	Arrival	CPU Burst	RT	WT	TAT
P3	0	3	0	0	3
P2	0	3	3	3	6
P1	0	24	6	6	30

Gantt's chart

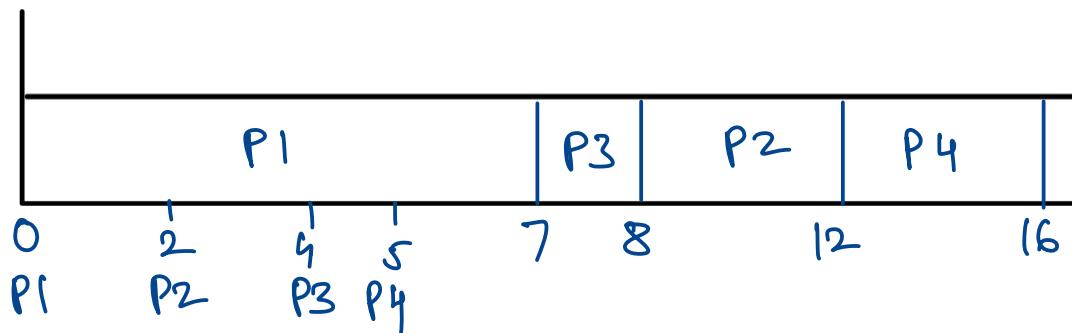


Conroy effect : due to arrival of longer process early, all other processes has to wait for longer time

SJF (Shortest Job First)

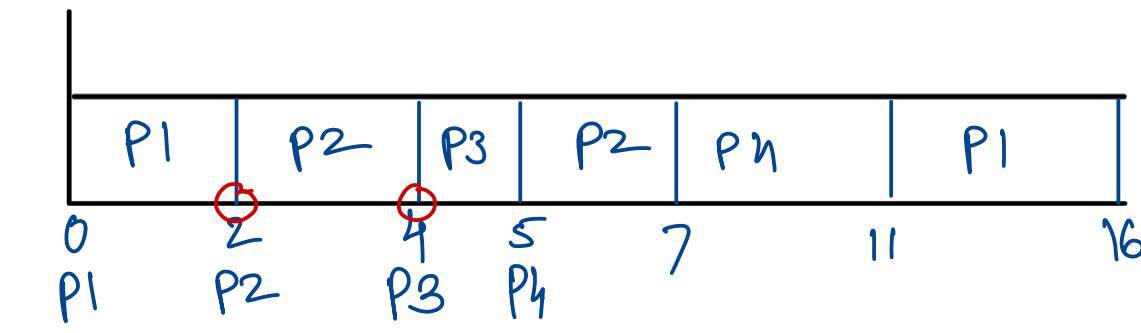
(Non pre emptive)

Process	Arrival	CPU Burst	WT	RT	TAT
P1	0	7	0	0	7
P2	2	4	6	6	10
P3	4	1	3	3	4
P4	5	4	7	7	11



Shortest Remaining Time First (SRTF)
(preemptive)

Process	Arrival	CPU Burst	Remain time	WT	RT	TAT
P1	0	7	5	9	0	16
P2	2	4	2	1	0	5
P3	4	1	0	0	1	1
P4	5	4	2	2	2	6



Starvation: due to longer CPU burst process has to wait for longer time to get CPU access

Priority

(Non pre emptive)

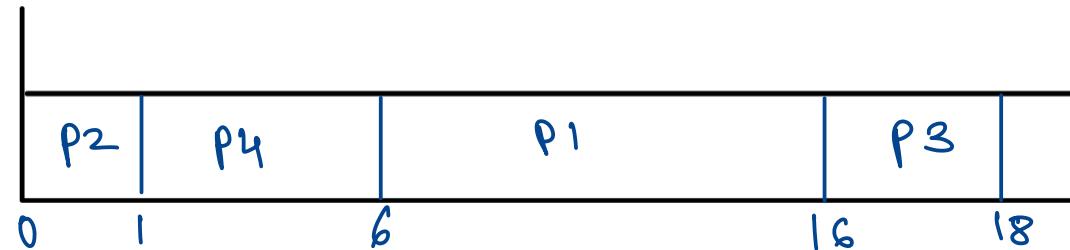
Process	Arrival	CPU Burst	Priority
P1	0	10	3
P2	0	1	1 (H)
P3	0	2	4 (L)
P4	0	5	2

	WT	RT	TAT
P1	6	6	16
P2	0	0	1
P3	16	16	18
P4	1	1	6

(Pre emptive)

Process	Arrival	CPU Burst	Priority
P1	0	10	3
P2	1	1	1
P3	3	2	4
P4	0	5	2

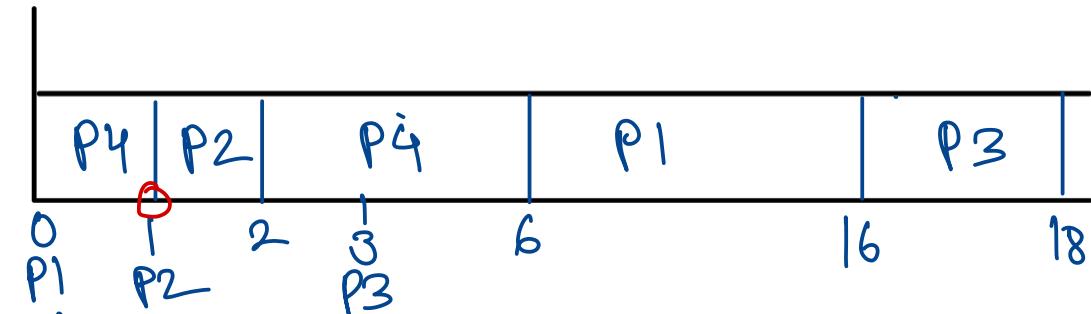
	WT	RT	TAT
P1	6	6	16
P2	0	0	1
P3	13	13	15
P4	1	0	6



P1(6)
 P2(8)
 P3(7)
 P4(6)
 P5(5)
 P6(5)
 P7(6)

P1
 P5
 P4
 P3
 P6
 P7
 P2

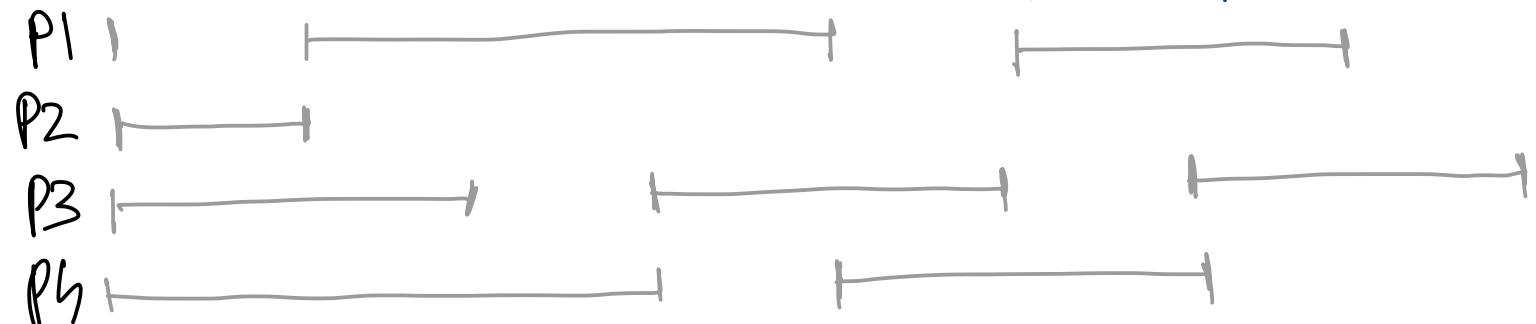
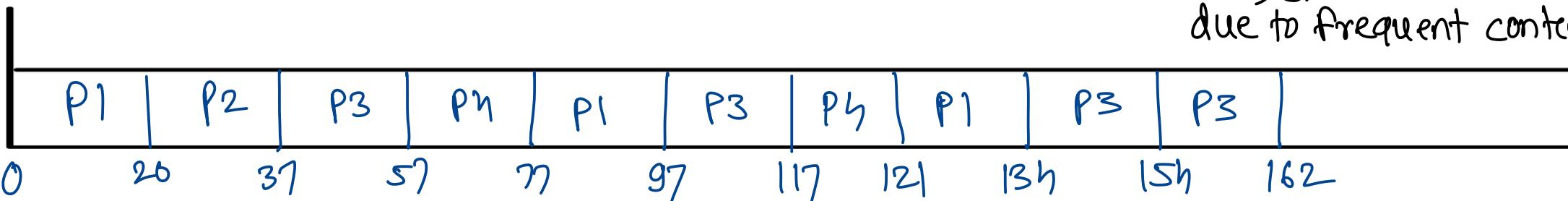
starvation:
 due to low priority process
 do not get enough time to execute
 on CPU
aging:
 priority of process is increased
 gradually.



RR (Round Robin)

Process	CPU Burst
P1	53
P2	17
P3	68
P4	24

33, 13, 0 0 + 57 + 24
 0 20
 48, 28, 8 37 + 40 + 17
 4, 0 57 + 40



Time quantum : CPU time slice

$$TQ = 20$$

- next process will be scheduled in each time quantum.

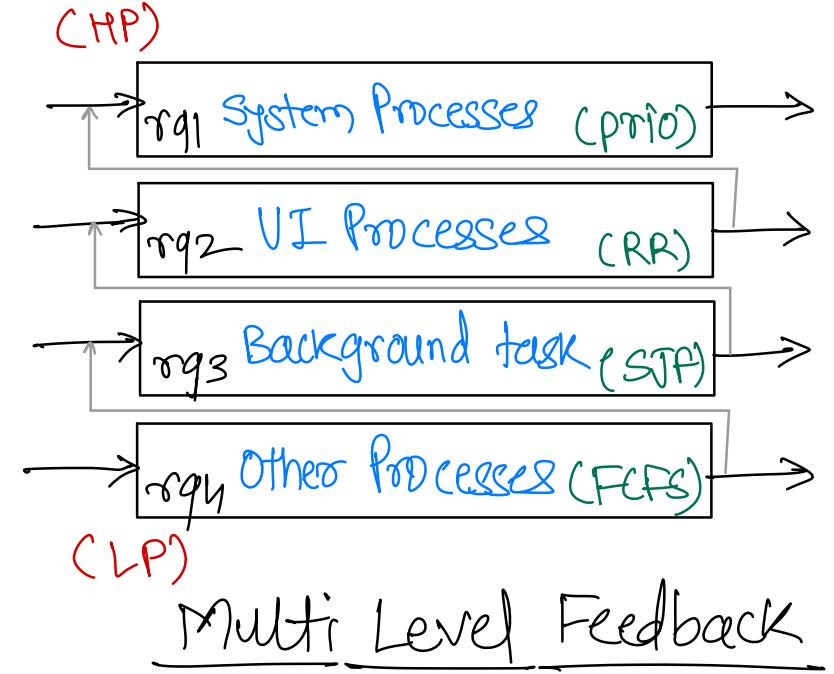
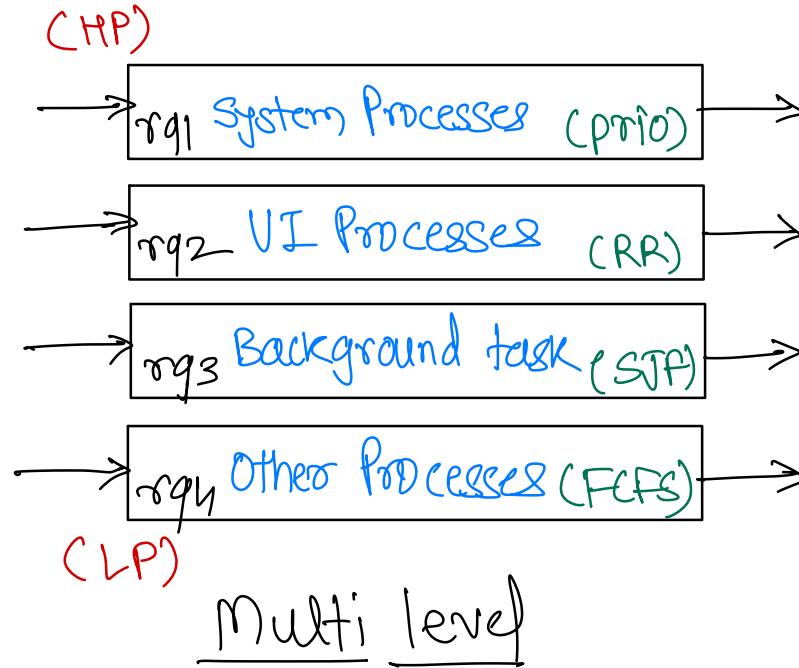
$$TQ = 100$$

↳ behave like FCFS

$$TQ = 4$$

↳ CPU overhead will increase due to frequent context switch

Multi Level Ready Queue





Thank you!!!

Devendra Dhande

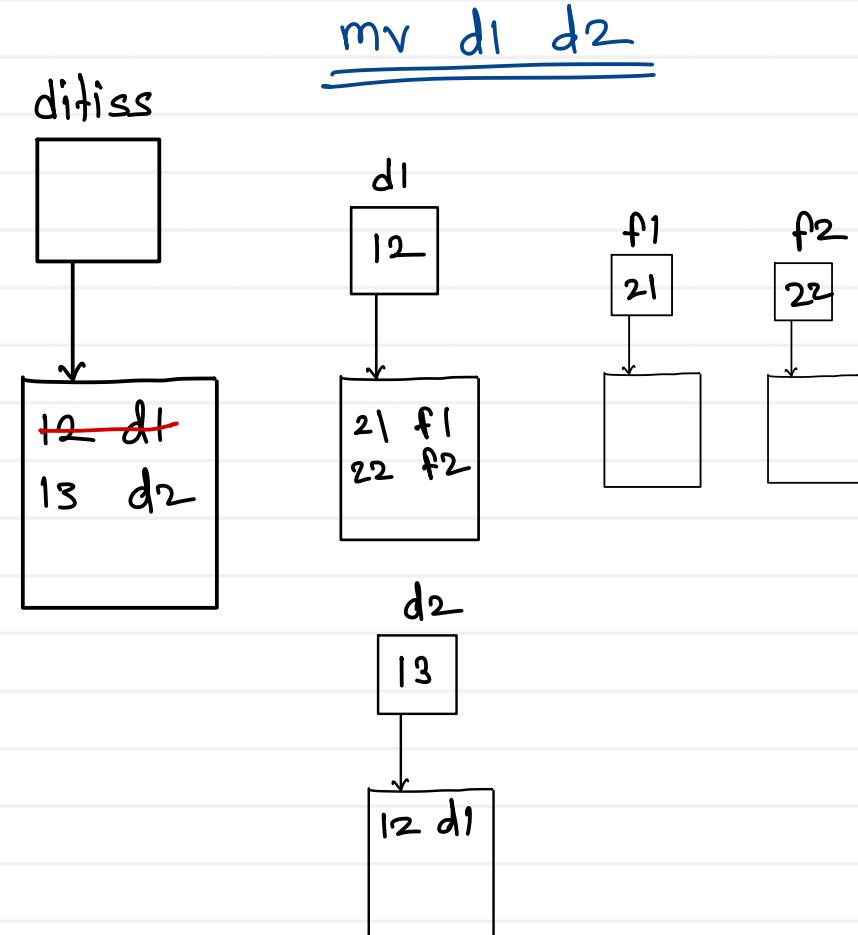
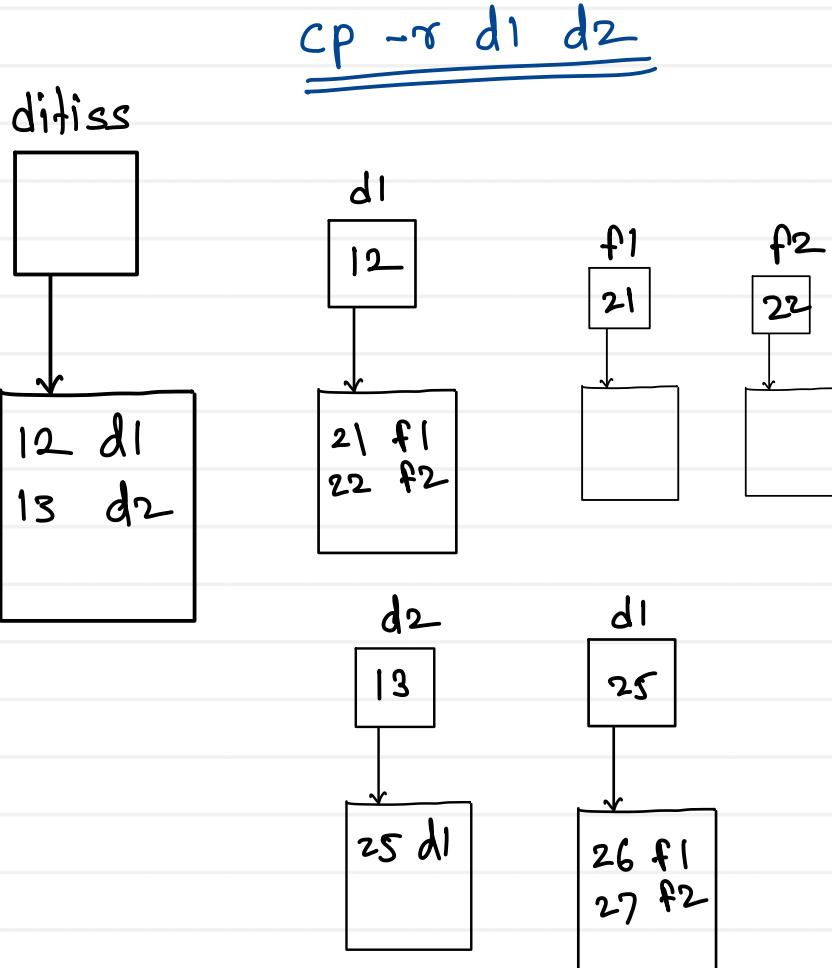
devendra.dhande@sunbeaminfo.com



**Sunbeam Institute of Information Technology
Pune and Karad**

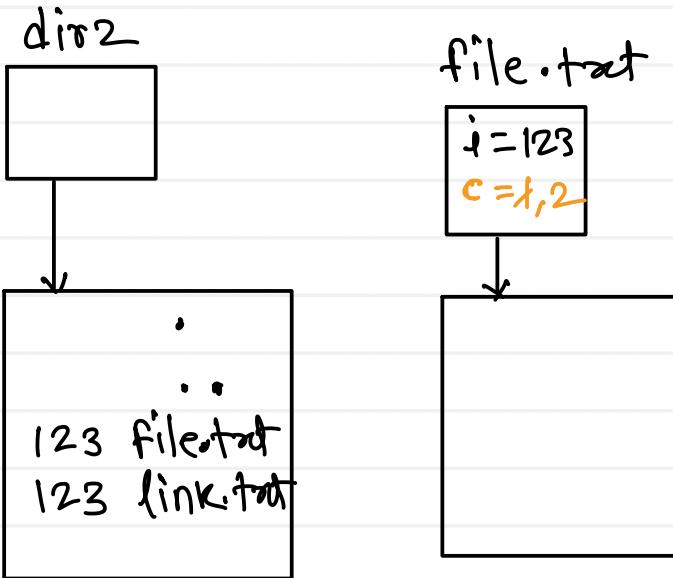
Module - Concepts of Operating System

Trainer - Devendra Dhande
Email – devendra.dhande@sunbeaminfo.com

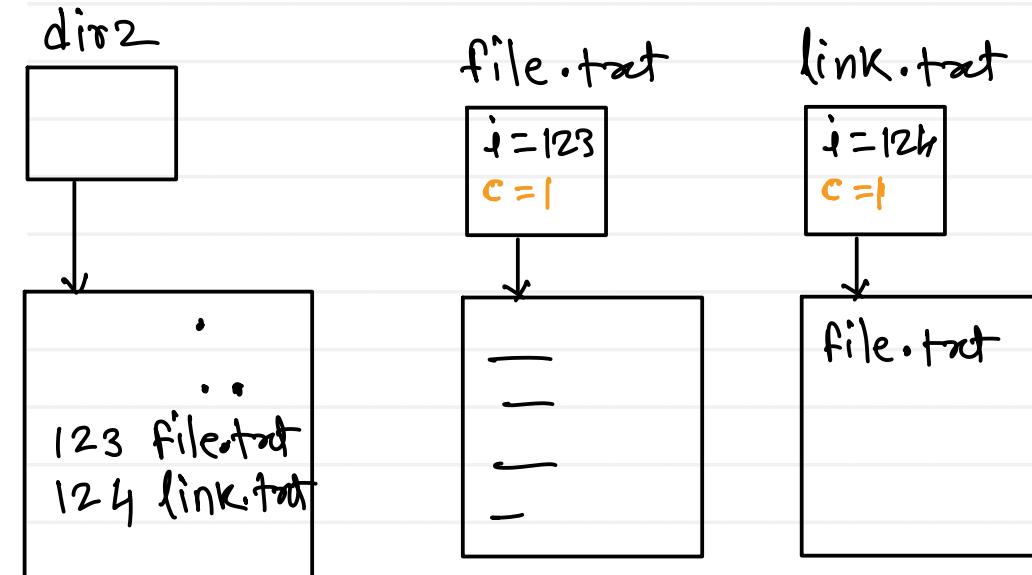


Hard link vs Symbolic link

In file.txt link.txt



In -S file.txt link.txt





Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com



Linux Shell Script

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com



Sunbeam Infotech

www.sunbeaminfo.com

Shell Script : Introduction

- Shell script is collection of commands along with programming constructs.
- Shell script syntax will differ from shell to shell.
- Shell scripts are interpreted by shell (interpreter - line by line).
- Speed is slower.

- Comments in shell script begin with # symbol.

- **echo command**
 - -e : enable escape sequences e.g. \n, \t, ...
 - -n : no newline after echo.

- **shebang line (#!/bin/bash)**
 - Line 1 of shell script should contain name of shell program which will execute that script followed by #!.
 - While running script on terminal (./demo01.sh), OS reads first line and load corresponding shell program, which in turn execute that shell script.



Sunbeam Infotech

www.sunbeaminfo.com

Shell Script : Variables

- **Shell variables**

- Shell scripts are type-less. There is no concept of data types.
- Also no need to declare variables before using them.

- Assign value to variable

- varname=value

- Read value from variable

- \$varname

- Assign output of command to variable

- varname='command ...'
- varname=\$(command ...)

- To perform arithmetic – expr and bc

- **Environment variables**

- Few variables are initialized from values in the environment.
- Normally these variables are in uppercase to distinguish from user defined variables.
- Variables created depends on your personal configuration.

- e.g.

- \$HOME – gives home directory
- \$PATH – path of all executables
- \$USER – gives user name
- \$SHELL – gives which shell is currently running.



Shell Script : Conditions

- Test conditions and perform different actions based on those decisions.
- For checking conditions you can use following two syntaxes:
 - test condition
 - [condition]
- Condition types that can be used are of three types:
 - String comparison
 - Arithmetic comparison
 - File conditions

String Comparison	Result
str1 = str2	True if str1 and str2 are equal
str1 != str2	True if str1 and str2 are not equal
-n str	True if the str is not null
-z string	True if the str is null (empty)

Arith Comparison	Result
exp1 -eq exp2	True if equal
exp1 -ne exp2	True if not equal
exp1 -gt exp2	True if exp1 is greater than exp2
exp1 -ge exp2	True if exp1 is greater or equal exp2
exp1 -lt exp2	True if exp1 is less than exp2
exp1 -le exp2	True if exp1 is less or equal exp2

File Conditionals	Result
-e file	True if file exists
-f file	True if file is regular file
-d file	True if file is directory
-r file	True if file readable
-w file	True if file is writable
-x file	True if file executable



Shell Script : Control Structures

- if

```
if [ condition ]
then
    # ...
fi
```

```
if [ condition ]
then
    # ...
elif [ condition ]
then
    # ...
else
    # ...
fi
```

```
if [ condition ]
then
    # ...
else
    # ...
fi
```

```
if [ condition ]
then
    # ...
else
    if [ condition ]
    then
        # ...
    else
        # ...
    fi
fi
```

- case

```
case $var in
c1|const1|case1)
    # ...
;;
c2|case2)
    # ...
;;
c3)
    # ...
;;
*) esac
```

- for

```
for var in collection
do
    # ...
done
```

- while

```
# initialization
while [ condition ]
do
    # ...
    # modification
done
```

- until

```
# initialization
until [ condition ]
do
    # ...
    # modification
done
```



Shell Script : Functions

- Function without return

```
# function definition:
function fn_name()
{
    # args are accessed as $1, $2, $3, ...
}

# function call:
fn_name arg1 arg2 arg3
```

- Function with return

```
# function definition:
function fn_name()
{
    # args are accessed as $1, $2, $3, ...
    # ...
    echo result
}

# function call:
var=$(fn_name arg1 arg2 arg3)
```



Shell Script : Positional Parameters

- Positional parameters (like command line arguments in C)
- While executing shell script on command line, we can pass additional information called as "positional parameters".
- terminal> ./dem.sh one two three four
- To access positional parameters in the script: \$1 \$2 \$3 \$4 ... \$9
- List of all positional parameters: \$*
- Shell script name: \$0
- Number of positional parameters: \$#
- **shift N** command is used to skip N parameters from left
- This will enable access to the next parameters.
- **N+1** parameter will become \$1
- **N+2** parameter will become \$2



Shell Script : Array

- Array is collection of values
 - arr=(1,2,3,4,5)
- To print all values: \${arr[*]} or \${arr[@]}
- To print individual element \${arr[i]}
- To print number of elements: \${#arr[*]}
- declaration is optional
 - declare -a arr



Shell Script : .bashrc and .profile

- .bashrc file
 - .bashrc is shell script that is executed when a new CLI bash shell is started.
 - We can add commands to be executed when new shell starts.
 - Example:
 - alias c=clear
 - echo "Welcome to bash!"
 - export PATH=/some/path:\$PATH
 - To edit the file
 - terminal> vim ~/.bashrc
 - Add your commands to the end of file.
 - These changes will be visible when new shell is started.
 - Close current terminal and open new terminal.
- .profile file
 - .profile is shell script that is executed when new login shell is started.
 - This will run for tty terminals or gui terminals.
 - When we need to execute some commands when any new login is done, those commands should be written in .profile.



Sunbeam Infotech

www.sunbeaminfo.com



Thank you!

Devendra Dhande <devendra.dhande@sunbeaminfo.com>



Sunbeam Infotech

www.sunbeaminfo.com

Memory Management

- In multi-programming OS, multiple programs are loaded in memory.
- RAM memory should be divided for multiple processes running concurrently.
- Memory Mgmt scheme used by any OS depends on the MMU hardware used in the machine.
- There are three memory management schemes available (as per MMU hardware).
 1. Contiguous Allocation
 2. Segmentation
 3. Paging

Contiguous Allocation

Fixed Partition

- RAM is divided into fixed sized partitions.
- This method is easy to implement.
- Number of processes are limited to number of partitions.
- Size of process is limited to size of partition.
- If process is not utilizing entire partition allocated to it, the remaining memory is wasted. This is called as "internal fragmentation".

Dynamic Partition

- Memory is allocated to each process as per its availability in the RAM. After allocation and deallocation of few processes, RAM will have few used slots and few free slots.
- OS keeps track of free slots in form of a table.
- For any new process, OS uses one of the following mechanisms to allocate the free slot.
 - First Fit: Allocate first free slot which can accommodate the process.
 - Best Fit: Allocate that free slot to the process in which minimum free space will remain.
 - Worst Fit: Allocate that free slot to the process in which maximum free space will remain.
- Statistically it is proven that First fit is faster algo; while best fit provides better memory utilization.
- Memory info (physical base address and size) of each process is stored in its PCB and will be loaded into MMU registers (base & limit) during context switch.
- CPU requests virtual address (address of the process) and is converted into physical address by MMU as shown in diag.
- If invalid virtual address is requested by the CPU, process will be terminated.
- If amount of memory required for a process is available but not contiguous, then it is called as "external fragmentation".
- To resolve this problem, processes in memory can be shifted/moved so that max contiguous free space will be available. This is called as "compaction".

Virtual Memory

- The portion of the hard disk which is used by OS as an extension of RAM, is called as "virtual memory".
- If sufficient RAM is not available to execute a new program or grow existing process, then some of the inactive process is shifted from main memory (RAM), so that new program can execute in RAM (or existing process can grow). It is also called as "swap area" or "swap space".

- Shifting a process from RAM to swap area is called as "swap out" and shifting a process from swap to RAM is called as "swap in".
- In few OS, swap area is created in form of a partition. E.g. UNIX, Linux, ...
- In few OS, swap area is created in form of a file E.g. Windows (pagefile.sys), ...
- Virtual memory advantages:
 - Can execute more number of programs.
 - Can execute bigger sized programs.

Segmentation

- Instead of allocating contiguous memory for the whole process, contiguous memory for each segment can be allocated. This scheme is known as "segmentation".
- Since process does not need contiguous memory for entire process, external fragmentation will be reduced.
- In this scheme, PCB is associated with a segment table which contains base and limit (size) of each segment of the process.
- During context switch these values will be loaded into MMU segment table.
- CPU request virtual address in form of segment address and offset address.
- Based on segment address appropriate base-limit pair from MMU is used to calculate physical address as shown in diag.
- MMU also contains STBR register which contains address of current process's segment table in the RAM.

Demand Segmentation

- If virtual memory concept is used along with segmentation scheme, in case low memory, OS may swap out a segment of inactive process.
- When that process again starts executing and asks for same segment (swapped out), the segment will be loaded back in the RAM. This is called as "demand segmentation".
- Each entry of the segment table contains base & limit of a segment. It also contains additional bits like segment permissions, valid bit, dirty bit, etc
- If segment is present in main memory, its entry in seg table is said to be valid ($v=1$). If segment is swapped out, its entry in segment table is said to be invalid ($v=0$).

Paging

- RAM is divided into small equal sized partitions called as "frames" / "physical pages".
- Process is divided into small equal sized parts called as "pages" or "logical/virtual pages".
- page size = frame size.
- One page is allocated to one empty frame.
- OS keeps track of free frames in form of a linked list.
- Each PCB is associated with a table storing mapping of page address to frame address. This table is called as "page table".
- During context switch this table is loaded into MMU.
- CPU requests a virtual address in form of page address and offset address. It will be converted into physical address as shown in diag.
- MMU also contains a PTBR, which keeps address of page table in RAM.

- If a page is not utilizing entire frame allocated to it (i.e. page contents are less than frame size), then it is called as "internal fragmentation".
- Frame size can be configured in the hardware. It can be 1KB, 2KB or 4KB, ...
- Typical Linux and Windows OS use page size = 4KB.

Page table entry

- Each PTE is of 32-bit (on x86 arch) and it contains
 - Frame address
 - Permissions (read or write)
 - Validity bit
 - Dirty bit
 - ...

TLB (Translation Look-Aside Buffer) Cache

- TLB is high-speed associative cache memory used for address translation in paging MMU.
- TLB has limited entries (e.g. in P6 arch TLB has 32 entries) storing recently translated page address and frame address mappings.
- The page address given by CPU, will be compared at once with all the entries in TLB and corresponding frame address is found.
- If frame address is found (TLB hit), then it is used to calculate actual physical address in RAM (as shown in diag).
- If frame address is not found (TLB miss), then PTBR is used to access actual page table of the process in the RAM (associated with PCB). Then page-frame address mapping is copied into TLB and thus physical address is calculated.
- If CPU requests for the same page again, its address will be found in the TLB and translation will be faster

Two Level Paging

- Primary page table has number of entries and each entry point to the secondary page table page.
- Secondary page table has number of entries and each entry point to the frame allocated for the process.
- Virtual address requested by a process is 32 bits including
 - p1 (10 bits) -> Primary page table index/addr
 - p2 (10 bits) -> Secondary page table index/addr
 - d (12 bits) -> Frame offset

Demand Paging

- When virtual memory is used with paging memory management, pages can be swapped out in case of low memory.
- The pages will be loaded into main memory, when they are requested by the CPU. This is called as "demand paging".
 - Swapped out pages
 - Pages from program image (executable file on disk)
 - Dynamically allocated pages

Virtual pages vs Logical pages

- By default all pages of user space process can be swapped out/in. This may change physical address of the page. All such pages whose physical address may change are referred as "Virtual pages".
- Few kernel pages are never swapped out. So their physical address remains same forever. All such pages whose physical address will not change are referred as "Logical pages".

Thrashing

- If number of programs are running in comparatively smaller RAM, a lot of system time will be spent into page swapping (paging) activity.
- Due to this overall system performance is reduced.
- The problem can be solved by increasing RAM size in the machine.

Page Fault

- Each page table entry contains frame address, permissions, dirty bit, valid bit, etc.
- If page is present in main memory its page table entry is valid (valid bit = 1).
- If page is not present in main memory, its page table entry is not valid (valid bit = 0).
- This is possible due to one of the following reasons:
 - Page address is not valid (dangling pointer).
 - Page is on disk/swapped out.
 - Page is not yet allocated.
- If CPU requests a page that is not present in main memory (i.e. page table entry valid bit=0), then "page fault" occurs.
- Then OS's page fault exception handler is invoked, which handles page faults as follows:
 1. Check virtual address due to which page fault occurred. If it is not valid (i.e. dangling pointer), terminate the process (sending SEGV signal). (Validity fault).
 2. Check if read-write operation is permitted on the address. If not, terminate the process (sending SEGV signal). (Protection fault).
 3. If virtual address is valid (i.e. page is swapped out), then locate one empty frame in the RAM.
 4. If page is on swap device or hard disk, swap in the page in that frame.
 5. Update page table entry i.e. add new frame address and valid bit = 1 into PTE.
 6. Restart the instruction for which page fault occurred.

Page Replacement Algorithms

- While handling page fault if no empty frame found (step 3), then some page of any process need to be swapped out. This page is called as "victim" page.
- The algorithm used to decide the victim page is called as "page replacement algorithm".
- There are three important page replacement algorithms.
 - FIFO
 - Optimal
 - LRU

FIFO

- The page brought in memory first, will be swapped out first.

- Sometimes in this algorithm, if number of frames are increased, number of page faults also increase.
- This abnormal behaviour is called as "Belady's Anomaly".

OPTIMAL

- The page not required in near future is swapped out.
- This algorithm gives minimum number of page faults.
- This algorithm is not practically implementable.

LRU

- The page which not used for longer duration will be swapped out.
- This algorithm is used in most OS like Linux, Windows, ...
- LRU mechanism is implemented using "stack based approach" or "counter based approach".
- This makes algorithm implementation slower.
- Approximate LRU algorithm close to LRU, however is much faster.

Dirty Bit

- Each entry in page table has a dirty bit.
- When page is swapped in, dirty bit is set to 0.
- When write operation is performed on any page, its dirty bit is set to 1. It indicate that copy of the page in RAM differ from the copy in swap area.
- When such page need to be swapped out again, OS check its dirty bit. If bit=0 (page is not modified) actual disk IO is skipped and improves performance of paging operation.
- If bit=1 (page is modified), page is physically overwritten on its older copy in the swap area.

Process Creation

- System Calls
 - Windows: CreateProcess()
 - UNIX: fork()
 - BSD UNIX: fork(), vfork()
 - Linux: clone(), fork(), vfork()

fork() syscall

- To execute certain task concurrently we can create a new process (using fork() on UNIX).
- fork() creates a new process by duplicating calling process.
- The new process is called as "child process", while calling process is called as "parent process".
- "child" process is exact duplicate of the "parent" process except few points pid, parent pid, etc.
- pid = fork();
 - On success, fork() returns pid of the child to the parent process and 0 to the child process.
 - On failure, fork() returns -1 to the parent.
- Even if child is copy of the parent process, after its creation it is independent of parent and both these processes will be scheduled separately by the scheduler.
- Based on CPU time given for each process, both processes will execute concurrently.

How fork() return two values i.e. in parent and in child?

- fork() creates new process by duplicating calling process.
- The child process PCB & kernel stack is also copied from parent process. So child process has copy of execution context of the parent.
- Now fork() write 0 in execution context (r0 register) of child process and child's pid into execution context (r0 register) of parent process.
- When each process is scheduled, the execution context will be restored (by dispatcher) and r0 is return value of the function.

getpid() vs getppid()

- pid1 = getpid(); // returns pid of the current process
- pid2 = getppid(); // returns pid of the parent of the current process

When fork() will fail?

- When no new PCB can be allocated, then fork() will fail.
- Linux has max process limit for the system and the user. When try to create more processes, fork() fails.
- terminal> cat /proc/sys/kernel/pid_max

Orphan process

- If parent of any process is terminated, that child process is known as orphan process.
- The ownership of such orphan process will be taken by "init" process.

Zombie process

- If process is terminated before its parent process and parent process is not reading its exit status, then even if process's memory/resources is released, its PCB will be maintained. This state is known as "zombie state".
- To avoid zombie state parent process should read exit status of the child process. It can be done using wait() syscall.

wait() syscall

- ret = wait(&s);
 - arg1: out param to get exit code of the process.
 - returns: pid of the child process whose exit code is collected.
- wait() performs 3 steps:
 - Pause execution parent until child process is terminated.
 - Read exit code from PCB of child process & return to parent process (via out param).
 - Release PCB of the child process.
- The exit status returned by the wait() contains exit status, reason of termination and other details.
- Few macros are provided to access details from the exit code.
 - WEXITSTATUS()

waitpid() syscall

- This extended version of wait() in Linux.
- ret = waitpid(child_pid, &s, flags);
 - arg1: pid of the child for which parent should wait.
 - -1 means any child.
 - arg2: out param to get exit code of the process.
 - arg3: extra flags to define behaviour of waitpid().
- returns: pid of the child process whose exit code is collected.
 - -1: if error occurred.

exec() syscall

- exec() syscall "loads a new program" in the calling process's memory (address space) and replaces the older (calling) one.
- If exec() succeed, it does not return (rather new program is executed).
- There are multiple functions in the family of exec():
 - execl(), execp(), execle(),
 - execv(), execvp(), execve(), execvpe()
- exec() family multiple functions have different syntaxes but same functionality.

Synchronization

- Multiple processes accessing same resource at the same time, is known as "race condition".
- When race condition occurs, resource may get corrupted (unexpected results).
- Peterson's problem, if two processes are trying to modify same variable at the same time, it can produce unexpected results.
- Code block to be executed by only one process at a time is referred as Critical section. If multiple processes execute the same code concurrently it may produce undesired results.
- To resolve race condition problem, one process can access resource at a time. This can be done using sync objects/primitives given by OS.
- OS Synchronization objects are:
 - Semaphore, Mutex

Semaphore

- Semaphore is a sync primitive given by OS.
- Internally semaphore is a counter. On semaphore two operations are supported:
 - wait operation: dec op: P operation:
 - semaphore count is decremented by 1.
 - if cnt < 0, then calling process is blocked.
 - typically wait operation is performed before accessing the resource.
 - signal operation: inc op: V operation:
 - semaphore count is incremented by 1.
 - if one or more processes are blocked on the semaphore, then one of the process will be resumed.
 - typically signal operation is performed after releasing the resource.
- Semaphore types
 - Counting Semaphore
 - Allow "n" number of processes to access resource at a time.

- Or allow "n" resources to be allocated to the process.
- Binary Semaphore
 - Allows only 1 process to access resource at a time or used as a flag/condition.

Mutex

- Mutex is used to ensure that only one process can access the resource at a time.
- Functionally it is same as "binary semaphore".
- Mutex can be unlocked by the same process/thread, which had locked it.

Semaphore vs Mutex

- S: Semaphore can be decremented by one process and incremented by same or another process.
- M: The process locking the mutex is owner of it. Only owner can unlock that mutex.
- S: Semaphore can be counting or binary.
- M: Mutex is like binary semaphore. Only two states: locked and unlocked.
- S: Semaphore can be used for counting, mutual exclusion or as a flag.
- M: Mutex can be used only for mutual exclusion.

Deadlock

- Deadlock occurs when four conditions/characteristics hold true at the same time.
 - No preemption: A resource should not be released until task is completed.
 - Mutual exclusion: Resources is not sharable.
 - Hold & Wait: Process holds a resource and wait for another resource.
 - Circular wait: Process P1 holds a resource needed for P2, P2 holds a resource needed for P3 and P3 holds a resource needed for P1.

Deadlock Prevention

- OS syscalls are designed so that at least one deadlock condition does not hold true.
- In UNIX multiple semaphore operations can be done at the same time.

Deadlock Avoidance

- Processes declare the required resources in advanced, based on which OS decides whether resource should be given to the process or not.
- Algorithms used for this are:
 - Resource allocation graph: OS maintains graph of resources and processes. A cycle in graph indicate circular wait will occur. In this case OS can deny a resource to a process.
 - Banker's algorithm: A bank always manage its cash so that they can satisfy all customers.
 - Safe state algorithm: OS maintains statistics of number of resources and number processes. Based on stats it decides whether giving resource to a process is safe or not (using a formula):
 - Max num of resources required < Num of resources + Num of processes
 - If condition is true, deadlock will never occur.
 - If condition is false, deadlock may occur



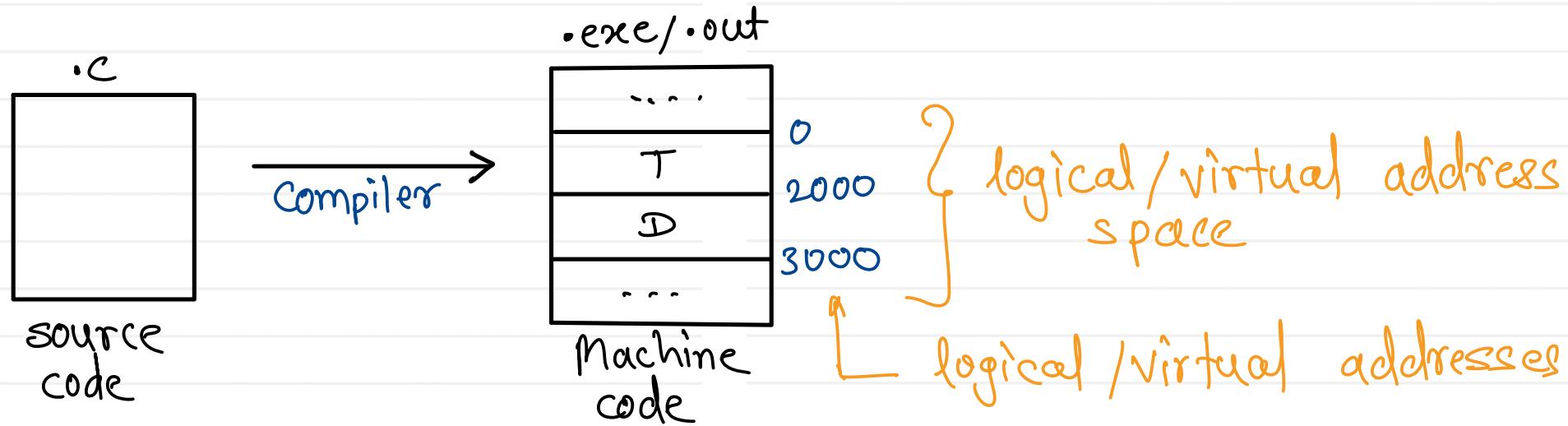
**Sunbeam Institute of Information Technology
Pune and Karad**

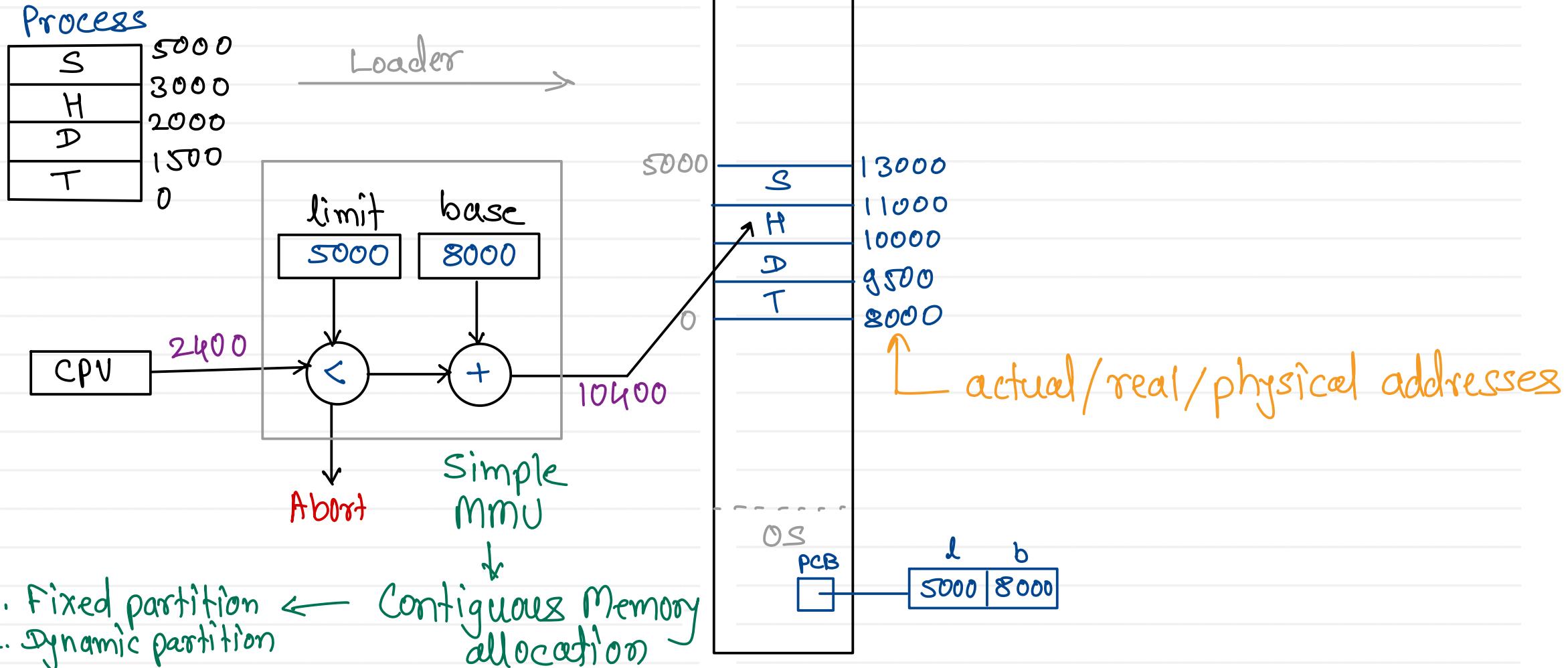
Module - Concepts of Operating System

Trainer - Devendra Dhande
Email – devendra.dhande@sunbeaminfo.com

Memory Management

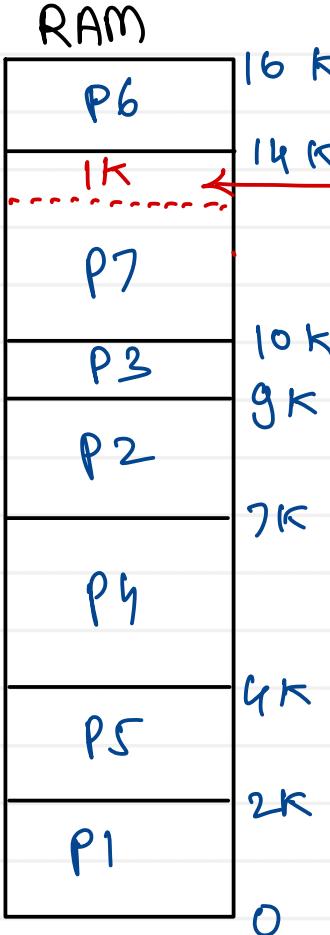
- compiler always assigns logical / virtual / imaginary addresses to the functions or variables





Contiguous memory allocation

Fixed partition

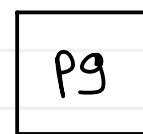


internal fragmentation
if process is not utilizing whole allocated partition, then some part will be wasted.

limitations :

1. Max no. of processes are no. of partitions
2. max process size is equal to max partition size

Dynamic Partition



gSD
4000

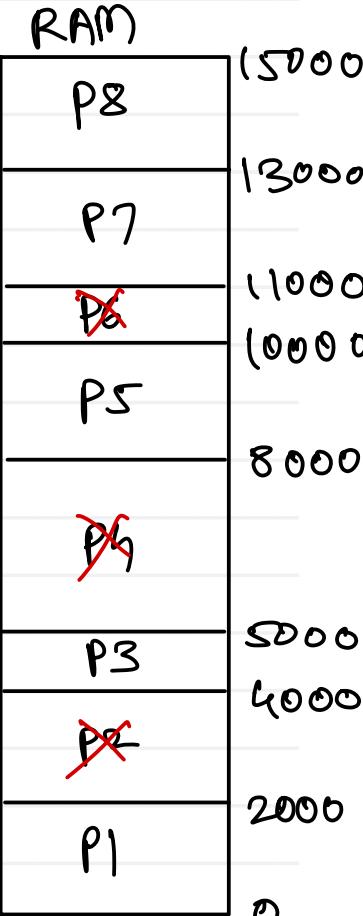
- first fit
- best fit
- worst fit

limit	base
2000	2000
1000	10000
3000	5000

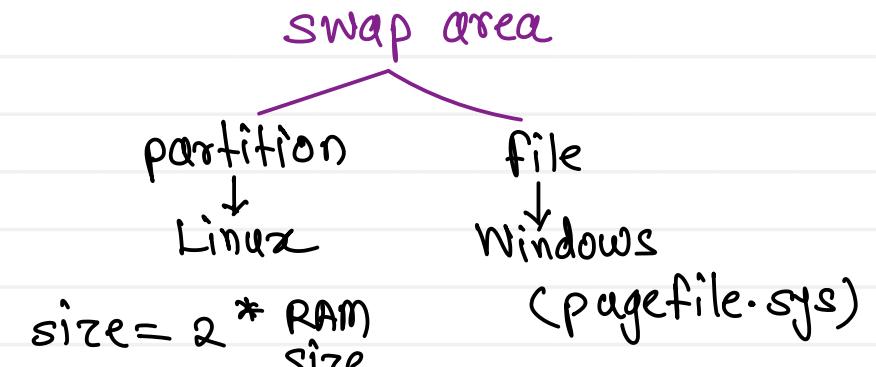
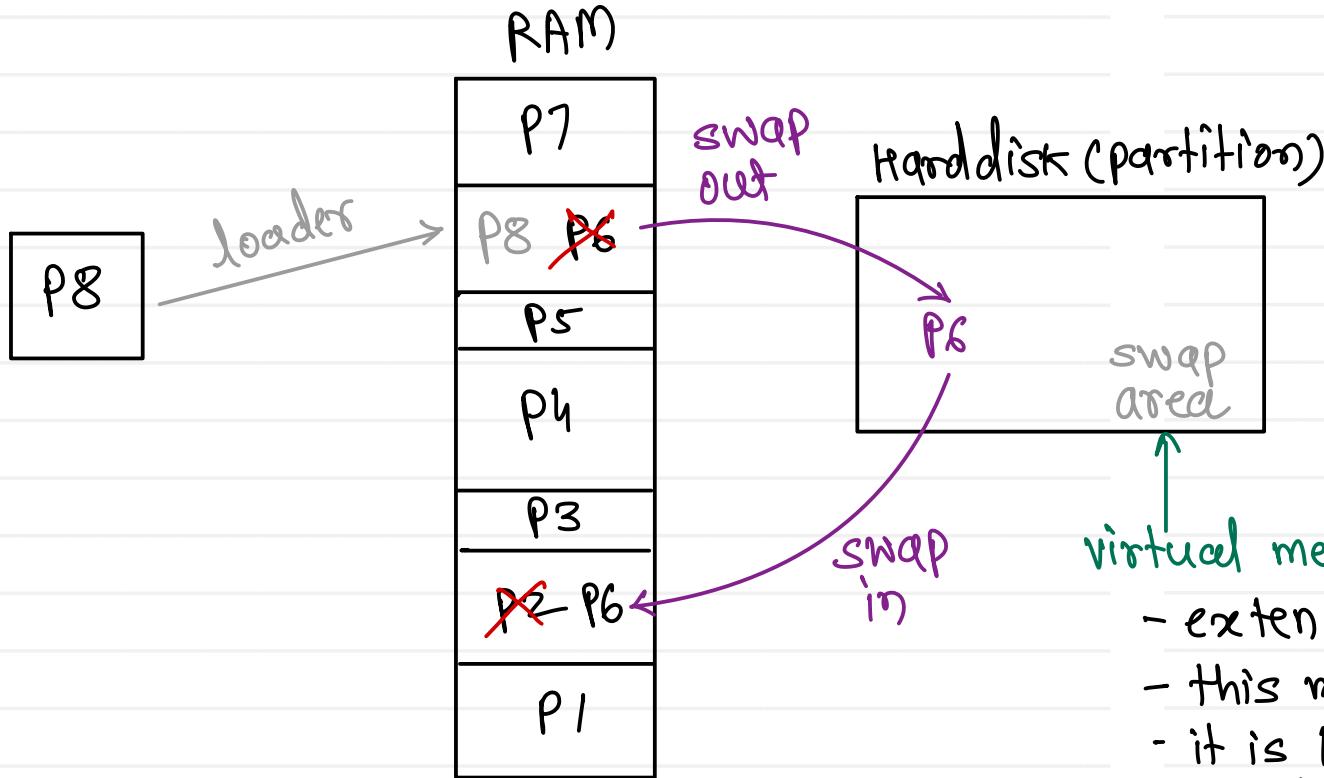
external fragmentation
if large contiguous free space is not available, we can not load process inside RAM

Compaction:

moving processes inside RAM to create large free space



Virtual memory



virtual memory

- extension to RAM (memory)
- this memory is formatted like RAM
- it is like RAM but not RAM that's why it is called as virtual memory.
- processes are only kept into this area (inactive)
- processes never executes inside virtual memory.



Segmentation MMU

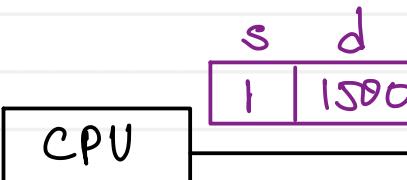
Process

1	S	5000
2	H	3000
3	D	2000
4	T	1500
		0

loader

Segment Table

	limit	base
1	2000	12000
2	1000	10000
3	500	8000
4	1500	15000



Abort

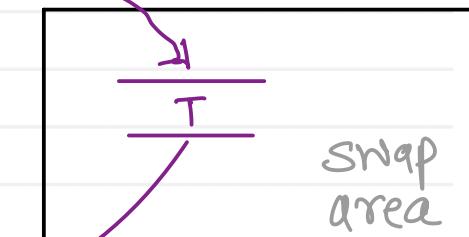
Segmentation
MMU

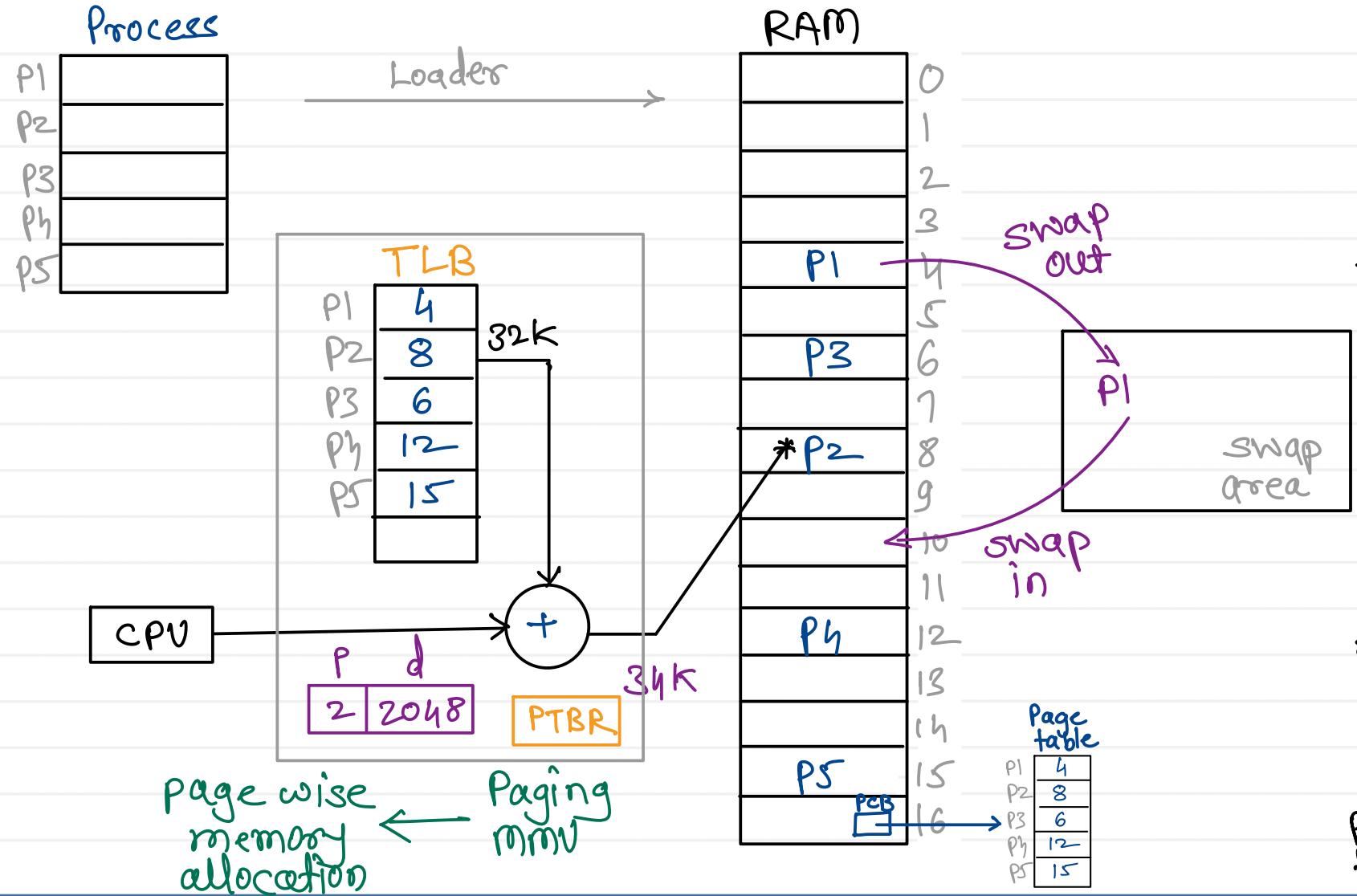
segment wise memory allocation

:	
T	
:	
*	S
H	
D	
:	
!	

limit	base
2000	12000
1000	10000
500	8000
1500	15000

Demand Segmentation
on demand segment is
swapped in.

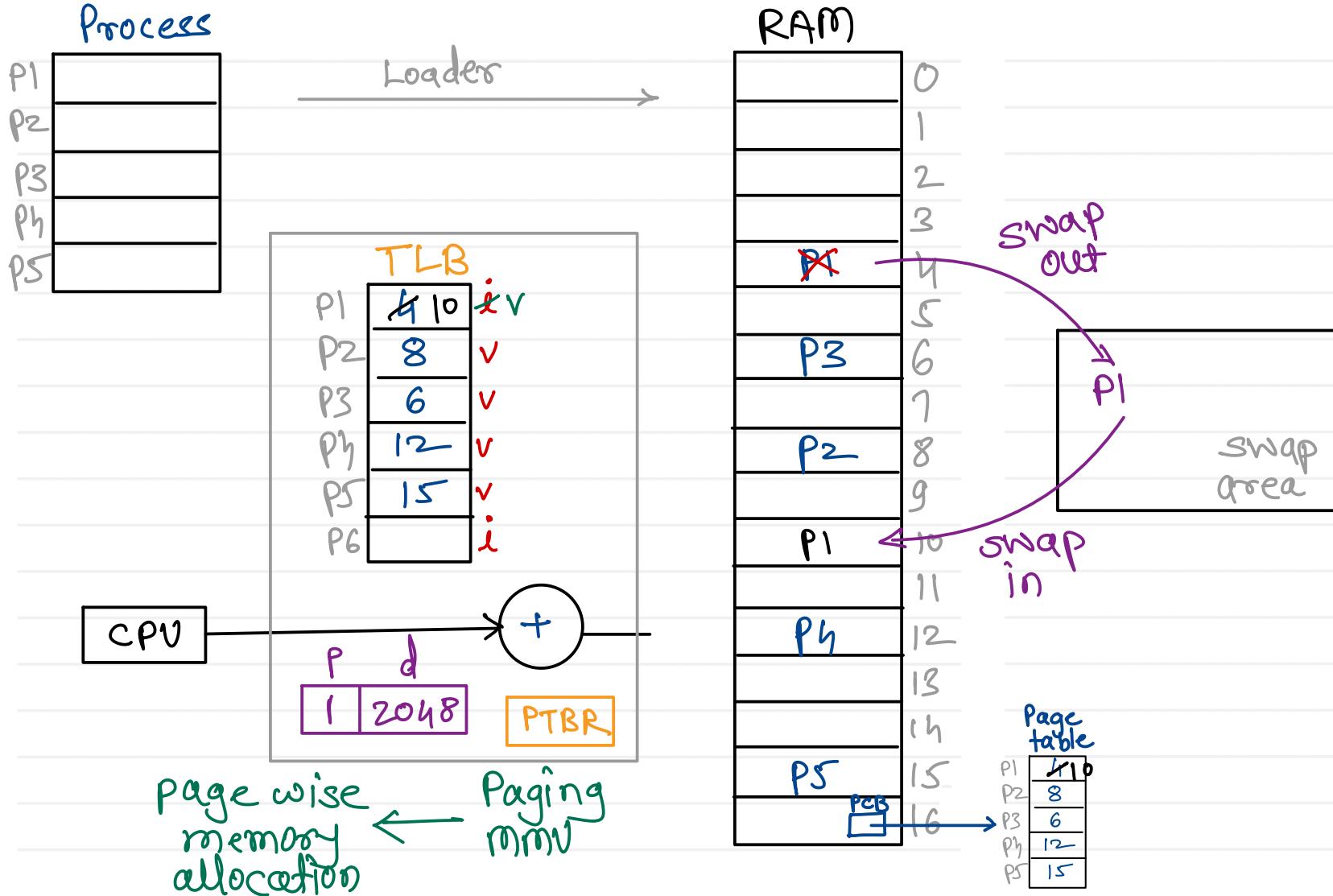




- RAM is divided into fixed equal size partitions
size = 4 kb (4096 bytes)
"frame"/"physical page"
- Process is also divided into partitions of size equal to frame size.
"page"/"logical page"

Demand paging:
on demand page is swapped in

Page fault



Page fault :
 Whenever CPU request for the address of some invalid entry of page table, this fault is generated.
 On every page fault, page fault handler of OS is called

pagefault_handler() :

1. validate the address
2. check for read/write perm
3. find free frame in RAM
4. swap in page into free frame
5. update mapping in page table and TLB.
6. re execute the command for which page was occurred

Thrashing: frequent swap in & swap out of pages
Solution: increase the size

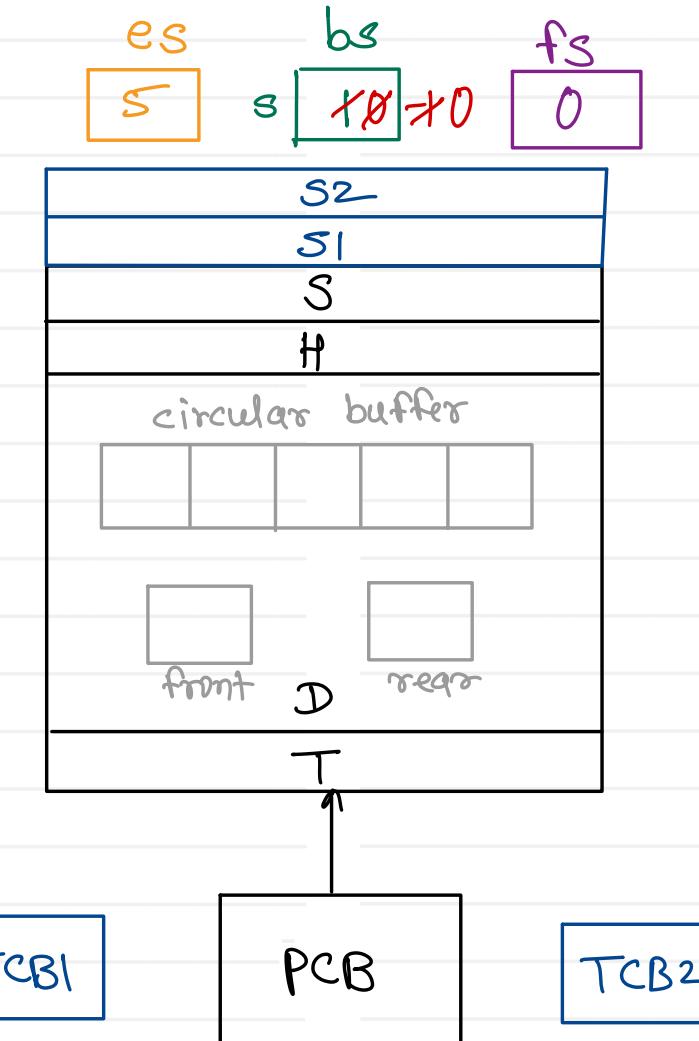
Producer - Consumer

Producers
thread1() {

```
while(1) {
    p(s)
    p(es)
    buf[rear] = ↓;
    v(s)
    v(fs)
}
```

}

}



Consumers
thread2() {

```
while(1) {
    p(s)
    p(fs)
    ▷ = buf[front];
    v(s)
    v(es)
}
```

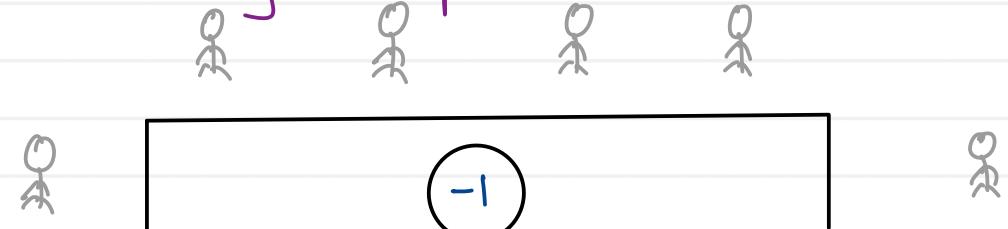
}

}

Semaphore

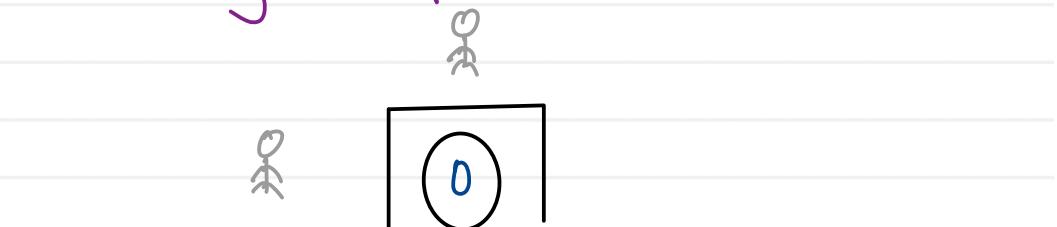
- semaphore is internally a counter
- Operations:
 1. Dec / wait / P():
 - a. dec count
 - b. if count < 0, then block the current process
 2. Inc / post / V():
 - a. inc count
 - b. if someone is blocked on this semaphore, wake up one

a. Counting semaphore



- counting no. of resources available
- counting no. of processes waiting

b. Binary semaphore



- only one should use resource at a time.



Mutex

Mutex = Mutual Exclusion

↳ one at a time

- lock/unlock operations are performed on mutex
- process who locks the mutex becomes owner of the mutex
- only owner can unlock the mutex



- infinite waiting for a resource
- deadlock occurs only when below four conditions hold true at a time

1. Mutual Exclusion
2. No preemption
3. Hold & wait
4. Circular wait



Prevention :

while implementing OS, it is always ensured that 1/4 condition will hold false.

Avoidance :

1. Banker's algorithm
2. Resource allocation graph
3. Safe state algorithm

Recover :

1. resource preemption
2. forceful termination of process



Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com

FIFO Page Replacement

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
2	2	2	2	1	1	1	1	1	3	3	3
3	3	3	3	3	2	2	2	2	2	4	4

Number of Page Fault = 9

Belady's Anomaly

Optimal Page Replacement

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	3	3	3
2	2	2	2	2	2	2	2	2	2	4	4
3	3	3	4	4	4	5	5	5	5	5	5

Number of Page Fault = 7

LRU Page Replacement

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	3	3	3
2	2	2	2	1	1	1	1	1	1	4	4
3	3	3	3	3	2	2	2	2	2	2	5

Number of Page Fault = 10



DevOps

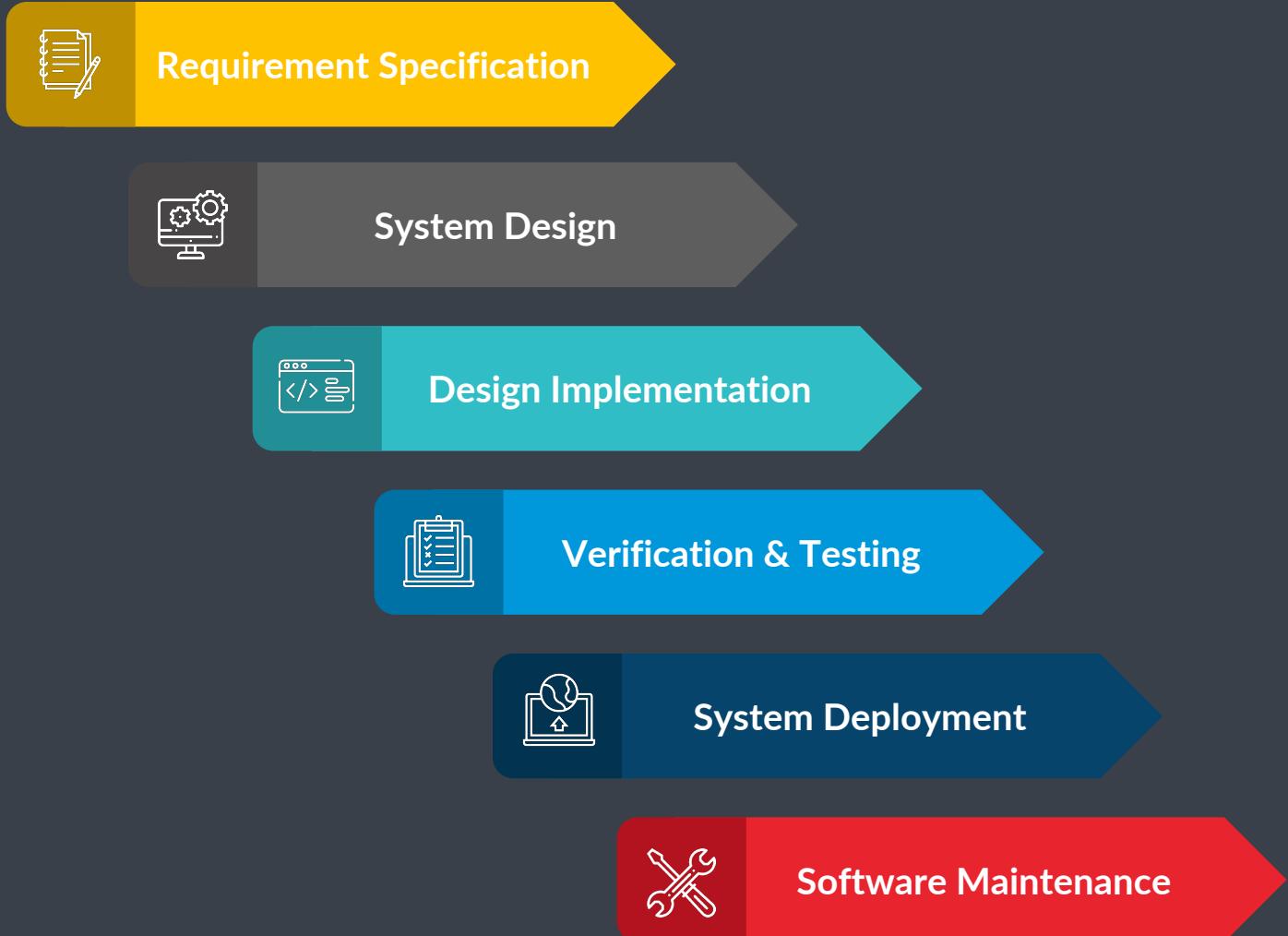


Software Development Lifecycle





Waterfall Model

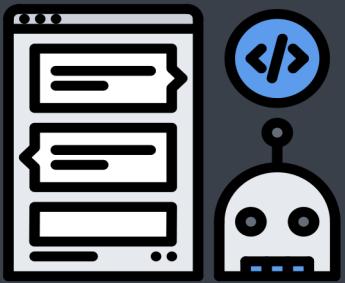




Entities involved



Developer



Testers



Operations Team



Responsibilities



Developers and Testers

- Developers
 - Develop the application
 - Package the application
 - Fix the bugs
 - Maintain the application
- Testers
 - Thoroughly test the application manually or using test automation
 - Report the bugs to the developer

Operations Team

- Make all the necessary resources ready
- Deploy the application
- Maintain multiple environments
- Continuously monitor the application
- Manage the resources





Challenges

Developers and Testers

- The process is slow
- The pressure to work on the newer features and fix the older code
- Not flexible



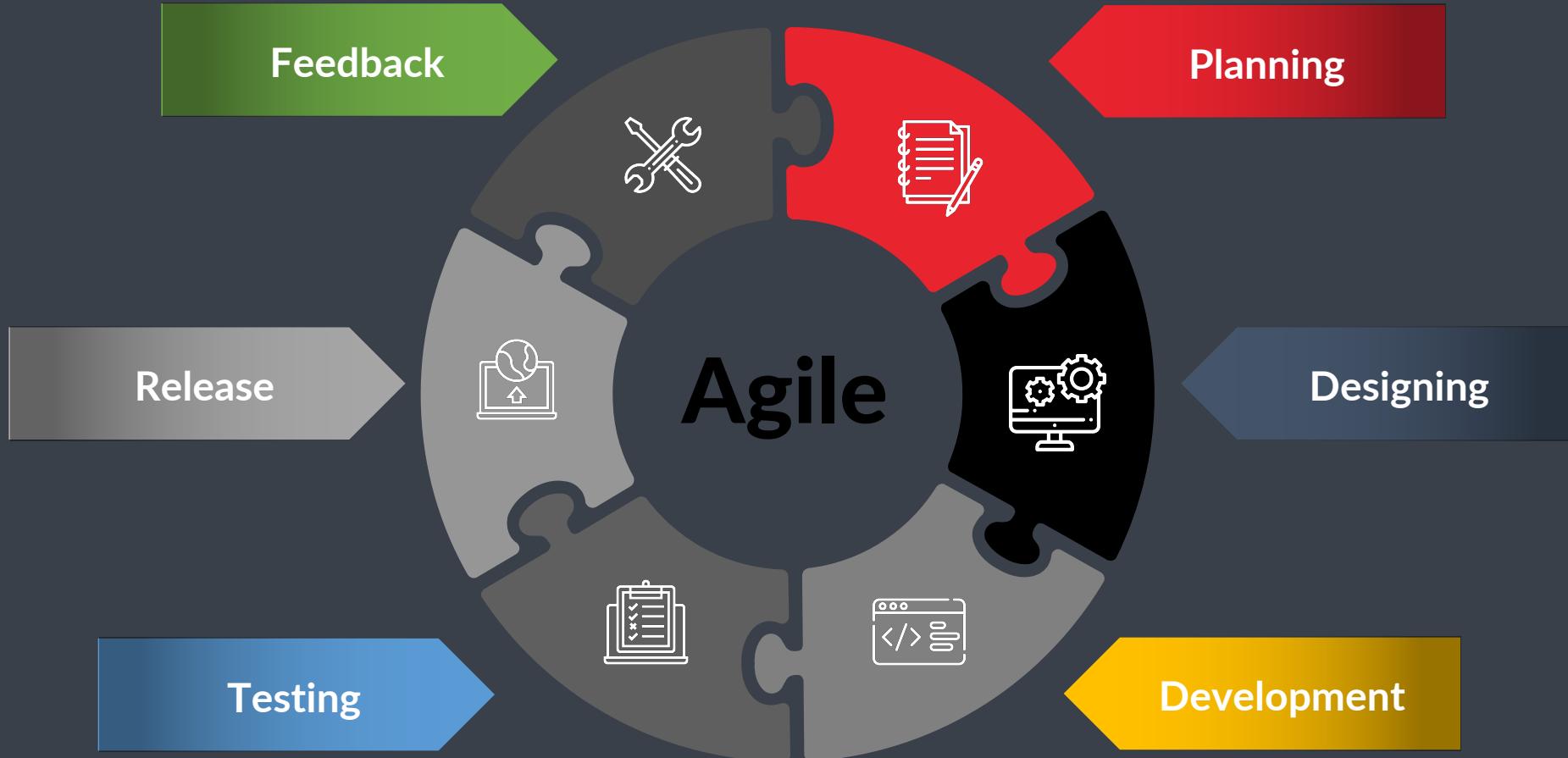
Operations Team

- Uptime
- Configure the huge infrastructure
- Diagnose and fix the issue



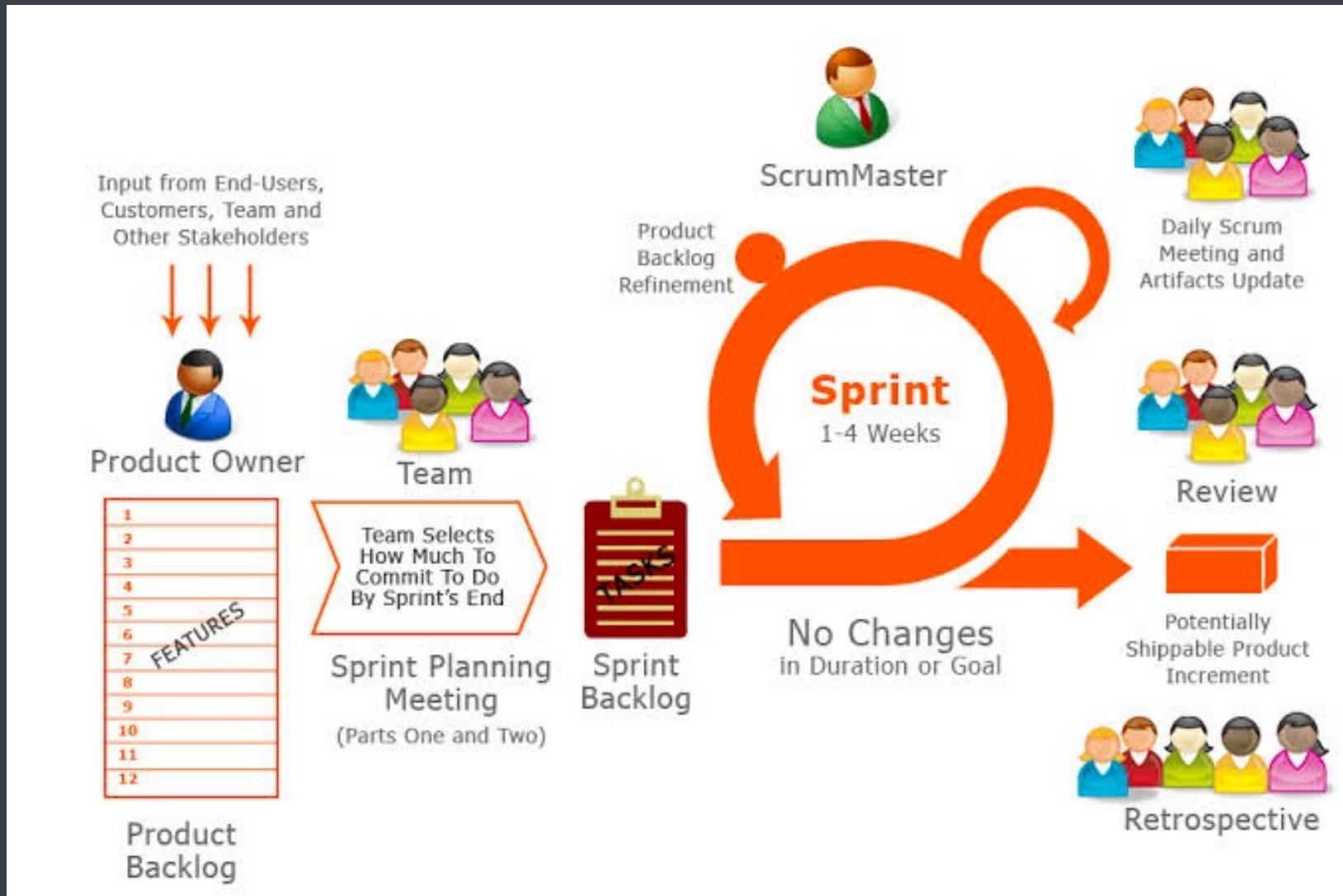


Agile Development





Scrum Process



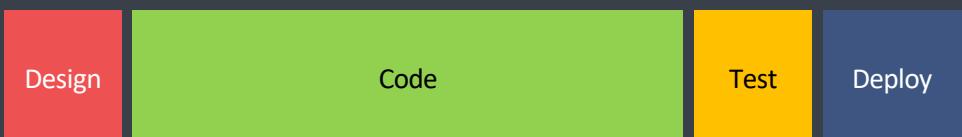
Waterfall Vs Agile



The Waterfall Process



This project has got so big.
I am not sure I will be able to deliver it!



The Agile Process



It is so much better delivering
this project in bite-sized sections





Problems

- Managing and tracking changes in the code is difficult
- Incremental builds are difficult to manage, test and deploy
- Manual testing and deployment of various components/modules takes a lot of time
- Ensuring consistency, adaptability and scalability across environments is very difficult task
- Environment dependencies makes the project behave differently in different environments



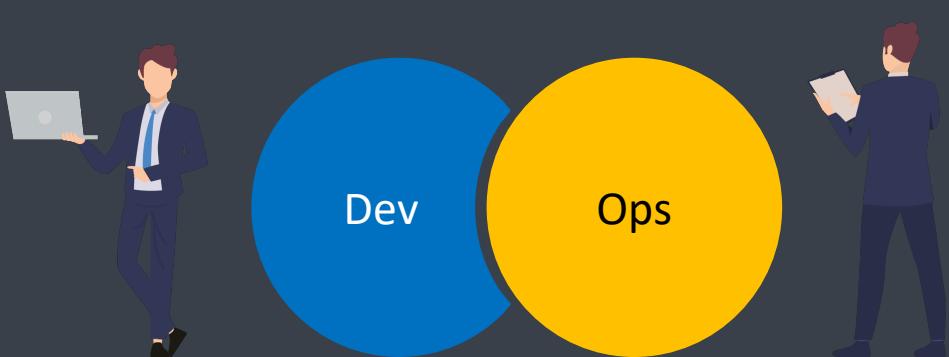
Solutions to the problem

- Managing and tracking changes in the code is difficult: **SCM tools**
- Incremental builds are difficult to manage, test and deploy: **Jenkins**
- Manual testing and deployment of various components/modules takes a lot of time: **Selenium**
- Ensuring consistency, adaptability and scalability across environments is very difficult task: **Puppet**
- Environment dependencies makes the project behave differently in different environments: **Docker**



What is DevOps ?

- DevOps is a combination of two words development and operations
- Promotes collaboration between Development and Operations Team to deploy code to production faster in an automated & repeatable way
- DevOps helps to increases an organization's speed to deliver applications and services
- It allows organizations to serve their customers better and compete more strongly in the market
- Can be defined as an alignment of development and IT operations with better communication and collaboration
- DevOps is not a goal but a never-ending process of continuous improvement
- It integrates Development and Operations teams
- It improves collaboration and productivity by
 - Automating infrastructure
 - Automating workflow
 - Continuously measuring application performance





Why DevOps is Needed?

- Before DevOps, the development and operation team worked in complete isolation
- Testing and Deployment were isolated activities done after design-build. Hence they consumed more time than actual build cycles.
- Without using DevOps, team members are spending a large amount of their time in testing, deploying, and designing instead of building the project.
- Manual code deployment leads to human errors in production
- Coding & operation teams have their separate timelines and are not in sync causing further delays



Common misunderstanding

- DevOps is not a role, person or organization
- DevOps is not a separate team
- DevOps is not a product or a tool
- DevOps is not just writing scripts or implementing tools

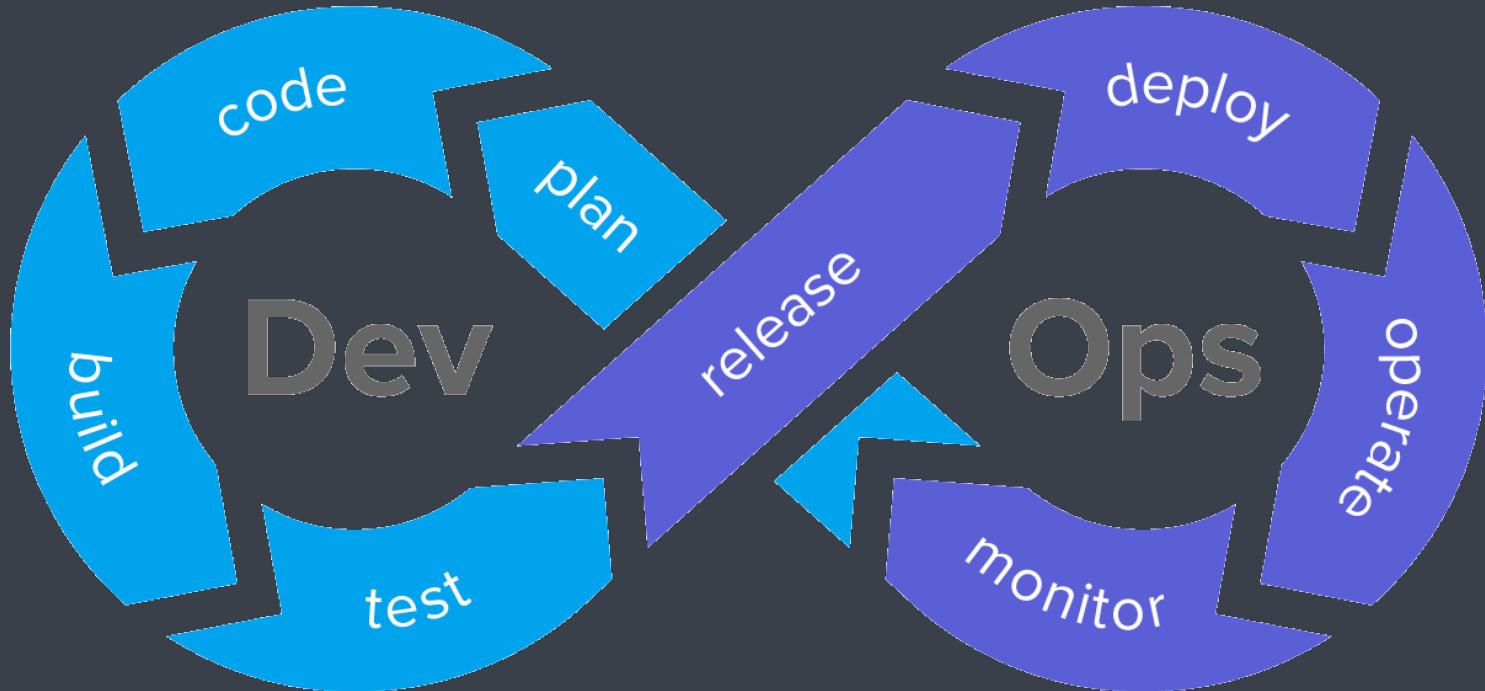


Reasons to use DevOps

- **Predictability**
 - DevOps offers significantly lower failure rate of new releases
- **Reproducibility**
 - Version everything so that earlier version can be restored anytime
- **Maintainability**
 - Effortless process of recovery in the event of a new release crashing or disabling the current system
- **Time to market**
 - DevOps reduces the time to market up to 50% through streamlined software delivery
 - This is particularly the case for digital and mobile applications
- **Greater Quality**
 - DevOps helps the team to provide improved quality of application development as it incorporates infrastructure issues
- **Reduced Risk**
 - DevOps incorporates security aspects in the software delivery lifecycle. It helps in reduction of defects across the lifecycle
- **Resiliency**
 - The Operational state of the software system is more stable, secure, and changes are auditable



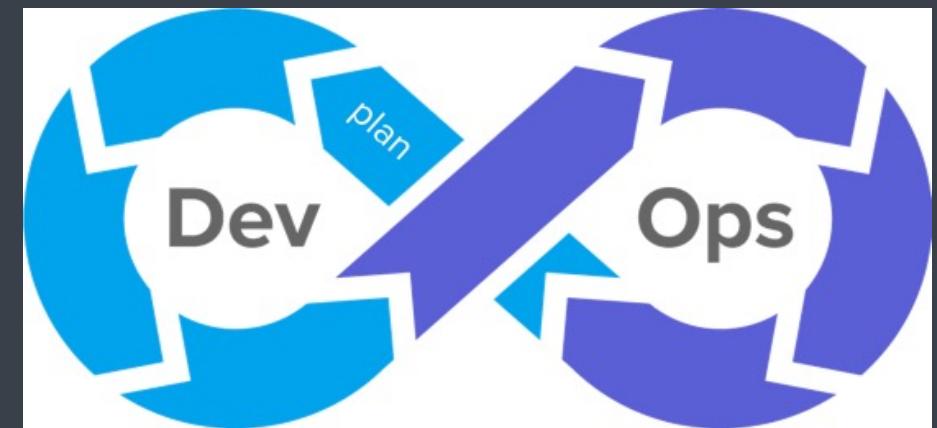
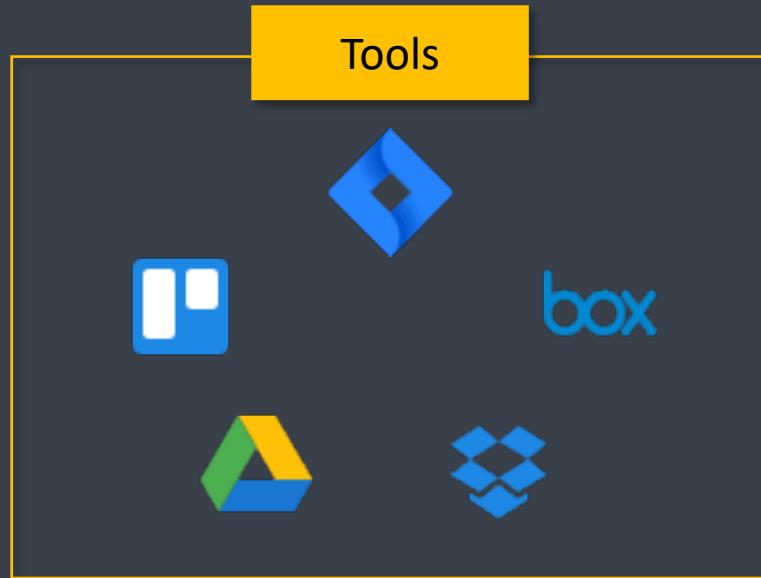
DevOps Lifecycle





DevOps Lifecycle - Plan

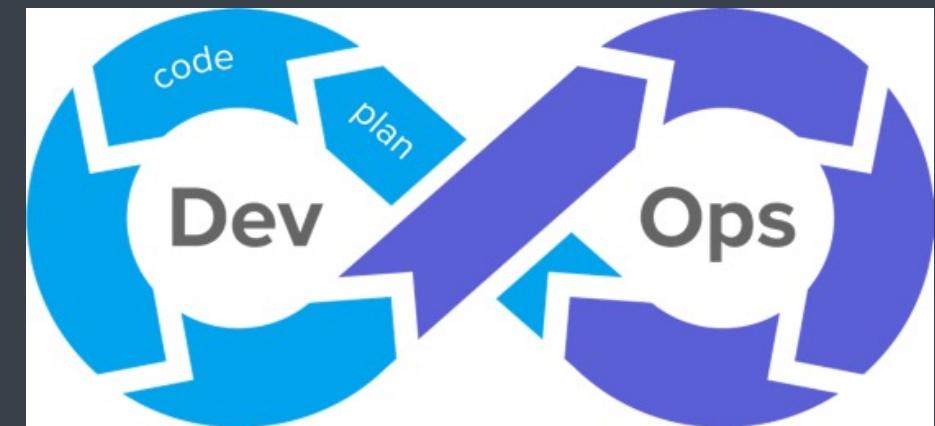
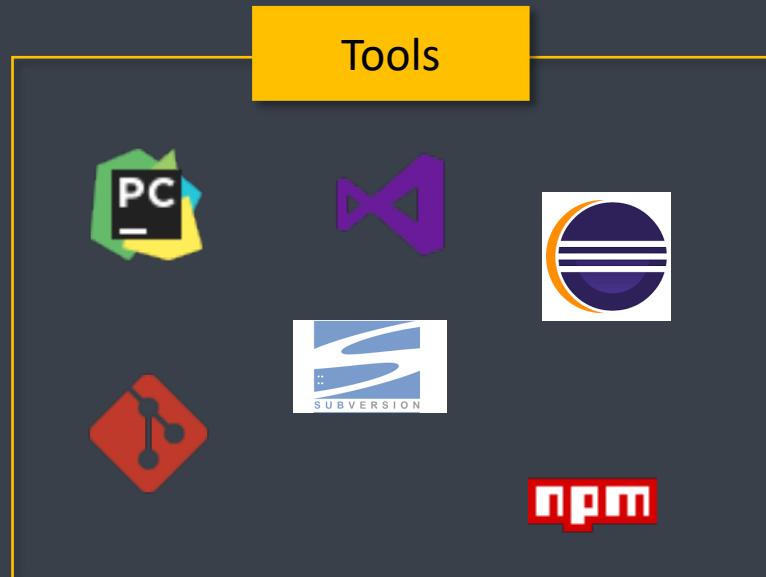
- First stage of DevOps lifecycle where you plan, track, visualize and summarize your project before you start working on it





DevOps Lifecycle - Code

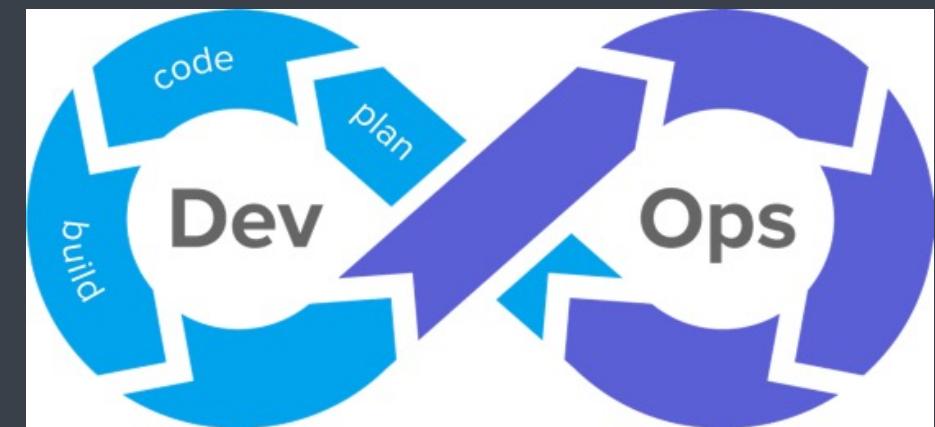
- Second stage where developer writes the code using favorite programming language





DevOps Lifecycle -Build

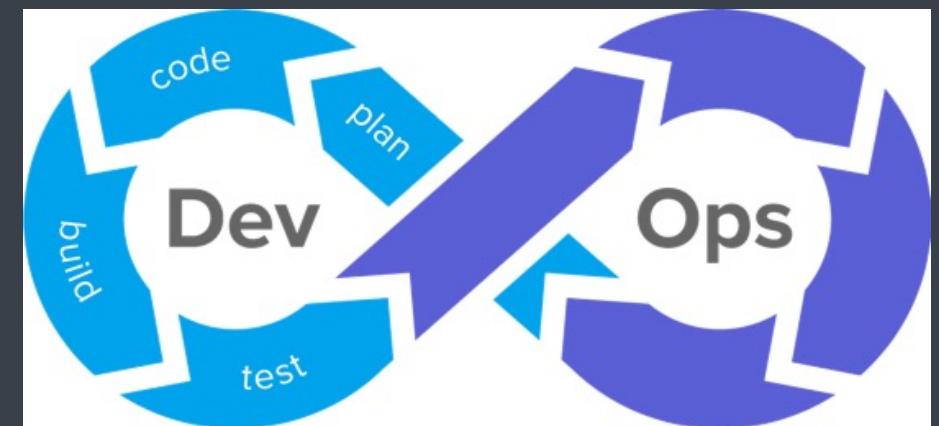
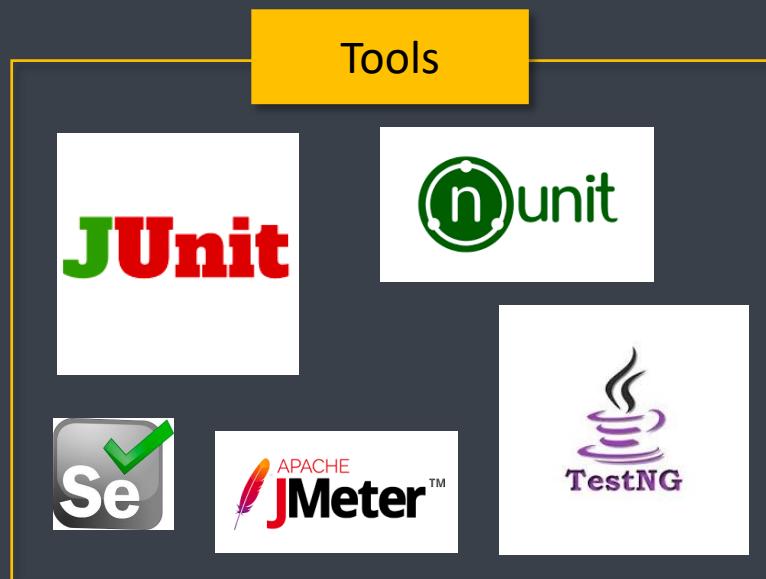
- Integrating the required libraries
- Compiling the source code
- Create deployable packages





DevOps Lifecycle - Test

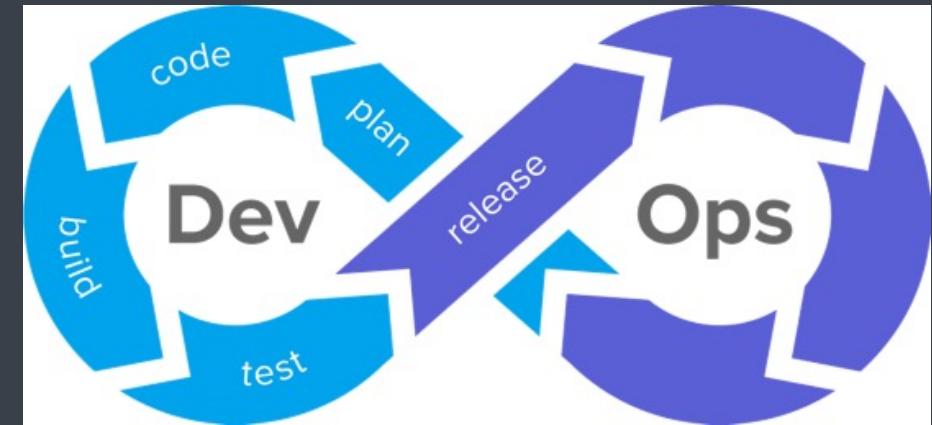
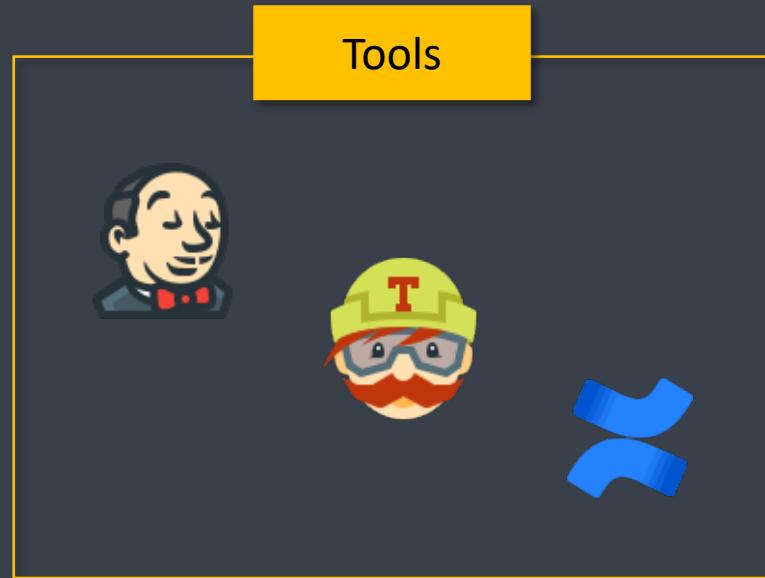
- Process of executing automated tests
- The goal here is to get the feedback about the changes as quickly as possible





DevOps Lifecycle - Release

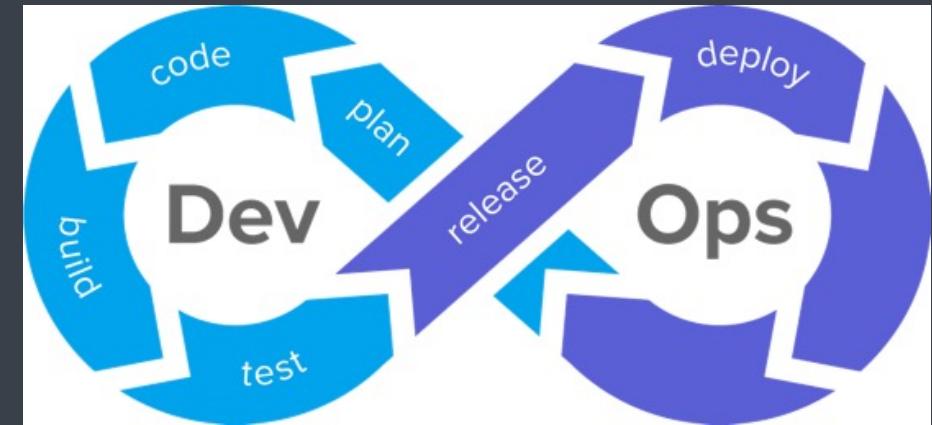
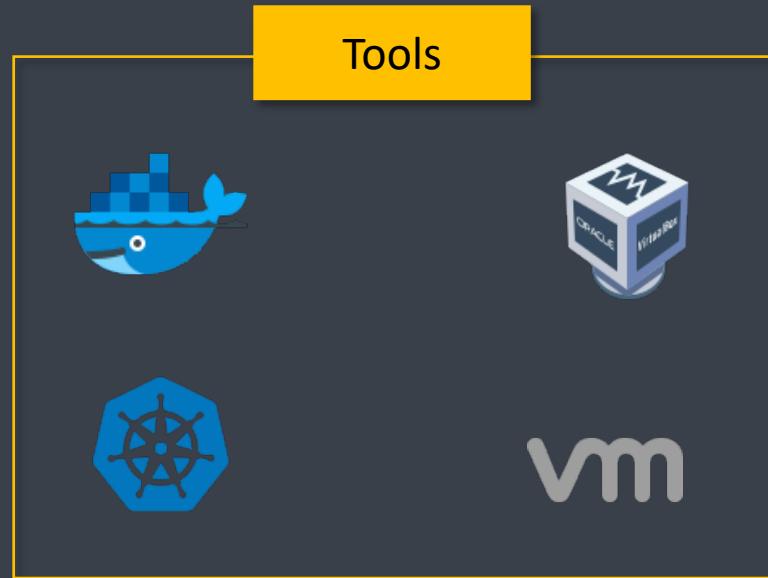
- This phase helps to integrate code into a shared repository using which you can detect and locate errors quickly and easily





DevOps Lifecycle - Deploy

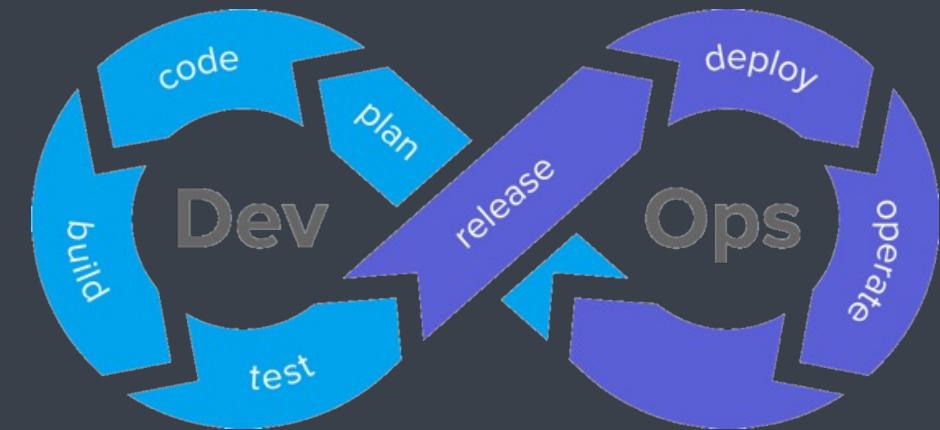
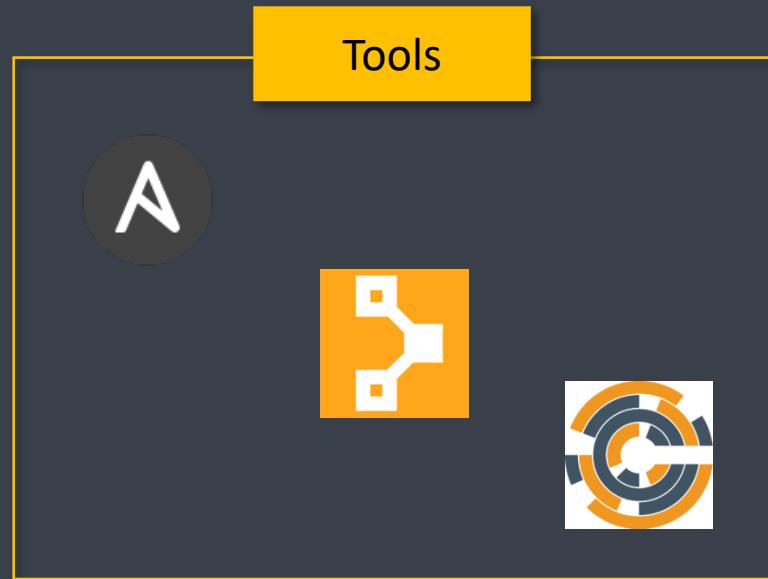
- Manage and maintain development and deployment of software systems and server in any computational environment





DevOps Lifecycle - Operate

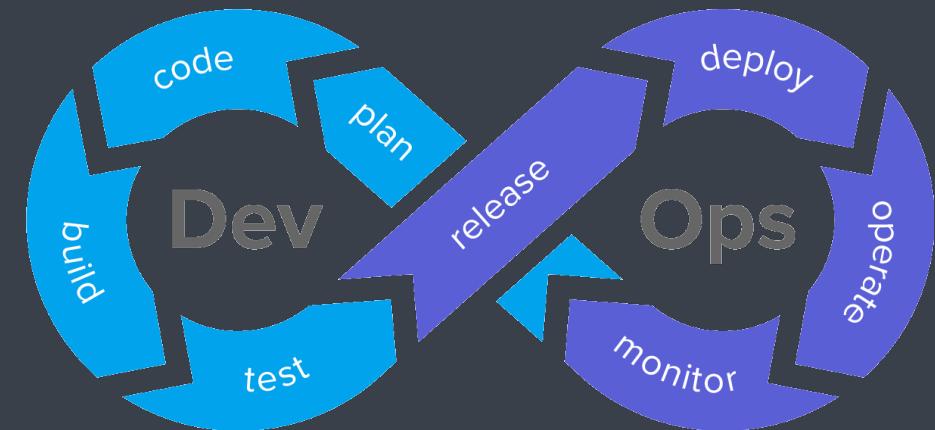
- This stage where the updated system gets operated





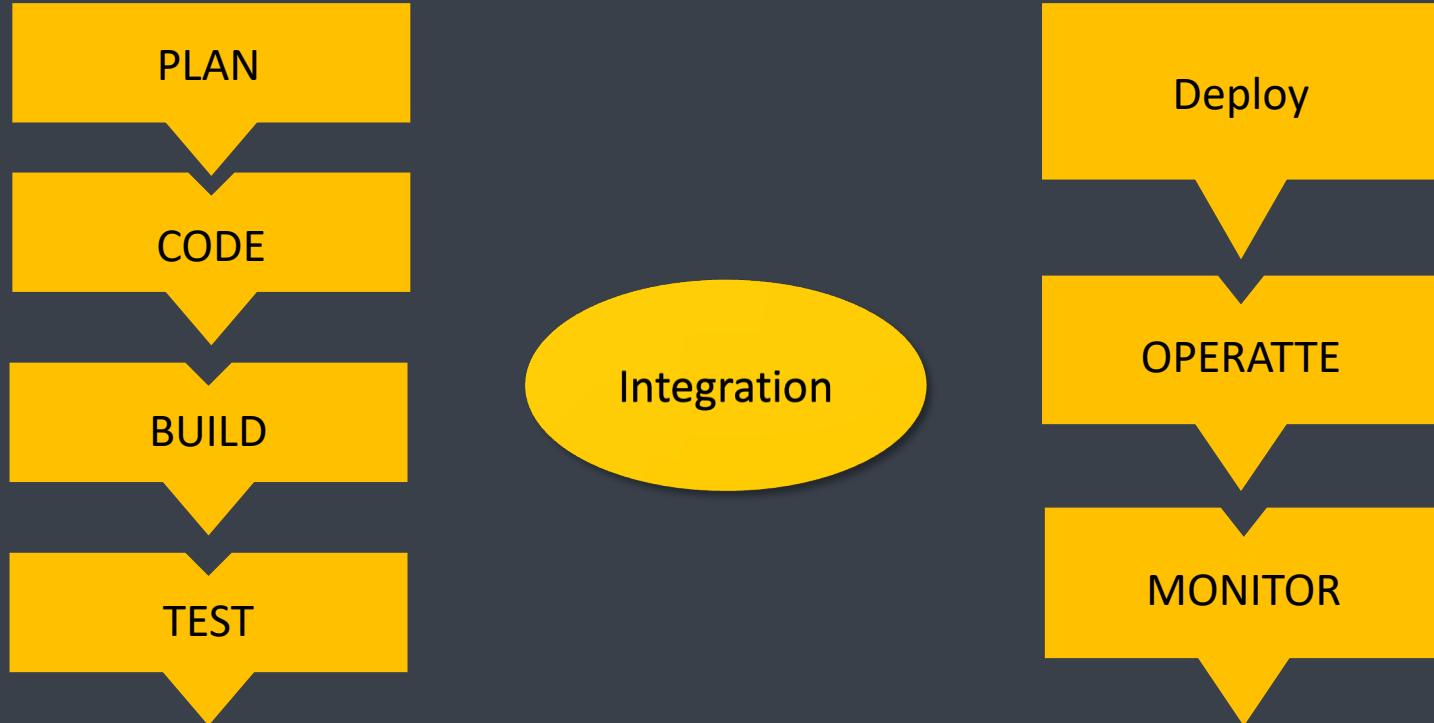
DevOps Lifecycle - Monitor

- It ensures that the application is performing as expected and the environment is stable
- It quickly determines when a service is unavailable and understand the underlying causes



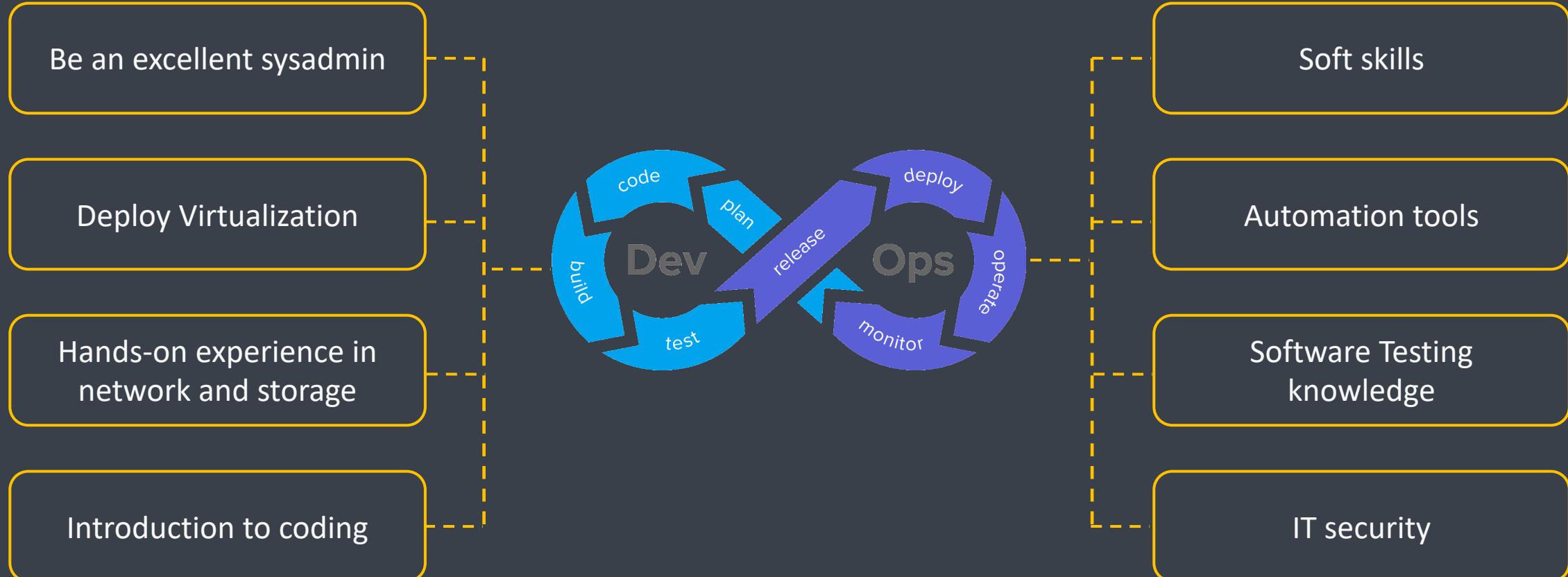


DevOps Terminologies





Responsibilities of DevOps Engineer



Skills of a DevOps Engineer



Skills	Description
Tools	<ul style="list-style-type: none">• Version Control – Git/SVN• Continuous Integration – Jenkins• Virtualization / Containerization – Docker/Kubernetes• Configuration Management – Puppet/Chef/Ansible• Monitoring – Nagios/Splunk
Network Skills	<ul style="list-style-type: none">• General Networking Skills• Maintaining connections/Port Forwarding
Other Skills	<ul style="list-style-type: none">• Cloud: AWS/Azure/GCP• Soft Skills• People management skill



Software Development Methodologies

Dev Ops



Contents

waterfall / Iterative

Software Development Lifecycle

SDLC

roles / user / scrum / teams

Agile

STLC, methods, levels, types

Testing

extension to agile development ,

DevOps

automation → docker, git, k8s, Jenkins

↳ devops lifecycle

version code system

source code management

} git, svn,

SCM VCS

OS virtualization : Docker, podman

Containerization

scalable ↗ vertical
horizontal → HA → cluster → k8s

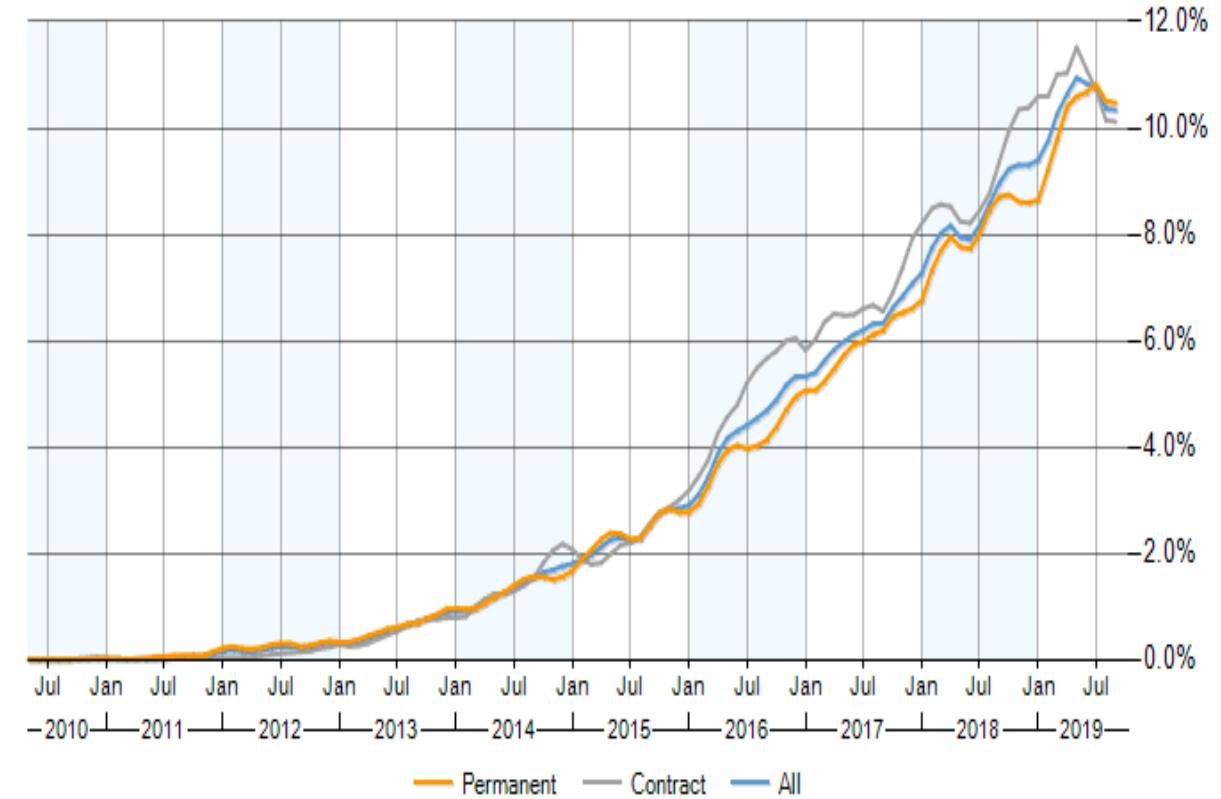
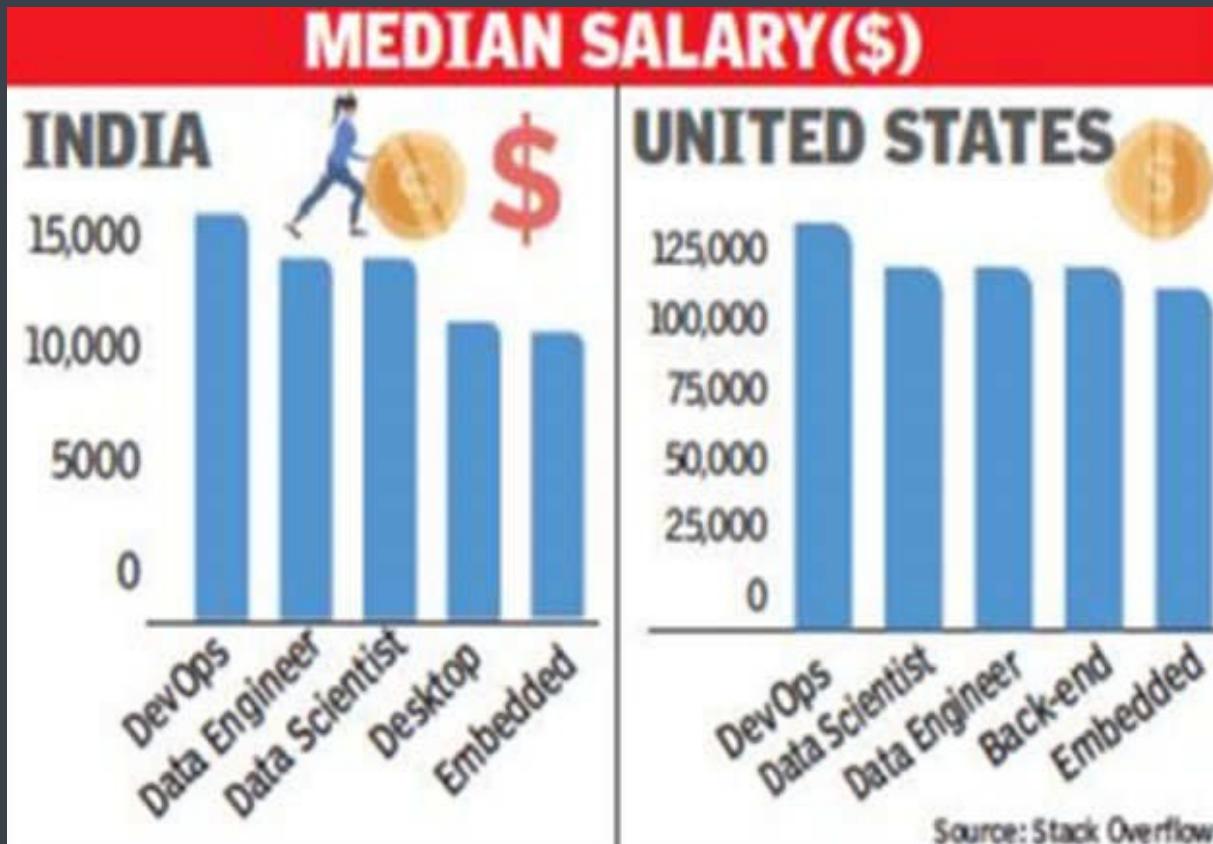
Container Orchestration

pipeline - sequence of stages : Jenkins

CI/CD Pipeline



Why should you bother about DevOps?





Pre-requisites

- Basic Linux knowledge ✓
- Any Development Platforms ✓✓
- Any language (preferred JS/Java) ✓
- Willingness of learning something new



Software Engineering



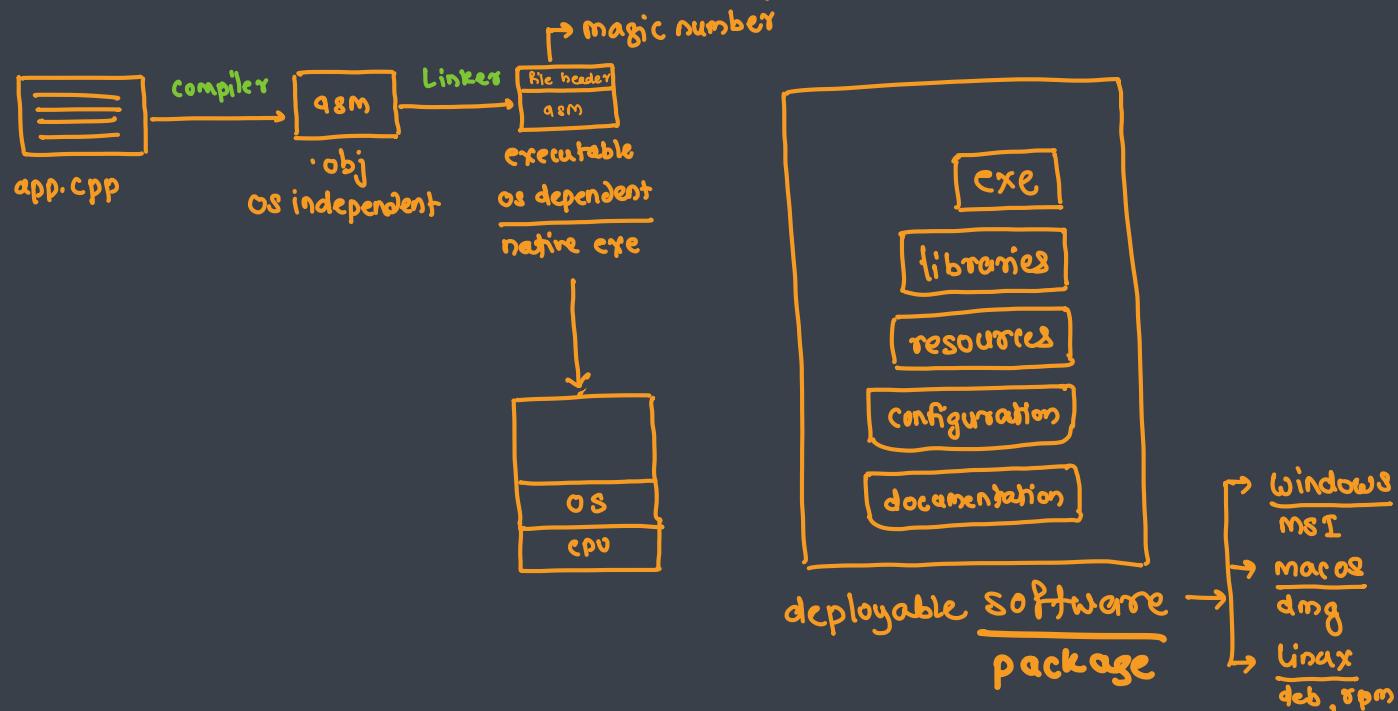
Introduction

Software

- Software is more than just a program code
- A program is an executable code, which serves some computational purpose
- Software is considered to be collection of executable programming code, associated libraries and documentations
- Software, when made for a specific requirement is called software product

Engineering

- All about developing products using well defined principles, methods and procedures



Executable formats

OS	Format	Self Exe	Library
		.o/.out.	static
		.a/.ar	dynamic
mac OS	Mach-O	.o/.out.	.a/.ar
Linux	ELF	.a/.ar	.so
Windows	PE/COFF	.exe	.dll

ELF → Executable and Linkable file format

PE/COFF → Portable Executable Common object ff.

.ar → archive , .so → shared object,



What is Software Engineering?

- Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures
- The application of a systematic, disciplined , quantifiable approach to the development, operation and maintenance of software
- Establishment and use of sound engineering principles in order to obtain software that is reliable and work efficiently on real machines
- The process of developing a software product using software engineering principles and methods



Software Types

- System Software : OS, system utilities
- Application Software : ms office, Adobe photoshop, canva
- Engineering/Scientific Software : Auto CAD, Cadia
- Embedded Software : apps running on raspberry Pi
- AI Software : chatGPT, Grok, Gemini
- Legacy Software : old applications
- Web/Mobile Software : websites → mobile app: what app



Why SE is important

- Helps to build complex systems in timely manner → **stages**
- Ensures high quality of software → **testing**
- Imposes discipline to work
- Minimizes software cost
- Decreases time



Characteristics of good software

→ features → speed / accuracy

- A software product can be judged by what it offers and how well it can be used
- Well-engineered and crafted software is expected to have the following characteristics
 - ① ■ Operational
 - ② ■ Transactional
 - ③ ■ Maintenance



Operational → Non-functional

- This tells us how well software works in operations

- Can be measured on:

- Budget: Software should be developed within the budget
- Usability: Software should be usable by the end user → UI, language (locale), How, memorization
- Efficiency: Software should efficiently use the storage and space → speed / resource usage
- Correctness: Software should provide all the functionalities correctly without having any bug/issue
- Functionality: Software should meet all the requirements of the user
- Dependability: Software should contain all the dependencies in its package → libraries / configurations / resources
- Security: Software should keep the data safe from any external threat
- Safety: Software should not be hazardous or harmful to the environment

→ consistency

→ authentication

→ authorization → JWT tokens

→ availability

→ security at rest (storage) } encryption → conversion of plain text into cipher text

→ security in transit

↳ https

↳ one way encryption → decryption is NOT possible → password encryption

↳ two way encryption → decryption will convert cipher text into plain text



Transitional

environment

- This aspect is important when the software is moved from one platform to another
- Can be measured on
 - Portability: If the software can perform the same operations on different environments and platforms, that shows its Portability
 - Interoperability: It is the ability of the software to use the information transparently
 - Reusability: If on doing slight modifications to the code of the software, we can use it for a different purpose, then it is reusable
 - Adaptability: It is an ability to adapt the new changes in the environment



Maintenance

- Briefs about how well a software has the capabilities to maintain itself in the ever-changing environment
- Can be measured on
 - Maintainability: The software should be easy to maintain by any user
 - Flexibility: The software should be flexible to any changes made to it
 - Extensibility: There should not be any problem with the software on increasing the number of functions performed by it
 - Testability: It should be easy to test the software
 - Modularity: A software is of high modularity if it can be divided into separate independent parts and can be modified and tested separately
 - Scalability: The software should be easy to upgrade
 - vertical
 - horizontal



traditional SDLC methods

modern

Waterfall, Iterative, Agile → DevOps

Software Development

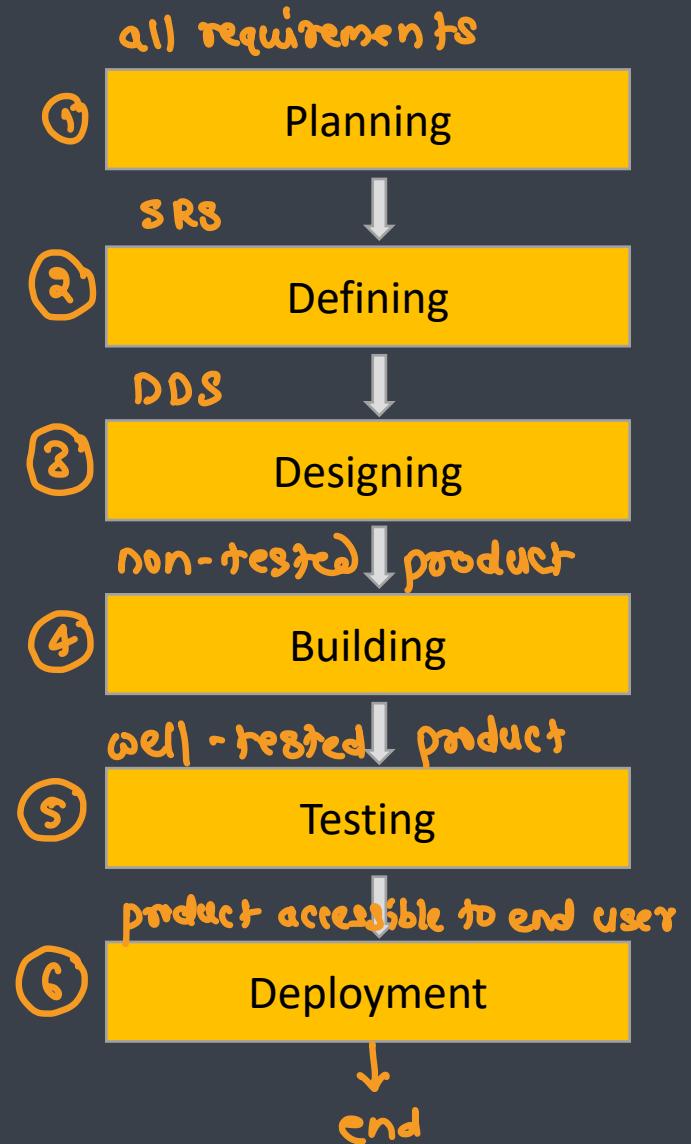
Life Cycle

stages



Overview

- Also called as Software Development Process [SDP]
- Is a well-defined, structured sequence of stages in software engineering to develop the intended software product
- Is a framework defining tasks performed at each step in the software development process
- Aims to produce a high-quality software that
 - meets or exceeds customer expectations
 - reaches completion within times and cost estimates
- Consists of detailed plan (stages) of describing how to develop, test, deploy and maintain the software product

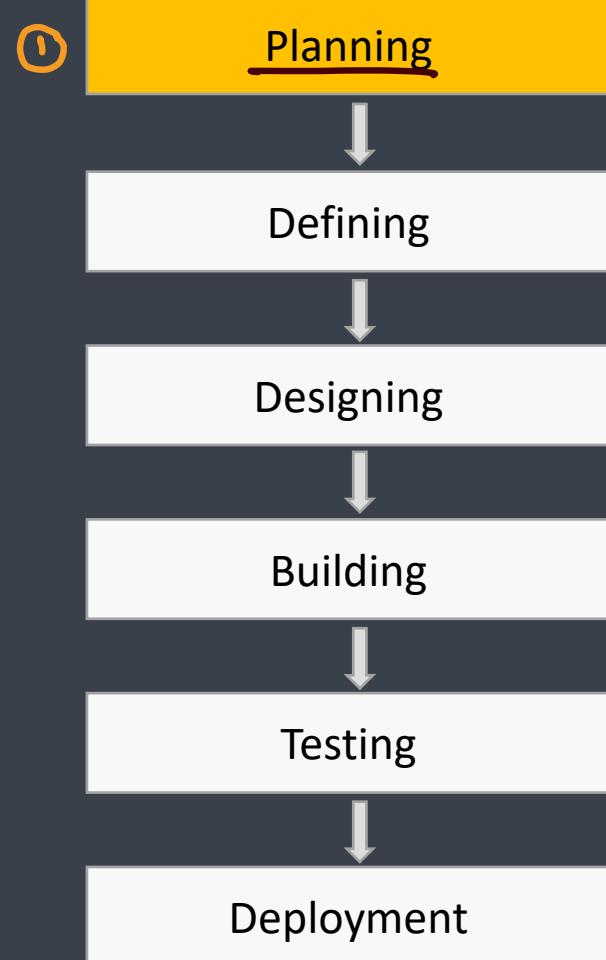


Planning and Requirement Analysis

Requirement gathering + Requirement Analysis ↗ Feasibility
↳ understanding

- The most important and fundamental stage in SDLC
- It is performed by the senior members of the team with inputs from the customer, the sales department, market surveys and domain experts in the industry → **Subject Mater Expert (SME)**
- This information is then used to plan the basic project approach and to conduct product feasibility study in the economical, operational and technical areas
- Planning for the quality assurance requirements and identification of the risks associated with the project is also done in the planning stage
- The outcome of the technical feasibility study is to define the various technical approaches that can be followed to implement the project successfully with minimum risks

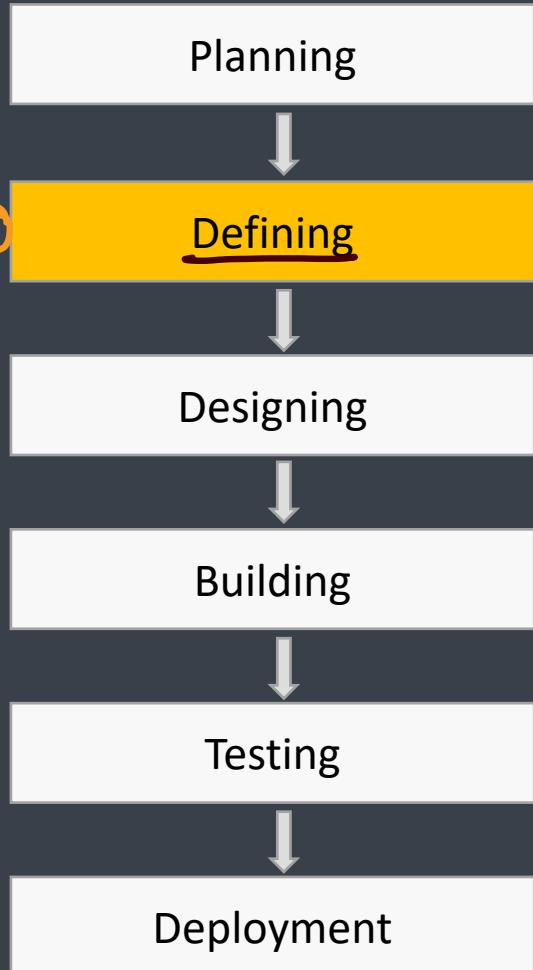
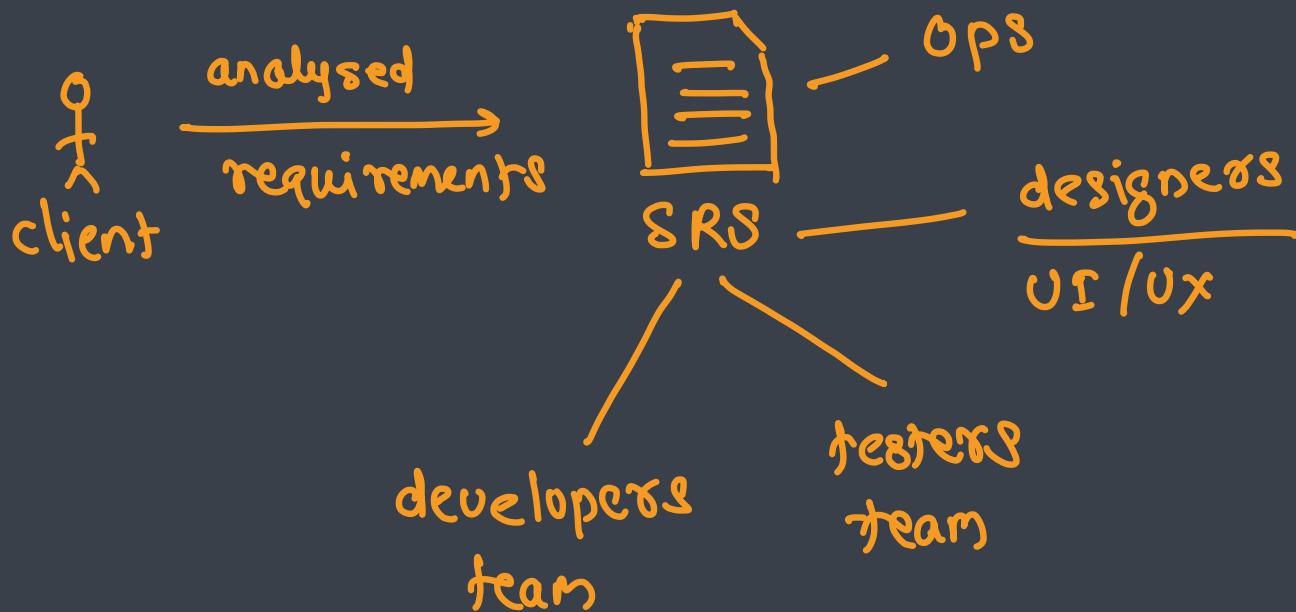
↳ **fronted** + **Backend** + **Database**
React / Angular Express / NestJS PostgreSQL + MongoDB





Defining Requirements

- Once the requirement analysis is done the next step is to clearly define and document the product requirements and get them approved from the customer or the market analysts
- This is done through an **SRS (Software Requirement Specification)** document which consists of all the product requirements to be designed and developed during the project life cycle

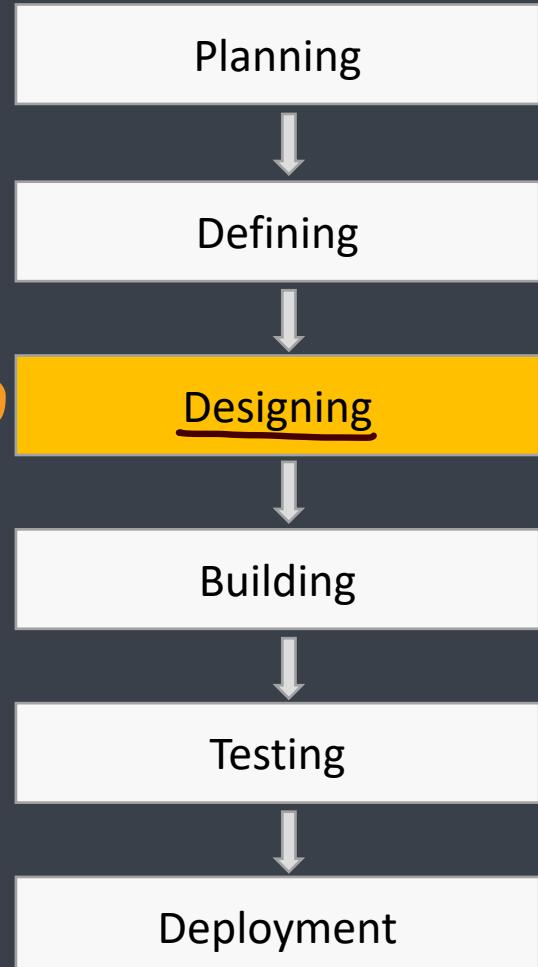
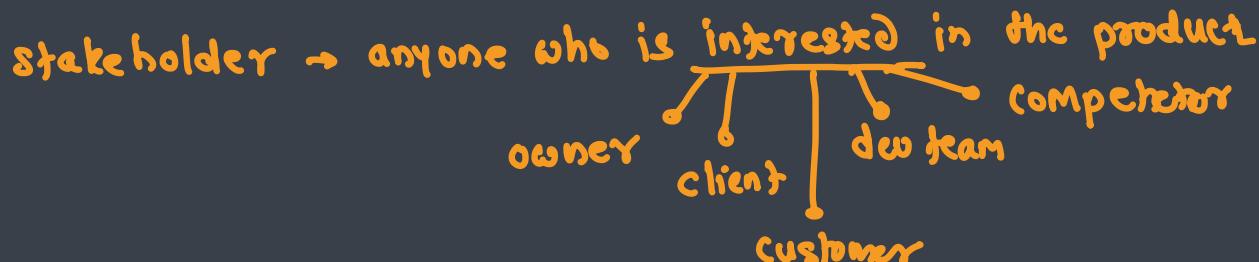




Designing the Product Architecture

monolithic / micro-services / client-server / peer to peer

- SRS is the reference for product architects to come out with the best architecture for the product to be developed
- Based on the requirements specified in SRS, usually more than one design approach for the product architecture is proposed and documented in a DDS - Design Document Specification → possible architectures
- This DDS is reviewed by all the important stakeholders and based on various parameters as risk assessment, product robustness, design modularity, budget and time constraints, the best design approach is selected for the product
- A design approach clearly defines all the architectural modules of the product along with its communication and data flow representation with the external and third party modules (if any)
- The internal design of all the modules of the proposed architecture should be clearly defined with the minutest of the details in DDS



design specification

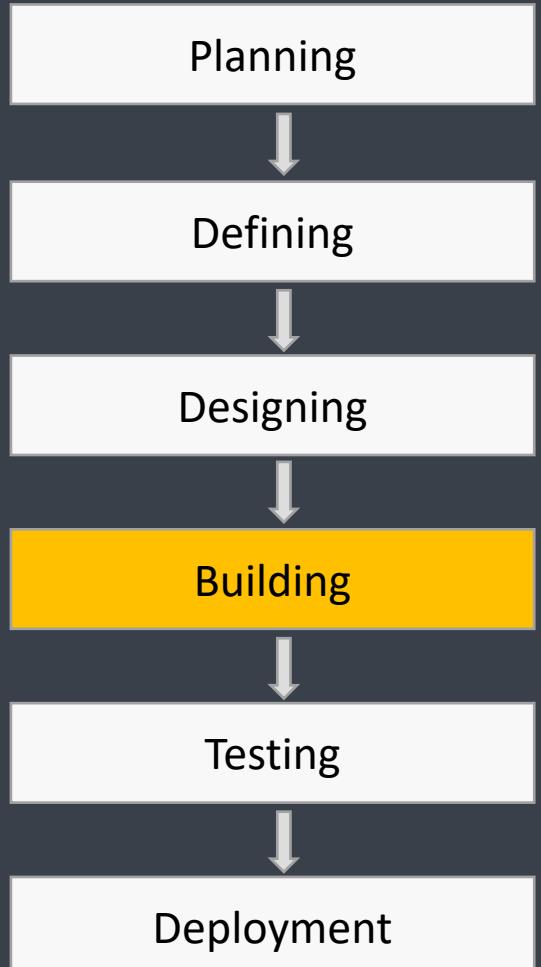
- Technology
 - architecture
 - components
 - tools
 - technical requirement
 - Database
 - Environments
 - Deployment strategies
- 
- application UI/UX
 - database design
 - wireframes
 - screen flow



Building or Developing the Product

- In this stage of SDLC the actual development starts and the product is built
and SRS
- The programming code is generated as per DDS during this stage
- If the design is performed in a detailed and organized manner, code generation can be accomplished without much hassle
- Developers must follow the coding guidelines defined by their organization and programming tools like compilers, interpreters, debuggers, etc. are used to generate the code
- Different high level programming languages such as C, C++, Java, PHP etc. are used for coding
- The programming language is chosen with respect to the type of software being developed

→ always prefer TS over JS

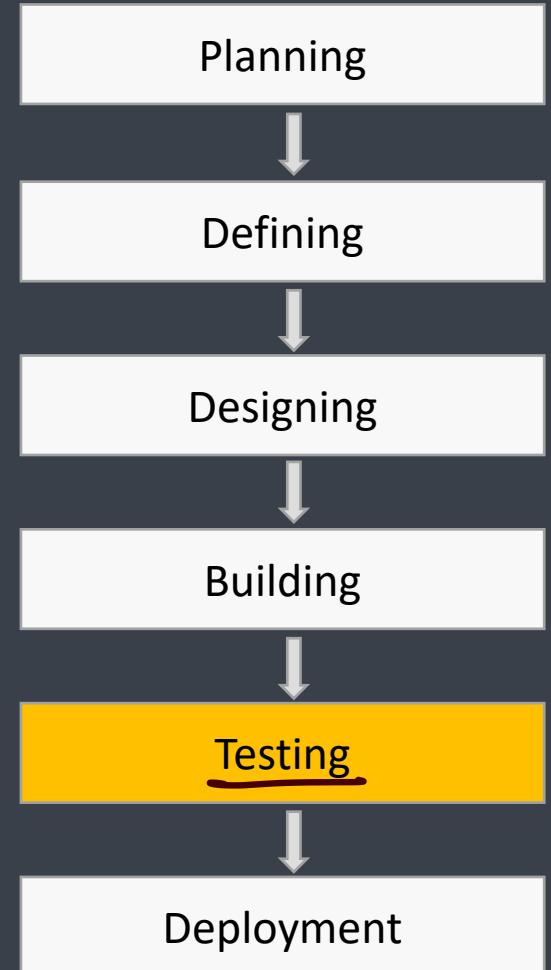
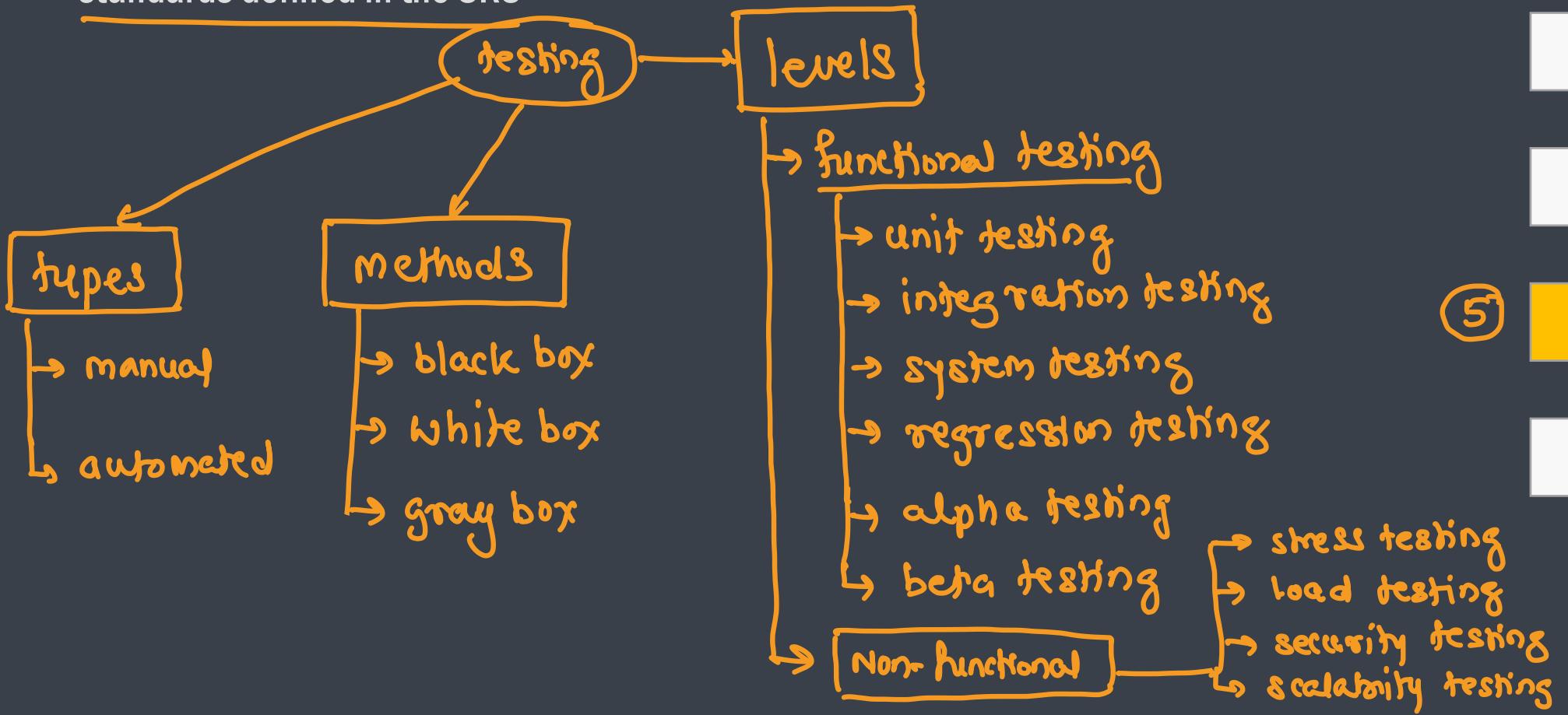




Testing the Product

testing → verifying the functionality

- This stage is usually a subset of all the stages as in the modern SDLC models
- The testing activities are mostly involved in all the stages of SDLC
- However, this stage refers to the testing only stage of the product where product defects are reported, tracked, fixed and retested, until the product reaches the quality standards defined in the SRS



Deployment in the Market and Maintenance

Environment → used to run the software
→ includes → OS / runtime / VM / libraries / frameworks

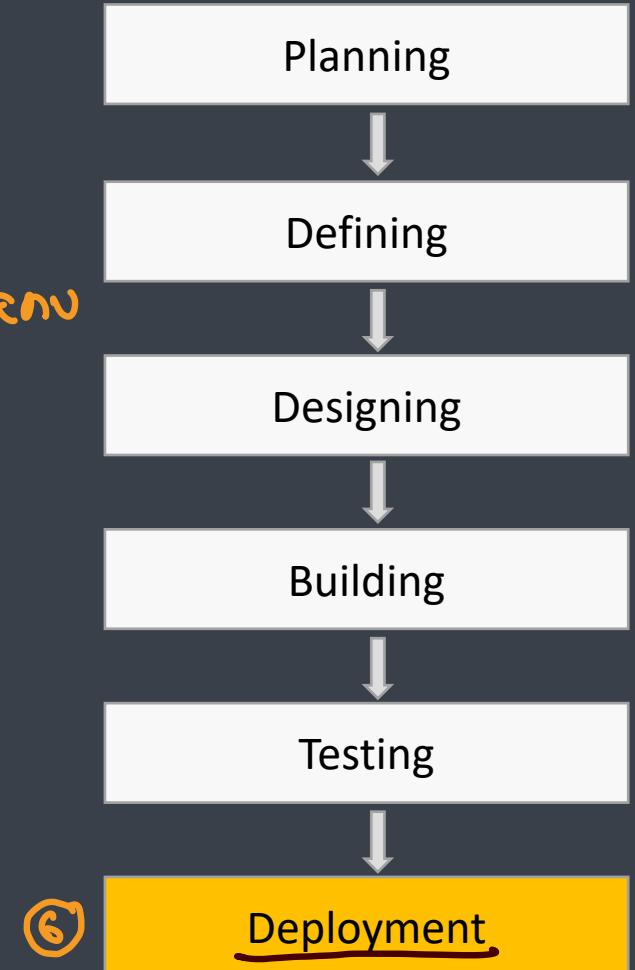
- Once the product is tested and ready to be deployed it is released formally in the appropriate market
- Sometimes product deployment happens in stages as per the business strategy of that organization → Dev env, staging, pre-prod, production env
- The product may first be released in a limited segment and tested in the real business environment (UAT- User acceptance testing)
- Then based on the feedback, the product may be released as it is or with suggested enhancements in the targeting market segment
- After the product is released in the market, its maintenance is done for the existing customer base

Market
→ Website → cloud
aws
azure
gcp
others

→ ios app → apple app store

→ android app → google play store, amazon marketplace,
samsung market

→ Native apps → web sites



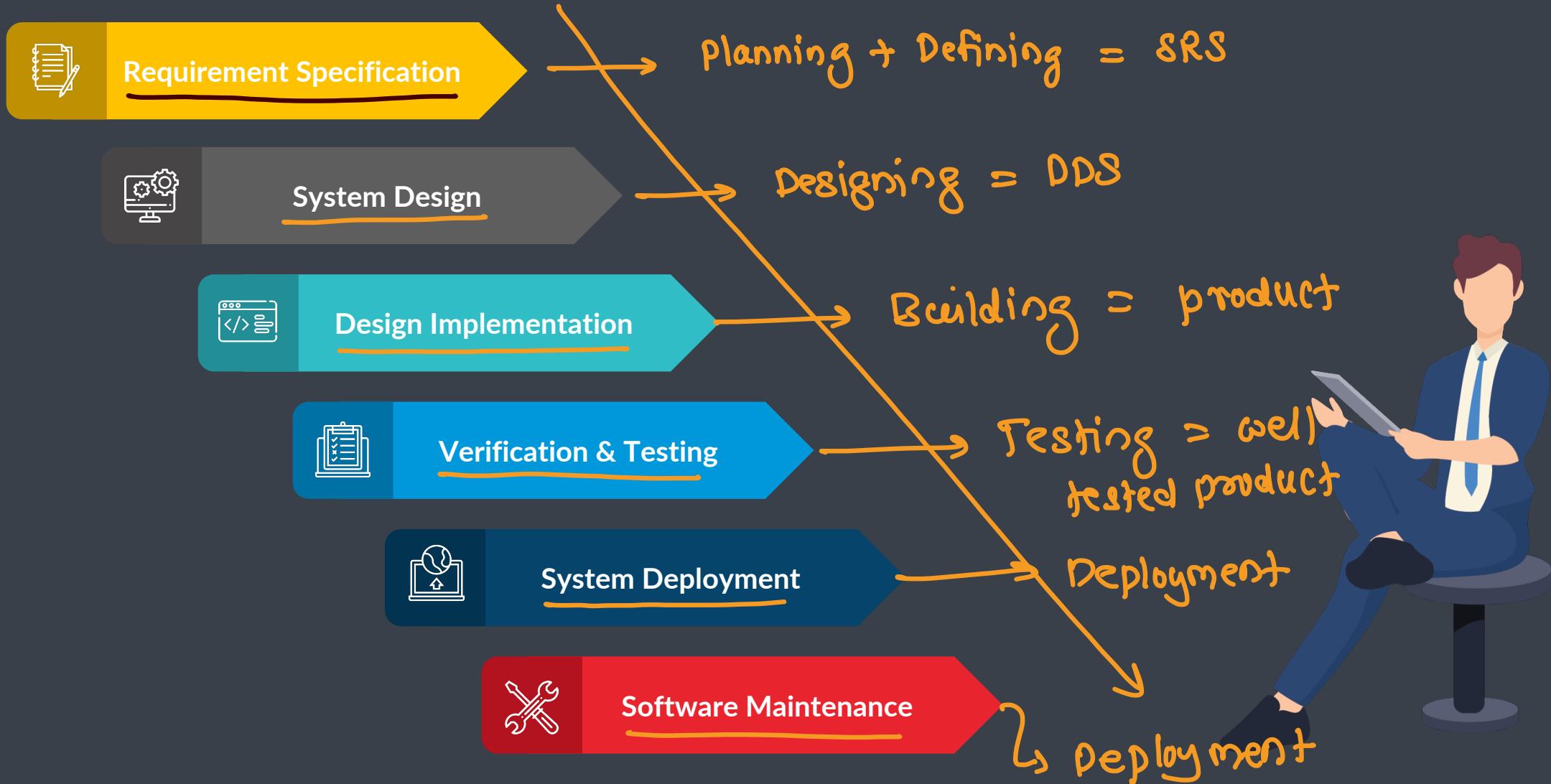


SDLC Models

- There are various software development life cycle models defined and designed which are followed during the software development process
- Also referred as Software Development Process Models
- Models
 - ■ Waterfall Model
 - Iterative Model
 - Spiral Model
 - V-Model
 - Big Bang Model
 - ■ Agile Model *



Waterfall Model → top to bottom execution



e-commerce app

- product mgmt
- user mgmt
- order mgmt
- cart mgmt
- payment mgmt
- Notification mgmt
- recommendation [ML]

- * requirement gathering and analysis of whole product
- * prepare SRS for whole product
- * once SRS is finalized do not change the requirements



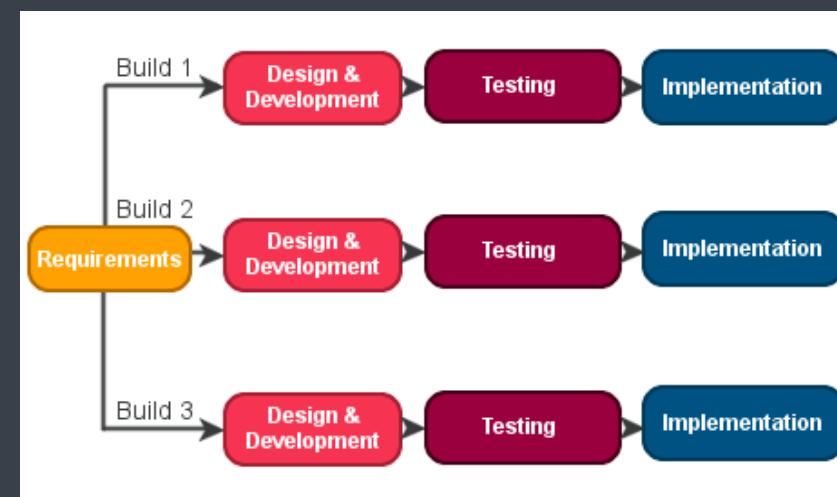
Waterfall Model – Procs and Cons

- Simple and easy to understand and use
- Easy to manage due to the rigidity of the model
- Each phase has specific deliverables and a review process
- Phases are processed and completed one at a time
- Works well for smaller projects where requirements are very well understood
- Clearly defined stages
- Well understood milestones
- Easy to arrange tasks
- Process and results are well documented
- No working software is produced until late during the life cycle
- High amounts of risk and uncertainty
- Not a good model for complex and object-oriented projects
- Poor model for long and ongoing projects
- Not suitable for the projects where requirements are at a moderate to high risk of changing. So, risk and uncertainty is high with this process model
- It is difficult to measure progress within stages
- Cannot accommodate changing requirements
- Adjusting scope during the life cycle can end a project
- Integration is done as a "big-bang. at the very end, which doesn't allow identifying any technological or business bottleneck or challenges early

Iterative Model → incremental approach → incrementing / adding features version by version

- In the Iterative model, iterative process starts with a simple implementation of a small set of the software requirements and iteratively enhances the evolving versions until the complete system is implemented and ready to be deployed
- An iterative life cycle model does not attempt to start with a full specification of requirements ~~*-*~~
- Instead, development begins by specifying and implementing just part of the software, which is then reviewed to identify further requirements
- This process is then repeated, producing a new version of the software at the end of each iteration of the model

<u>phase I</u>	<u>phase II</u>	<u>phase III</u>
users	cart	recommendation
categories	ordering	notification
products	payment	version 3
<u>version 1</u>	<u>version 2</u>	





Iterative Model Pros and Cons

- Some working functionality can be developed quickly and early in the life cycle
- Results are obtained early and periodically
- Parallel development can be planned
- Progress can be measured
- Less costly to change the scope/requirements
- Testing and debugging during smaller iteration is easy
- Risks are identified and resolved during iteration; and each iteration is an easily managed milestone
- Easier to manage risk - High risk part is done first
- With every increment, operational product is delivered
- Issues, challenges and risks identified from each increment can be utilized/applied to the next increment
- Risk analysis is better
- It supports changing requirements
- Initial Operating time is less
- More resources may be required
- Although cost of change is lesser, but it is not very suitable for changing requirements
- More management attention is required
- System architecture or design issues may arise because not all requirements are gathered in the beginning of the entire life cycle
- Defining increments may require definition of the complete system
- Not suitable for smaller projects
- Management complexity is more
- End of project may not be known which is a risk
- Highly skilled resources are required for risk analysis
- Projects progress is highly dependent upon the risk analysis phase

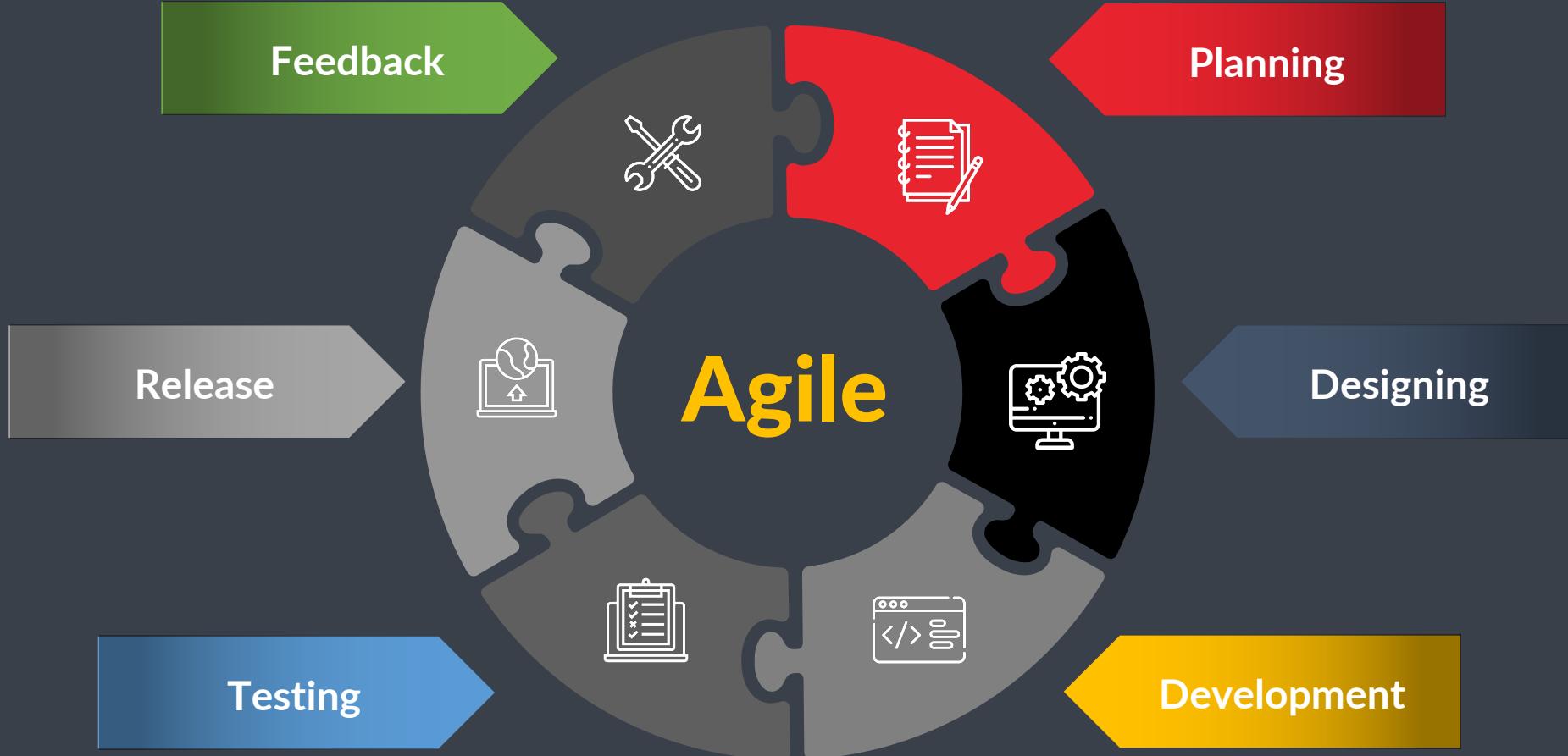


Agile Methodologies





Agile Development





Agile Manifesto

- In February 2001, at the Snowbird resort in Utah, 17 software developers met to discuss lightweight development methods
- The outcome of their meeting was the following Agile Manifesto for software development
 - We are uncovering better ways of developing software by doing it and helping others do it
 - Through this work, we have come to value ↳ practical approach
- **dev team**
 - Individuals and interactions over Processes and tools
 - Working software over Comprehensive documentation
 - Customer collaboration over Contract negotiation ↳ * → feed back
 - Responding to change over Following a plan
- That is, while there is value in the items on the right, we value the items on the left more



Principles of Agile Manifesto

■ Customer Satisfaction

- Highest priority is given to satisfy the requirements of customers through early and continuous delivery of valuable software

■ Welcome Change

- Changes are inevitable during software development
- Ever-changing requirements should be welcome, even late in the development phase.
- Agile processes should work to increase customers' competitive advantage

■ Deliver a Working Software

- Deliver a working software frequently, ranging from a few weeks to a few months, considering shorter time-scale

■ Collaboration

- Business people and developers must work together during the entire life of a project

■ Motivation

- Projects should be built around motivated individuals
- Provide an environment to support individual team members and trust them so as to make them feel responsible to get the job done

■ Face-to-face Conversation

- Face-to-face conversation is the most efficient and effective method of conveying information to and within a development team



Principles of Agile Manifesto

- Measure the Progress as per the Working Software

- Working software is the key and it should be the primary measure of progress

- Maintain Constant Pace

- Agile processes aim towards sustainable development
 - The business, the developers, and the users should be able to maintain a constant pace with the project

- Monitoring

- Pay regular attention to technical excellence and good design to enhance agility

- Simplicity

- Keep things simple and use simple terms to measure the work that is not completed

- Self-organized Teams

- An agile team should be self-organized and should not depend heavily on other teams because the best architectures, requirements, and designs emerge from self-organized teams

- Review the Work Regularly

- Review the work done at regular intervals so that the team can reflect on how to become more effective and adjust its behavior accordingly



Agile Methodologies

- The most popular Agile methods include

- Rational Unified Process

- ① ■ Scrum ★★★

- Crystal Clear

- Extreme Programming

- Adaptive Software Development

- Feature Driven Development

- Dynamic Systems Development Method (DSDM)

- ② • Kanban ★★★



What is Scrum ?

→ roles
→ artefacts
→ pillars
→ events

- Scrum isn't a process, it's a framework that facilitates processes amongst other things
- Is an agile way to manage a project
- Management framework with far reaching abilities to control and manage the iterations and increments in all project types
- One of the implementations of agile methodology
- Incremental builds are delivered to the customer in every two to three weeks time
- Ideally used in the project where the requirement is rapidly changing
- The framework is made up of a Scrum team, individual roles, events, artefacts, and rules



Scrum Principles

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
- Welcome changing requirements, even late in development
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale
- Business people and developers must work together daily throughout the project
- Build projects around motivated individuals
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
- Working software is the primary measure of progress
- Agile processes promote sustainable development
- Continuous attention to technical excellence and good design enhances agility
- Simplicity, the art of maximizing the amount of work not done, is essential
- The best architectures, requirements, and designs emerge from self-organizing teams
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

E-commerce app

- user mgmt
- product mgmt
- order mgmt
- cart mgmt
- payment mgmt
- Notification mgmt
- recommendation [ML]

Sprint → time boxed event used to complete one app version

→ mostly it is between 1 wk to 4 wks

→ most often duration = 2 wks

project

↳ modules

↳ functionality

↳ actions / stories

stories → backlog / product backlog

story = action to be implemented

Epic = collection of related stories

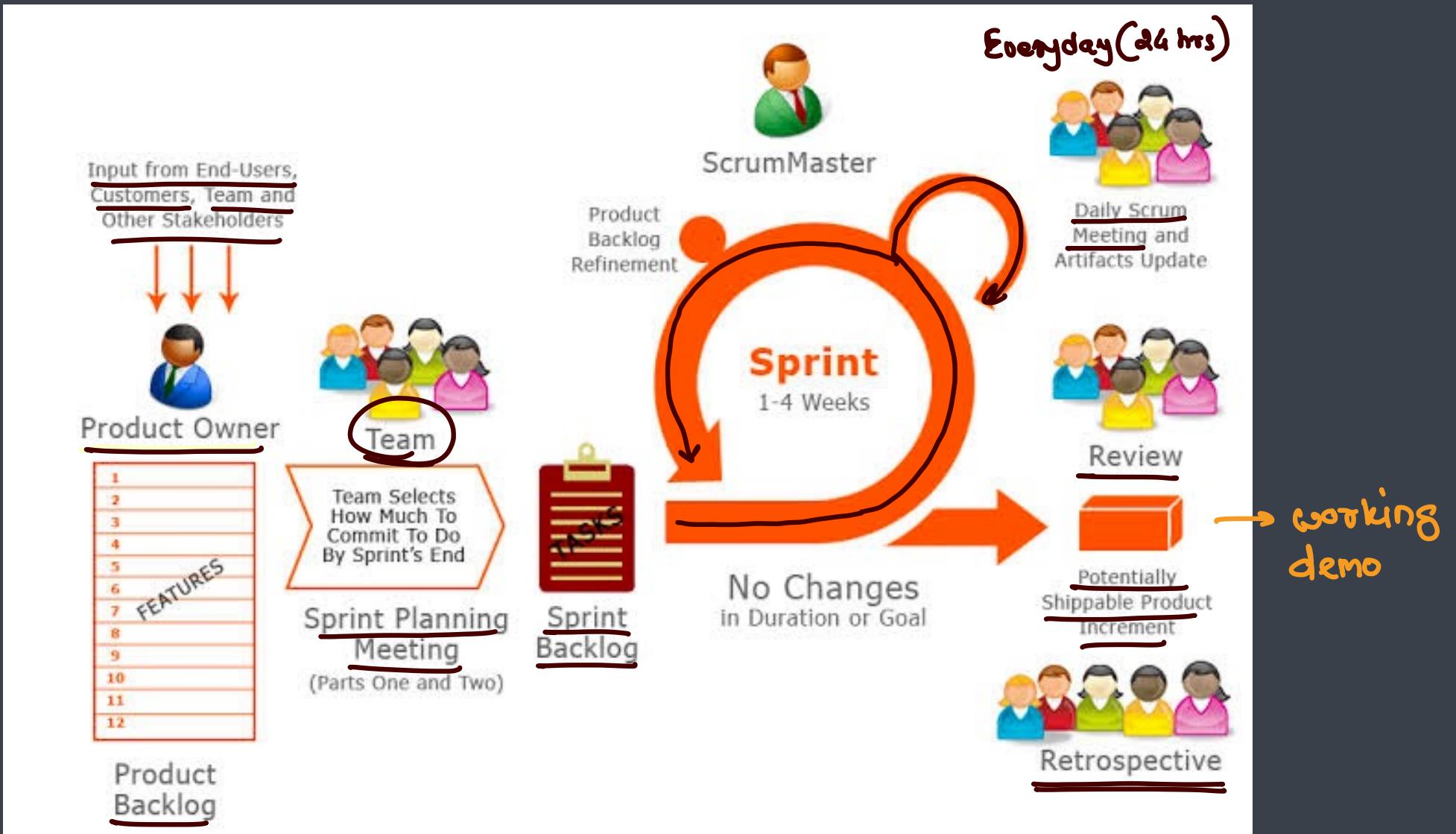
- ① create user table
- ② create API to register user
- ③ create screen for user registration and call register API
- ④ create an API for user login
- ⑤ create screen for user login and call login API
- ⑥ create API for resetting password
- ⑦ create screen for resetting user password
- ⑧ create table product
- ⑨ create an API for adding product
- ⑩ create screen to add product and call API

sprint backlog → list of stories selected in a sprint

- ① create user table
- ② create API to register user
- ③ create screen for user registration and call register API
- ④ create an API for user login
- ⑤ create screen for user login and call login API
- ⑥ create API for resetting password
- ⑦ create screen for resetting user password



Scrum Workflow





Scrum Events

Sprint Planning

- Happens at the start of every sprint
- it should probably be about between four and eight hours

daily standup → Every 24 hrs

Daily Scrum

- This is a very short time-boxed event
- Usually only lasting no more than 15 minutes

Sprint Review

- Collaborative events to demo what has been achieved and to help keep everyone who's involved working together

Sprint Retrospective

- About continuous process and improvement and we need to take what we've learned into the next sprint planning session

Sprint

- It is really the beating heart of scrum and all the scrum events take place in it



Scrum Roles

Product Owner

- Job to understand and engage with the stakeholders to understand what needs to be done and create that backlog
- Also need to prioritize that backlog

Scrum Master

- Helps the entire team achieve the scrum goals and work within scrum
- Support the product owner with their responsibilities in terms of managing the backlog as well as, supporting the development team

Dev Team

- The people who are creating the product or service and delivering done increments at the end of each sprint
- Includes developers, tester, writers, graphics artists and others



Scrum Artifacts

Product Backlog

list of all stories to complete the project

Sprint Backlog

list of selected stories to complete in a sprint

Increment

version of app to be delivered at the end of sprint



Scrum Pillars



All parts of the process should be transparent, open, and honest for everyone to see

Transparency



Everything that is worked on during a sprint can be inspected by the team to make sure that it is achieving what it needs to.

Inspection



If a member of the Scrum team or a stakeholder notices that things aren't going according to plan, the team will need to change up what they're doing to fix this as quickly as possible

Adaptation



Scrum Values

Courage

Courage to do the right thing and work on tough problems

Focus

Focus on the sprint and its goal

Commitment

Commitment to the team and sprint goal

Respect

Respect, for each other by helping people to learn the things that you're good at, and not judging the things that others aren't good at

Openness

Be open and honest and let people know what you're struggling with challenges and problems that are stopping you from achieving success



How scrum pillars help us?

Self-organization

- This results in healthier shared ownership among the team members
- It is also an innovative and creative environment which is conducive to growth

Collaboration

- Essential principle which focuses collaborative work

Time-boxing

- Defines how time is a limiting constraint in Scrum method
- Daily Sprint planning and Review Meetings

Iterative Development

- Emphasizes how to manage changes better and build products which satisfy customer needs
- Defines the organization's responsibilities regarding iterative development



Agile vs traditional models

Agile Methodologies	Traditional Methodologies
Incremental value and risk management	Phased approach with an attempt to know everything at the start
Embracing change	Change prevention
Deliver early, fail early	Deliver at the end, fail at the end
Transparency	Detailed planning, stagnant control
Inspect and adapt	Meta solutions, tightly controlled procedures and final answers
Self managed	Command and control
Continual learning	Learning is secondary to the pressure of delivery



Agile - Advantages

- Very realistic approach to software development
- Promotes teamwork and cross training
- Functionality can be developed rapidly and demonstrated
- Resource requirements are minimum
- Suitable for fixed or changing requirements
- Delivers early partial working solutions
- Good model for environments that change steadily
- Minimal rules, documentation easily employed
- Enables concurrent development and delivery within an overall planned context
- Little or no planning required
- Easy to manage
- Gives flexibility to developers



Agile - Disadvantages

- Not suitable for handling complex dependencies.
- More risk of sustainability, maintainability and extensibility.
- An overall plan, an agile leader and agile PM practice is a must without which it will not work.
- Strict delivery management dictates the scope, functionality to be delivered, and adjustments to meet the deadlines.
- Depends heavily on customer interaction, so if customer is not clear, team can be driven in the wrong direction.
- There is a very high individual dependency, since there is minimum documentation generated.
- Transfer of technology to new team members may be quite challenging due to lack of documentation.



Scrum tools

☞ **Jira** - <https://www.atlassian.com/software/jira/>

- **Clarizen** - <https://www.clarizen.com/>
- **GitScrum** - <https://site.gitscrum.com/>
- **Vivify Scrum** - <https://www.vivifyscrum.com/>
- **Yodiz** - <https://www.yodiz.com/>
- **ScrumDo** - <https://www.scrumdo.com/>
- **Quicksrum** - <https://www.quicksrum.com/>
- **Manuscript** - <https://www.manuscript.com/>
- **Scrumwise** - <https://www.scrumwise.com/>
- **Axosoft** - <https://www.axosoft.com/>



Agile Methodologies – Scrum Terminologies

- **Scrum**
 - A framework to support teams in complex product development
- **Scrum Board**
 - A physical board to visualize information for and by the Scrum Team, used to manage Sprint Backlog
- **Scrum Master**
 - The role within a Scrum Team accountable for guiding, coaching, teaching and assisting a Scrum Team and its environments in a proper understanding and use of Scrum
- **Scrum Team**
 - A self-organizing team consisting of a Product Owner, Development Team and Scrum Master
- **Self-organization**
 - The management principle that teams autonomously organize their work
- **Sprint**
 - Time-boxed event of 30 days, or less, that serves as a container for the other Scrum events and activities.
- **Sprint Backlog**
 - An overview of the development work to realize a Sprint's goal, typically a forecast of functionality and the work needed to deliver that functionality



Agile Methodologies – Scrum Terminologies

- **Sprint Goal**
 - A short expression of the purpose of a Sprint, often a business problem that is addressed
- **Sprint Retrospective**
 - Time-boxed event of 3 hours, or less, to end a Sprint to inspect the past Sprint and plan for improvements
- **Sprint Review**
 - Time-boxed event of 4 hours, or less, to conclude the development work of a Sprint
- **Stakeholder**
 - A person external to the Scrum Team with a specific interest in and knowledge of a product that is required for incremental discovery
- **Development Team**
 - The role within a Scrum Team accountable for managing, organizing and doing all development work
- **Daily Scrum**
 - Daily time-boxed event of 15 minutes for the Development Team to re-plan the next day of development work during a Sprint

git

basic workflow

```
# initialize an empty repository
> git init

# get the current status of every file present in working directory
> git status

# get short status
> git status -s

# note: git status -s command returns a status with two characters
# 1st character: shows the status of file with respect to the staging area
# 2nd character: shows the status of file with respect to the working area

# ?: untracked file (the repository does not recognize the file or does
not have any version of this file created yet)
# A : the changes are present in the staging area and will be added to the
repository after committing
# M: the changes are present in the working directory and the file is
modified in the working directory
# M : the changes are moved to the staging area

# add the file/files to staging area
> git add <file name>
> git add .

# create a version of all the files present in staging area
> git commit -m <message>

# get the list of all commits
> git log

# get the list of logs in one line with color and graph
> git log --oneline --graph --color

# get the difference between the latest version and the previous version
> git diff

# get the last version from repository and replace it with current version
# note: even if the file is deleted from the working directory, you can
still bring it (last version of it) back from the repository
> git checkout <file name>

# soft reset:
# - move all the changes from staging area to the working directory
# - no changes will be lost in the process
```

```
> git reset

# hard reset:
# - remove all the changes from staging area or working directory
# - get last version of all the files and replace with current version
# (irrespective of their location)
# - note: please execute this command on your own risk
> git reset --hard

# remove all the metadata or repository
# note: this command will remove all the history
> rm -rf .git
```

shared repository

```
# types
# - local
#   - present on local machine
# - shared
#   - present on the server
#   - also known as remote repository
#   - to create a shared repository
#   - create a project or repository on shared repository server
#   - connect the local repository to the remote one
#   - e.g. GitHub, GitLab, BitBucket

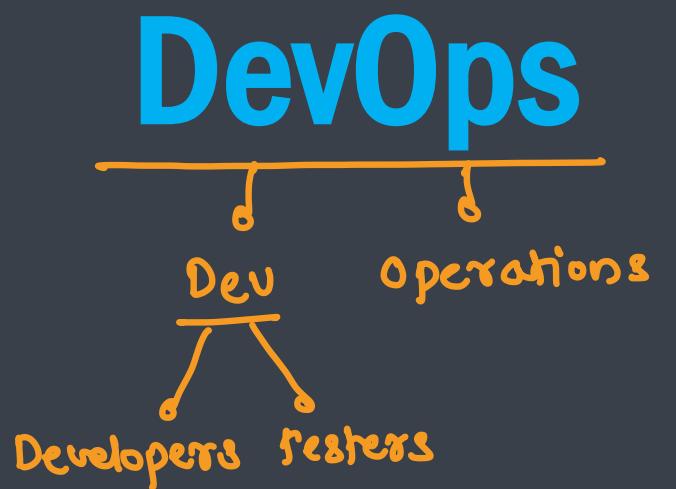
# get the remote repository url
> git remote -v

# add the remote repository
# > git remote add <server alias> <repository url>
> git remote add origin <repository>

# send all the local changes to the remote repository
> git push <alias> <branch name>

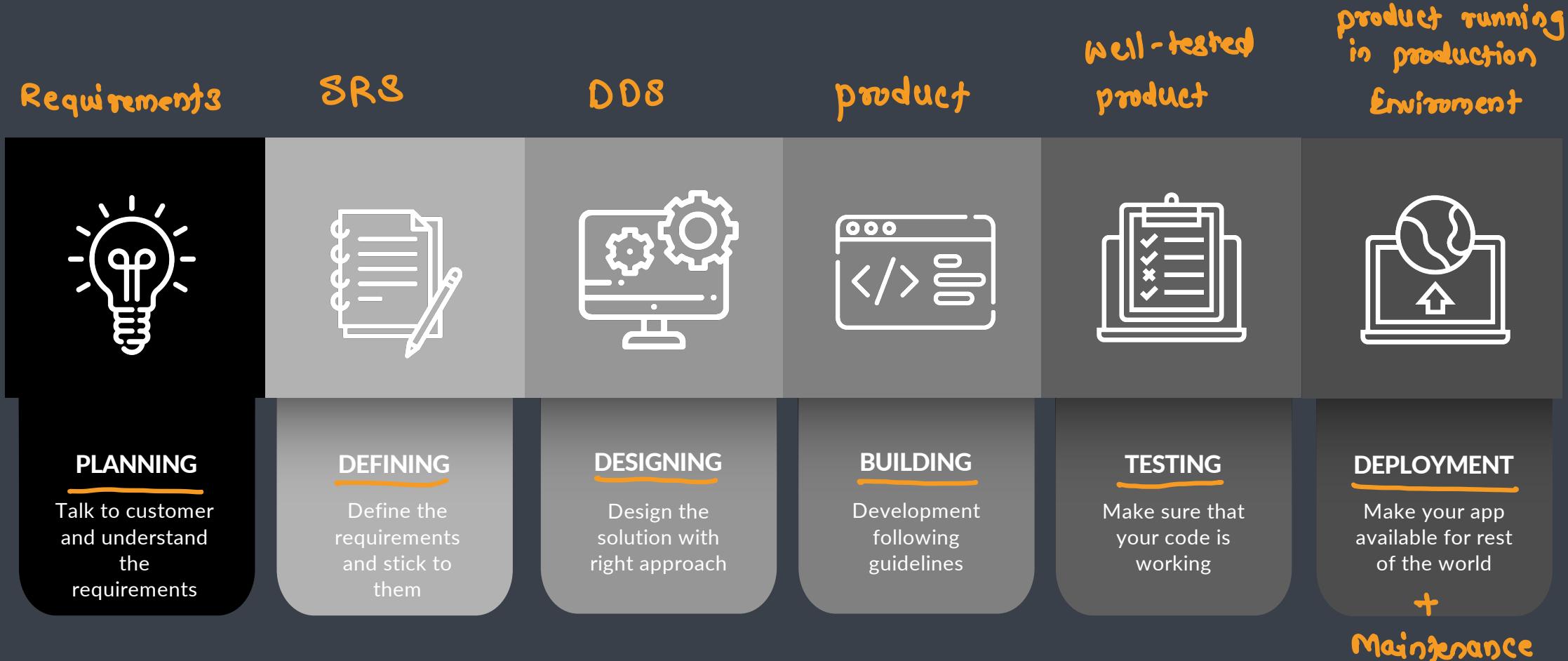
# pull all the changes from remote to local repository
> git pull <alias> <branch name>

# remove the remote repository from local one
> git remote remove <alias>
```



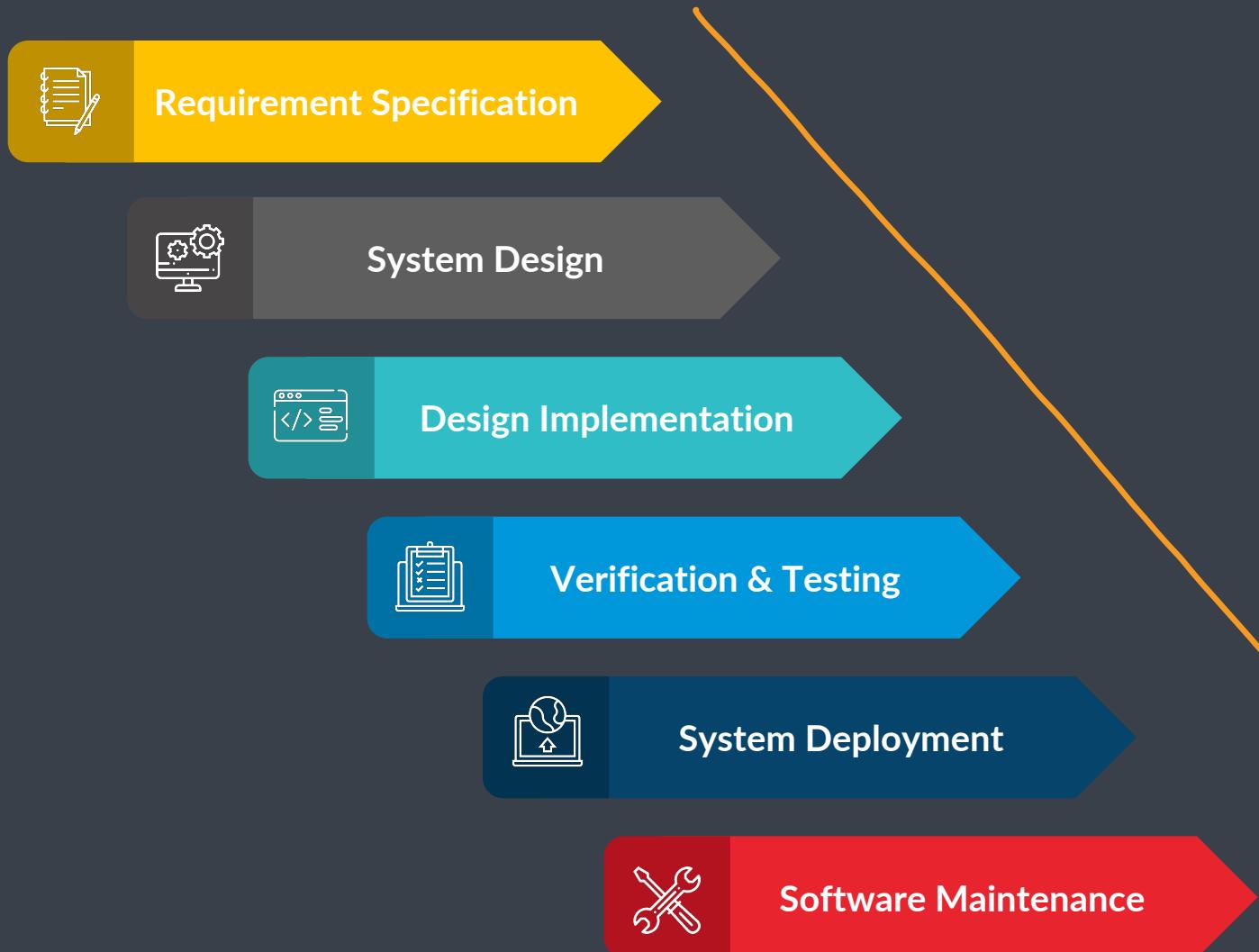


Software Development Lifecycle





Waterfall Model



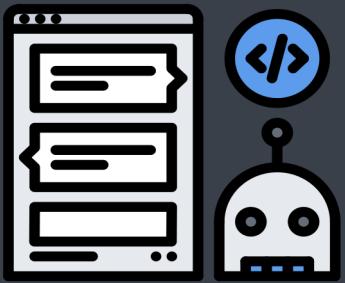


Entities involved



Developer

development/
programming/
coding



Testers

test the product



Operations Team

Responsibilities



Dev Team

android → .apk
ios → .ipa
website → webpack
, vite

Developers and Testers

- Developers ↗ coding
 - Develop the application
 - Package the application
 - Fix the bugs
 - Maintain the application
- Testers
 - Thoroughly test the application manually or using test automation
 - Report the bugs to the developer

windows native → .msi
linux native → .deb / .rpm
macos native → .dmg

Operations Team

- Make all the necessary resources ready ↗ moving app from one to another environment
- Deploy the application
- Maintain multiple environments
- Continuously monitor the application
- Manage the resources



→ computers
→ software
→ Licenses
→ Networking
→ Hardware



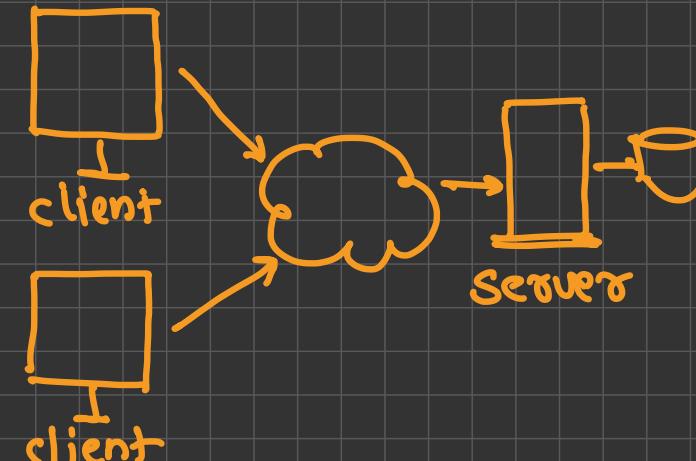
Dev Environment

developers



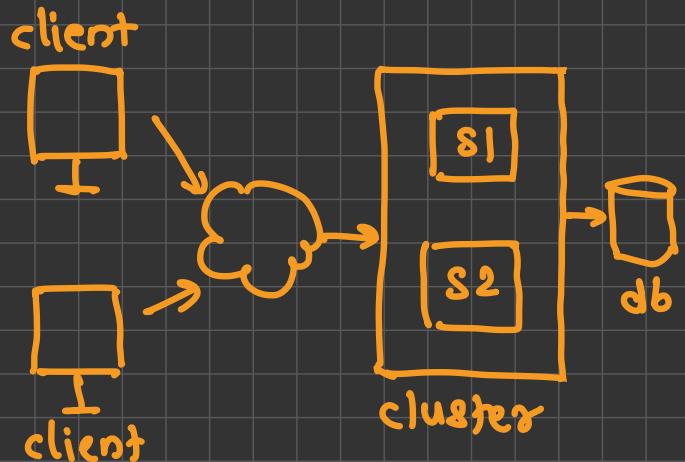
staging Environment

testers

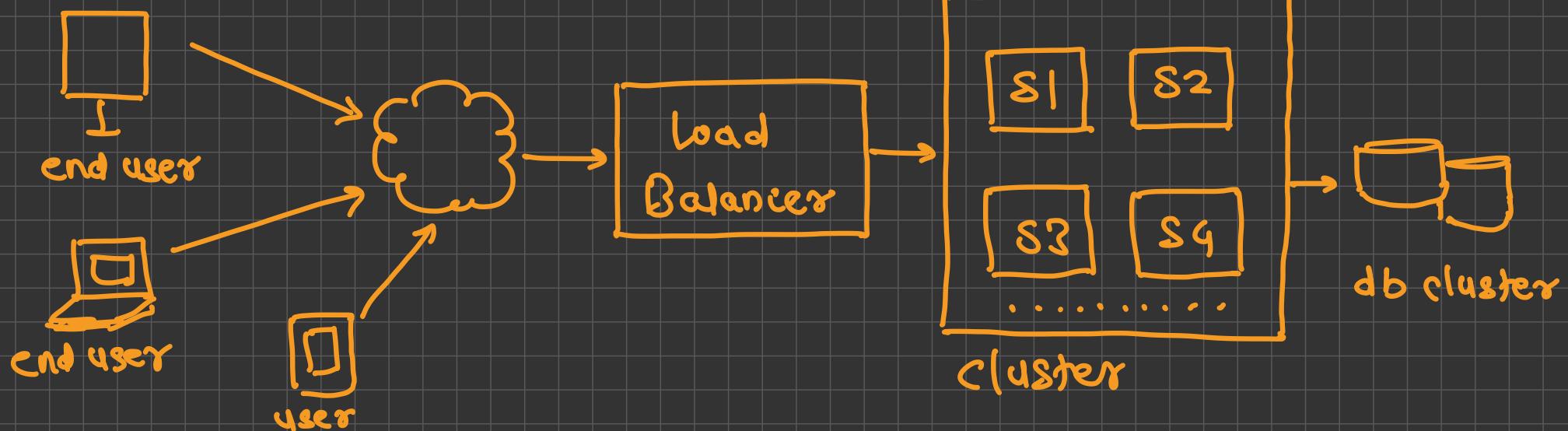


pre - production Environment

internal testers / beta testers



production Environment → Horizontal scaling → Highly Available
end users (external)





Challenges



Developers and Testers

- The process is slow
- The pressure to work on the newer features and fix the older code
- Not flexible

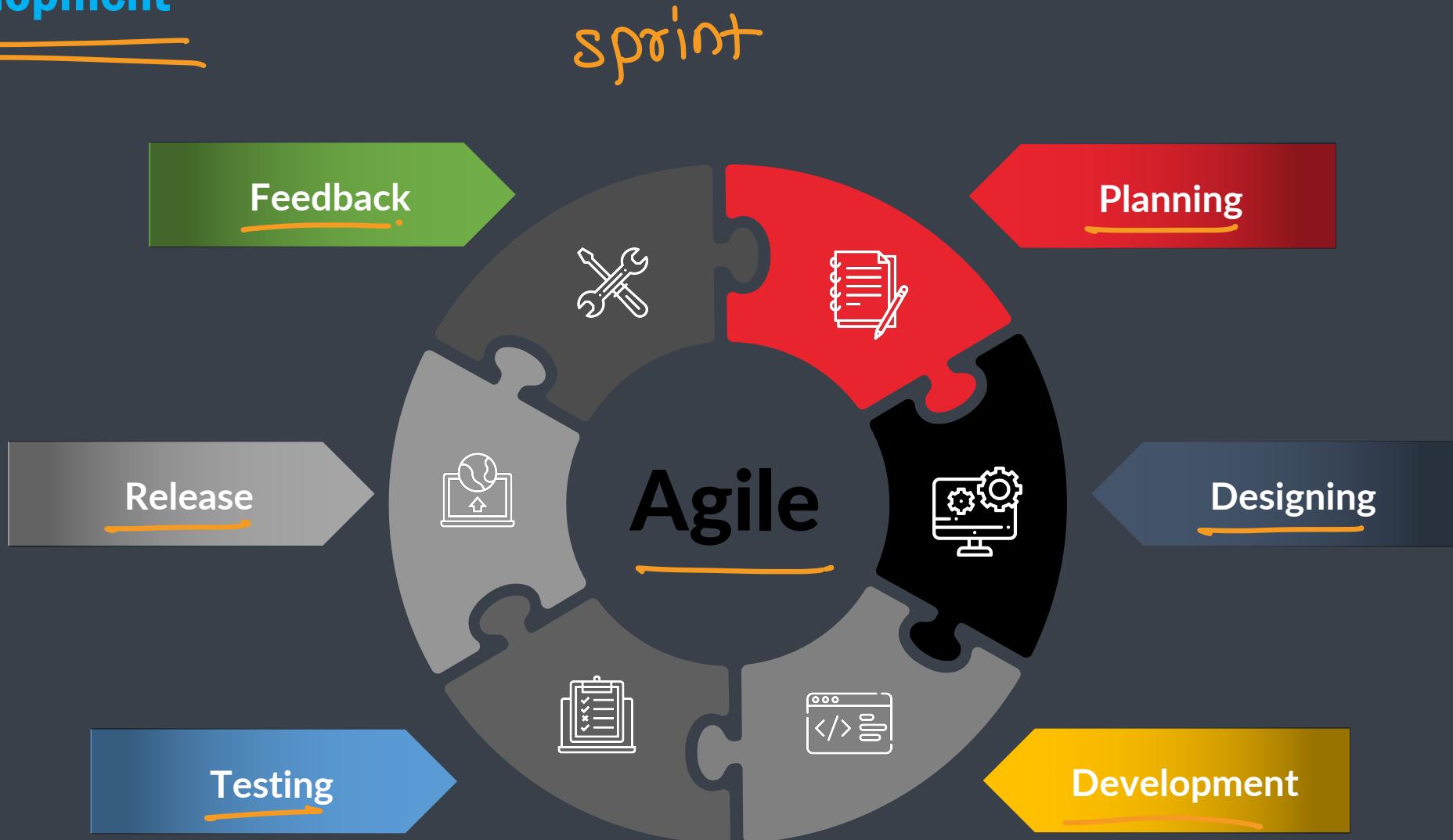


Operations Team

- Uptime ~~x downtime~~
- Configure the huge infrastructure
- Diagnose and fix the issue

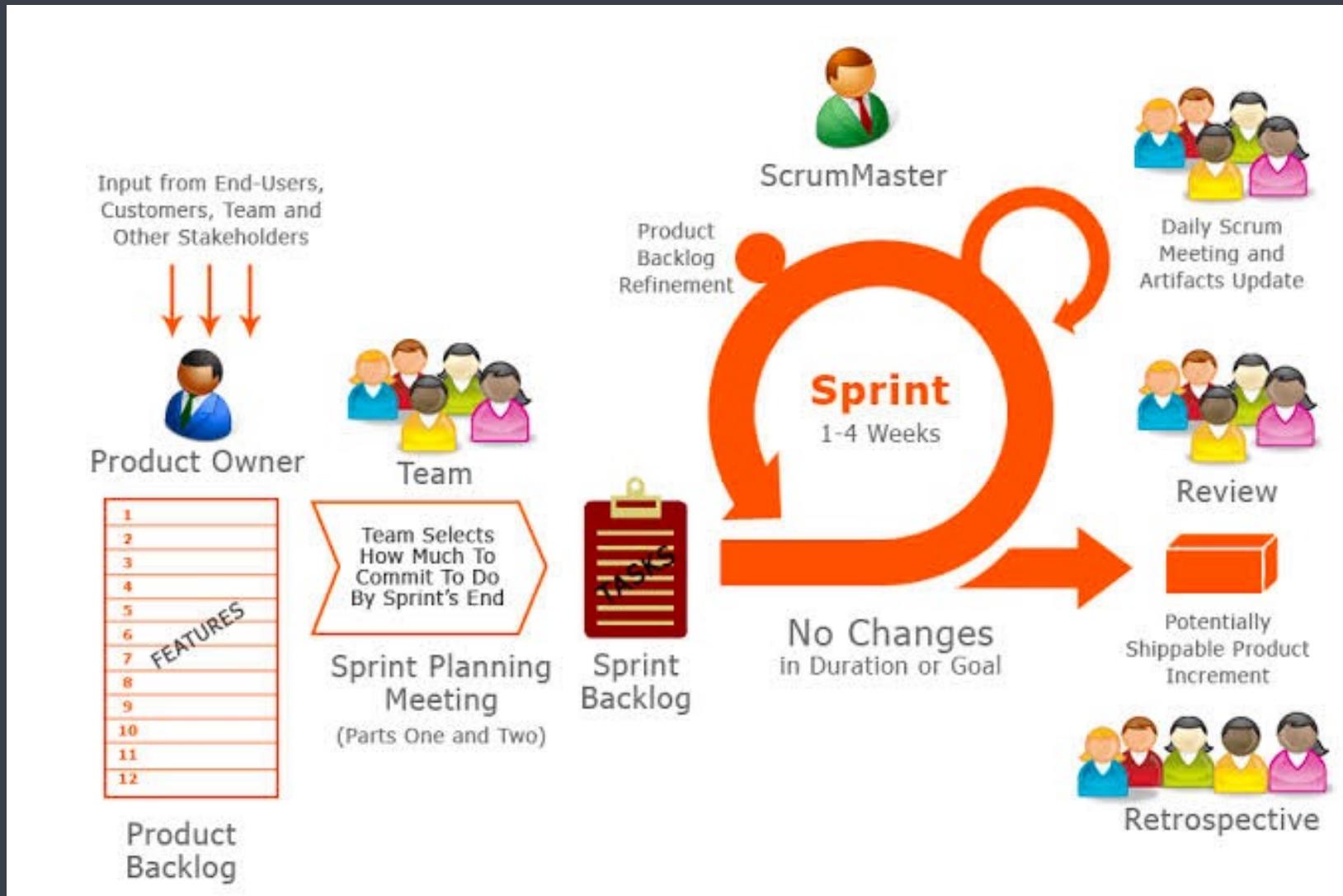


Agile Development





Scrum Process



Waterfall Vs Agile



The Waterfall Process



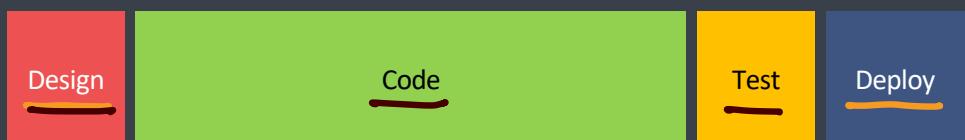
The Agile Process



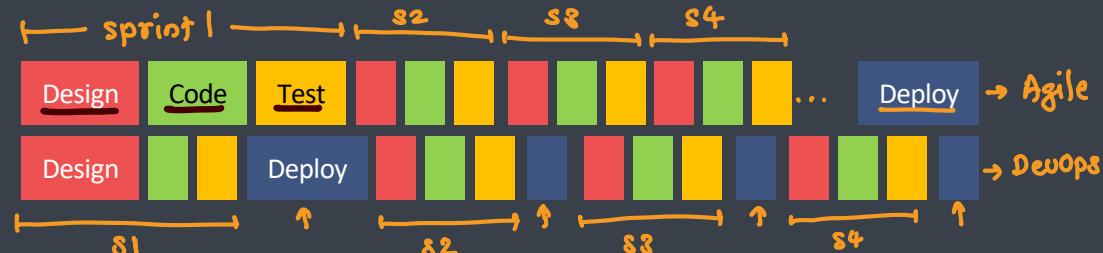
story



This project has got so big.
I am not sure I will be able to deliver it!



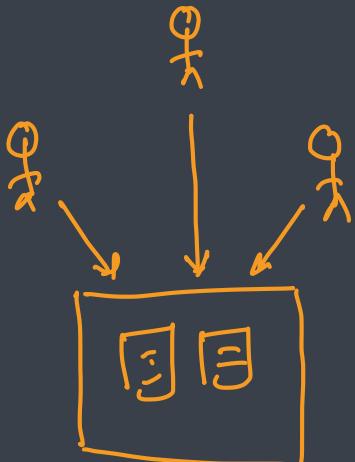
It is so much better delivering
this project in bite-sized sections → stories





Problems

- Managing and tracking changes in the code is difficult
- Incremental builds are difficult to manage, test and deploy
- Manual testing and deployment of various components/modules takes a lot of time
- Ensuring consistency, adaptability and scalability across environments is very difficult task
- Environment dependencies makes the project behave differently in different environments



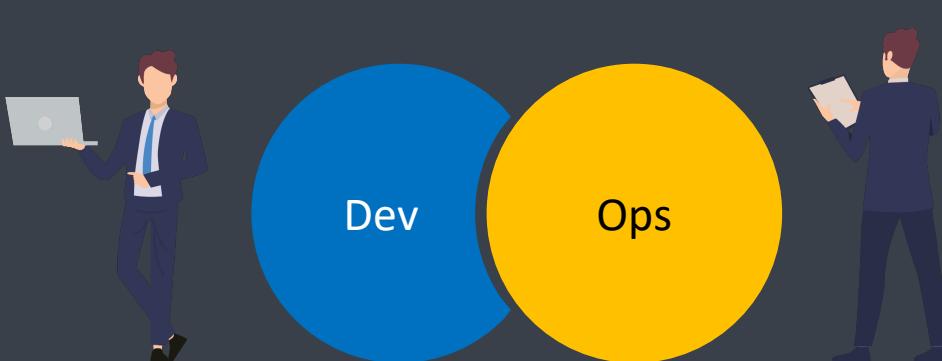


Solutions to the problem

- Managing and tracking changes in the code is difficult: SCM tools → git
- Incremental builds are difficult to manage, test and deploy: Jenkins → CI/CD pipeline
- Manual testing and deployment of various components/modules takes a lot of time: Selenium → automated testing
- Ensuring consistency, adaptability and scalability across environments is very difficult task: Puppet → continuous config
- Environment dependencies makes the project behave differently in different environments: Docker → Containerization

What is DevOps ?

dev testing opg





Why DevOps is Needed?

- Before DevOps, the development and operation team worked in complete isolation
- Testing and Deployment were isolated activities done after design-build. Hence they consumed more time than actual build cycles.
- Without using DevOps, team members are spending a large amount of their time in testing, deploying, and designing instead of building the project.
- Manual code deployment leads to human errors in production
- Coding & operation teams have their separate timelines and are not in sync causing further delays



Common misunderstanding

- DevOps is not a role, person or organization
- DevOps is not a separate team ➔
- DevOps is not a product or a tool
- DevOps is not just writing scripts or implementing tools

mindset of continuous improvement

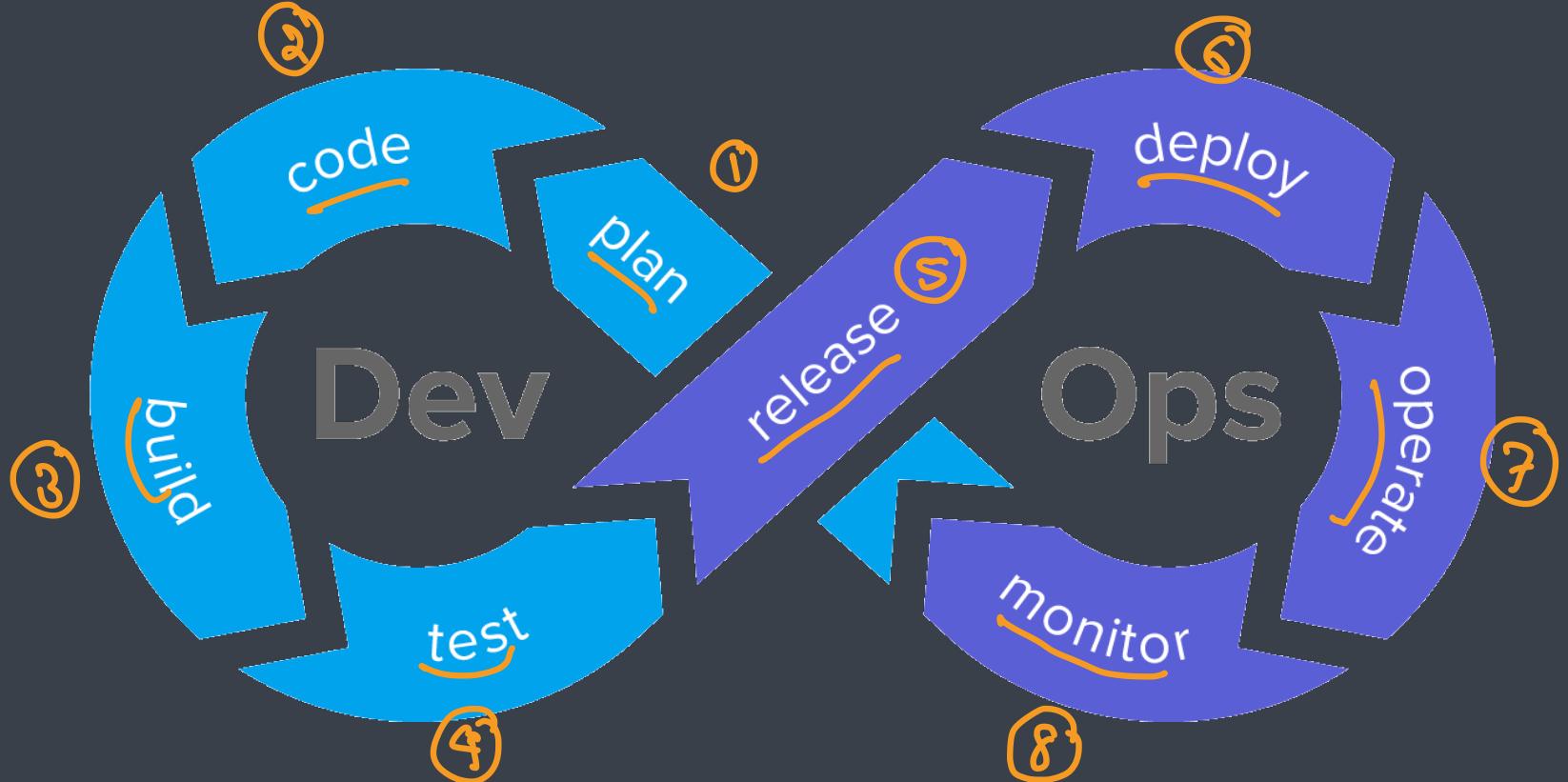


Reasons to use DevOps

- **Predictability**
 - DevOps offers significantly lower failure rate of new releases
- **Reproducibility**
 - Version everything so that earlier version can be restored anytime
- **Maintainability**
 - Effortless process of recovery in the event of a new release crashing or disabling the current system
- **Time to market**
 - DevOps reduces the time to market up to 50% through streamlined software delivery
 - This is particularly the case for digital and mobile applications
- **Greater Quality**
 - DevOps helps the team to provide improved quality of application development as it incorporates infrastructure issues
- **Reduced Risk**
 - DevOps incorporates security aspects in the software delivery lifecycle. It helps in reduction of defects across the lifecycle
- **Resiliency**
 - The Operational state of the software system is more stable, secure, and changes are auditable



DevOps Lifecycle

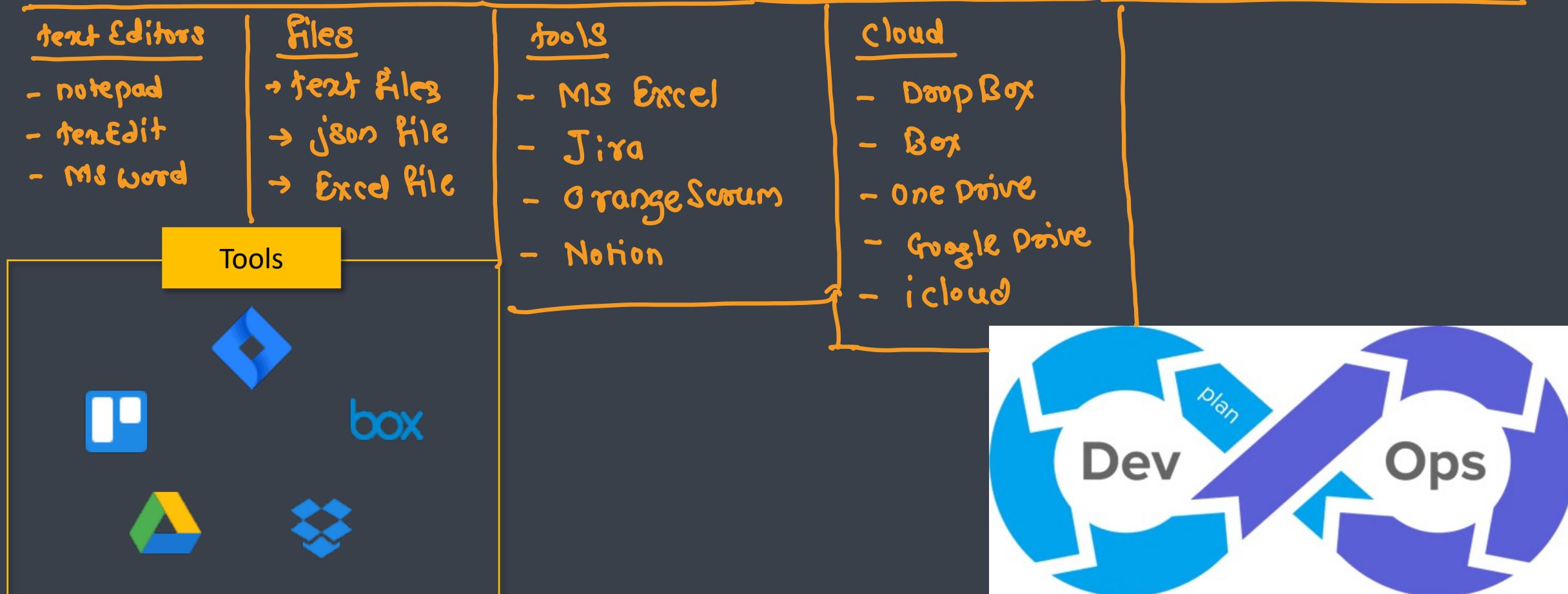


DevOps Lifecycle - Plan a sprint →

- sprint backlog formation
- assign the stories to developers
- track the progress



- First stage of DevOps lifecycle where you plan, track, visualize and summarize your project before you start working on it

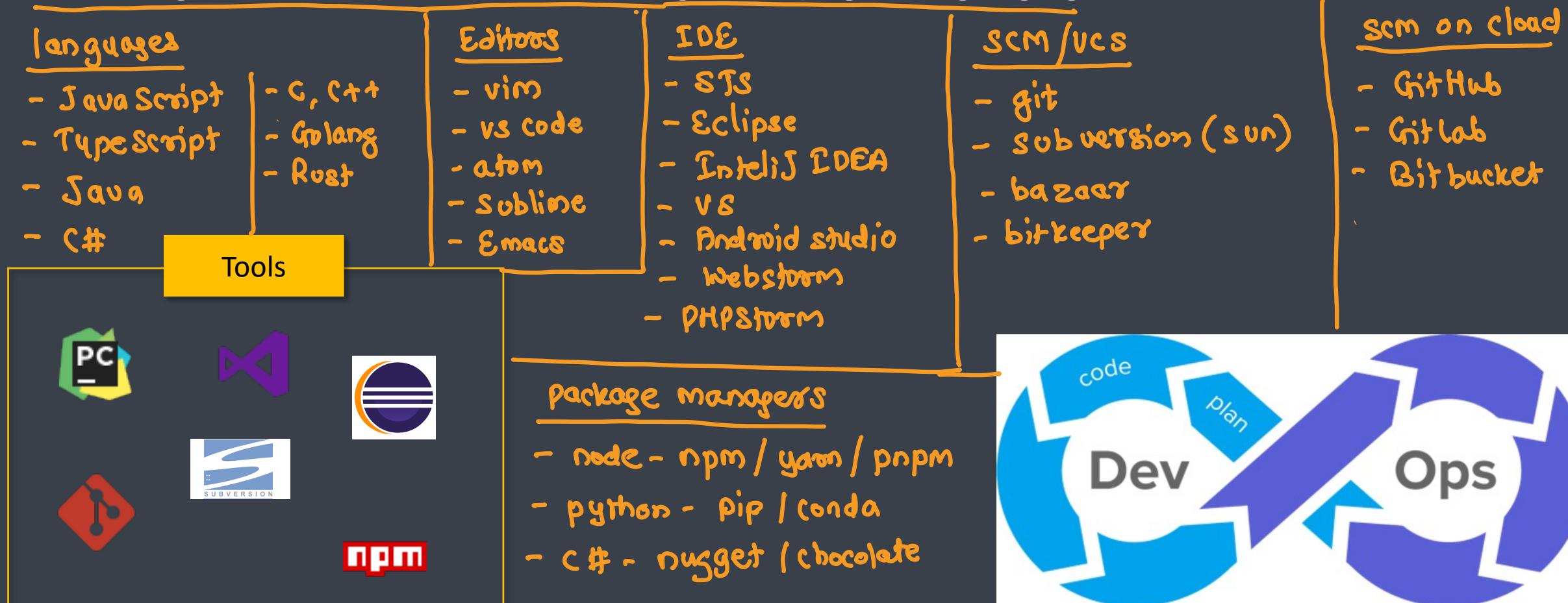


DevOps Lifecycle - Code → development of product

SATA → Serial ATA
ATA → AT Attached
AT → Advanced Technology

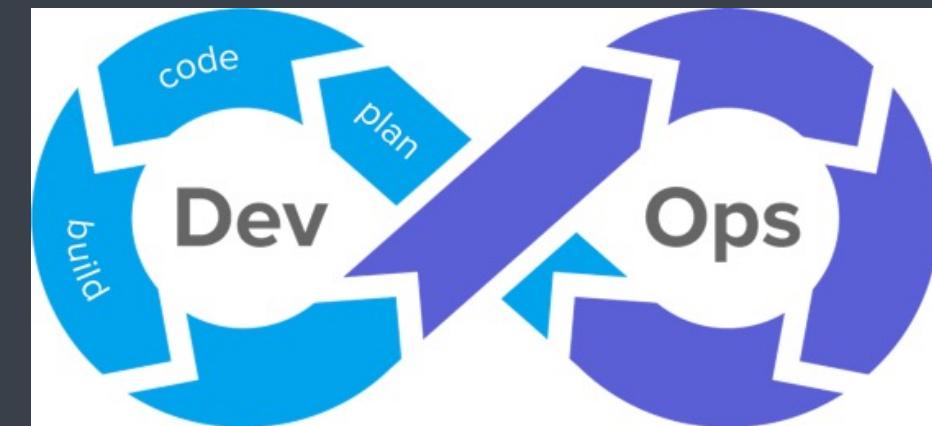
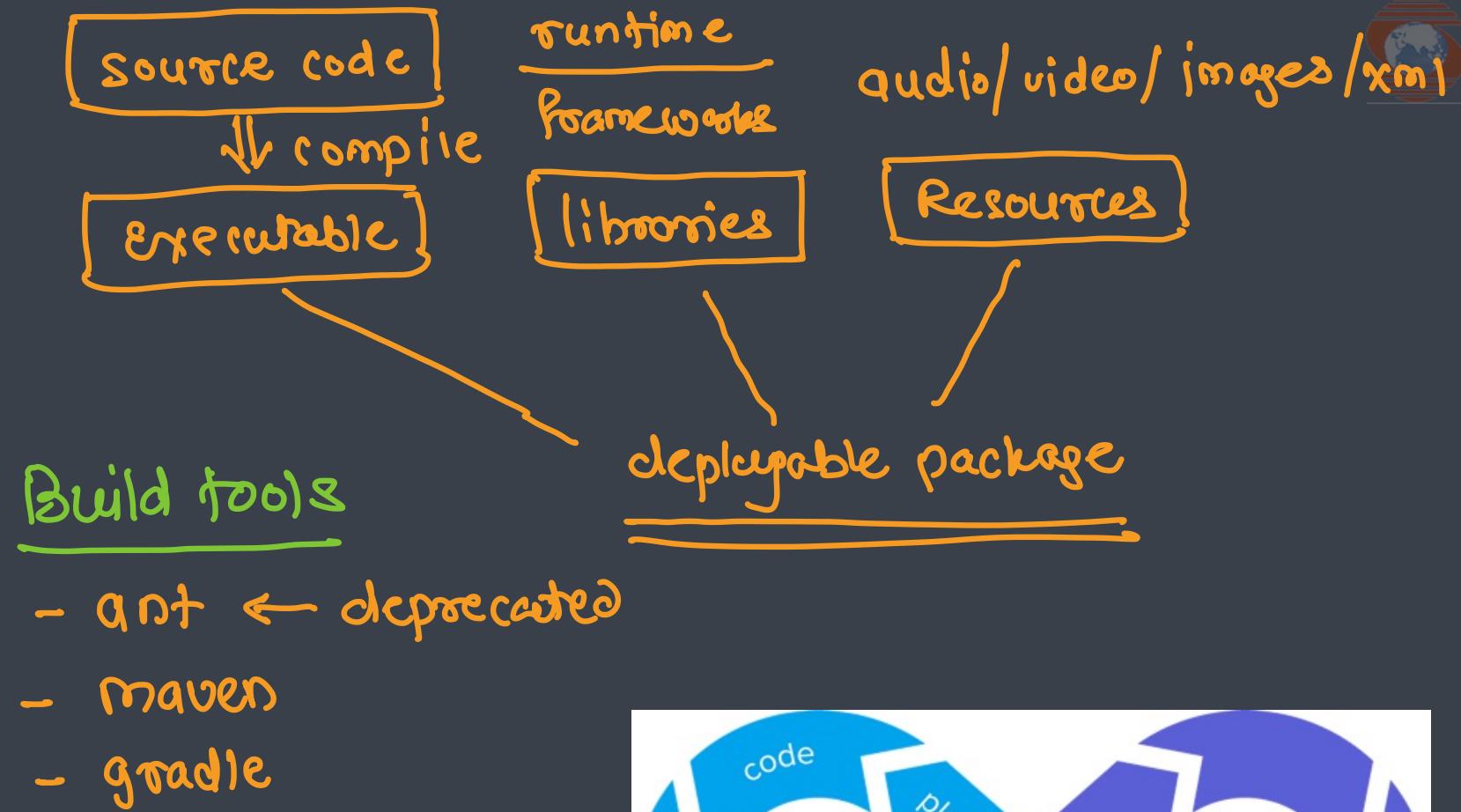
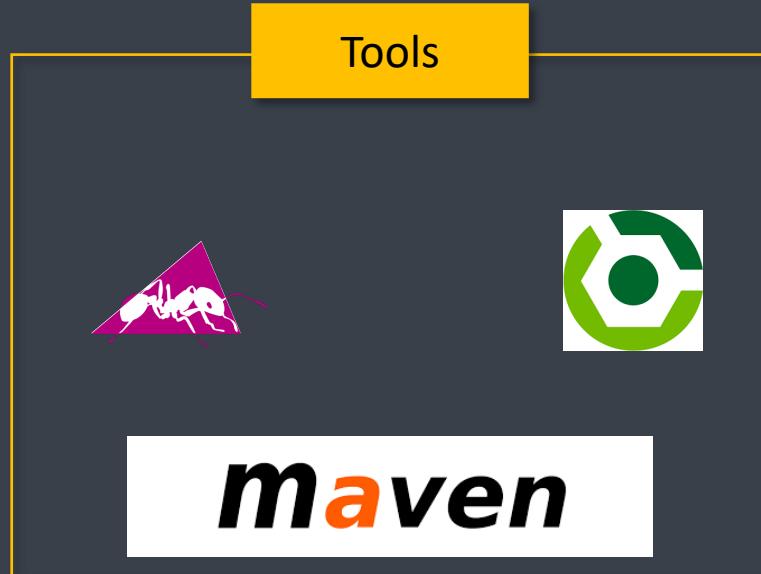


- Second stage where developer writes the code using favorite programming language



DevOps Lifecycle -Build

- Integrating the required libraries
- Compiling the source code
- Create deployable packages





DevOps Lifecycle - Test → automated testing

- Process of executing automated tests
- The goal here is to get the feedback about the changes as quickly as possible

unit testing

- python - pyunit
- java - JUnit
- C# - Nunit
- JS → Karma / Jasmine

Tools



Non-functional testing

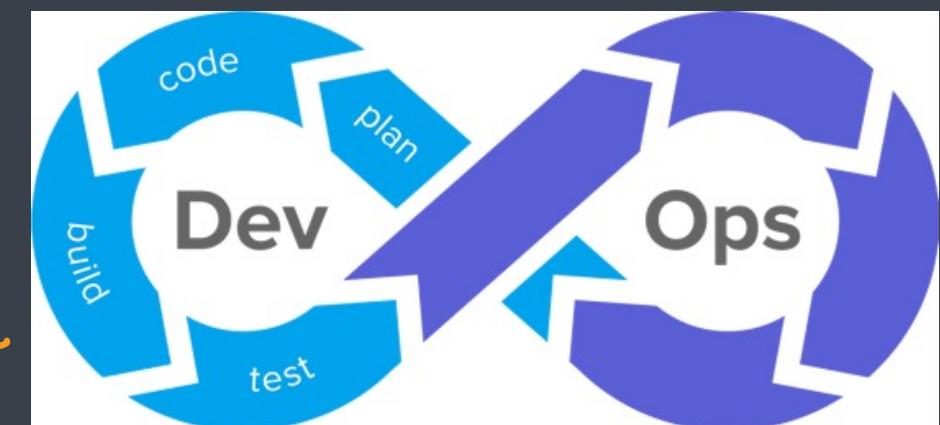
- load runner
- stress runner
- Jmeter

mobile testing

- apium
- browserstack

website GUI testing

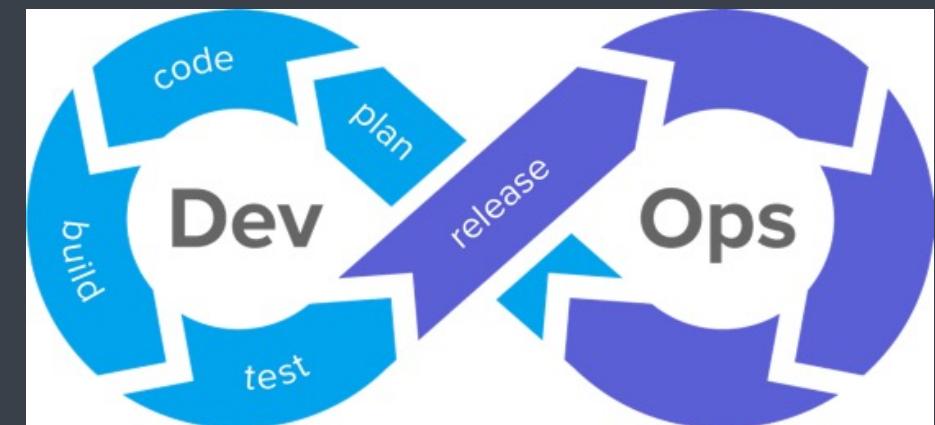
- Selenium
- TestNG
- Cypress





DevOps Lifecycle - Release → CI / CD pipeline

- This phase helps to integrate code into a shared repository using which you can detect and locate errors quickly and easily





DevOps Lifecycle - Deploy

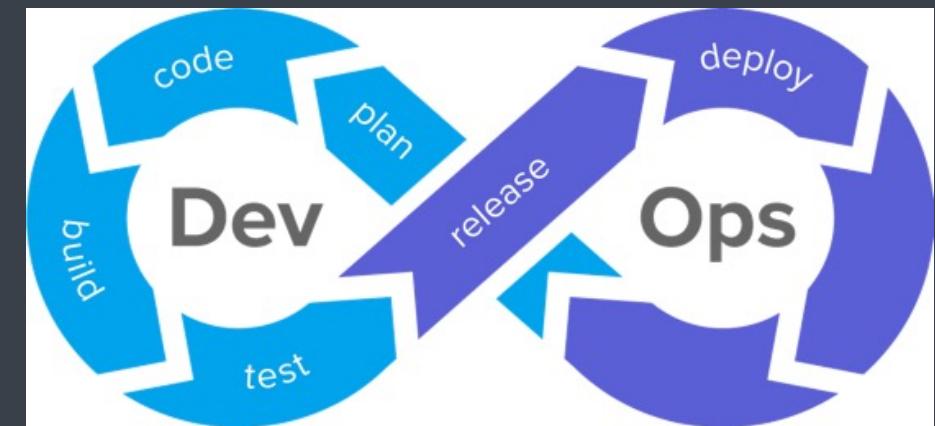
- Manage and maintain development and deployment of software systems and server in any computational environment
- deployment

→ traditional → using physical machines X
→ virtualized → using VM → VMware / VirtualBox / Hyper-V / Parallels
→ containerized → using Docker, podman, LXC, LXD, containerd



→ container orchestration

- Docker Swarm
- Kubernetes
- Apache Marathon
- Apache Mesos





DevOps Lifecycle - Operate → Environment Creation and configuration

- This stage where the updated system gets operated

Environment creation tools

- Terraform
- vagrant
- AWS Cloud Formation

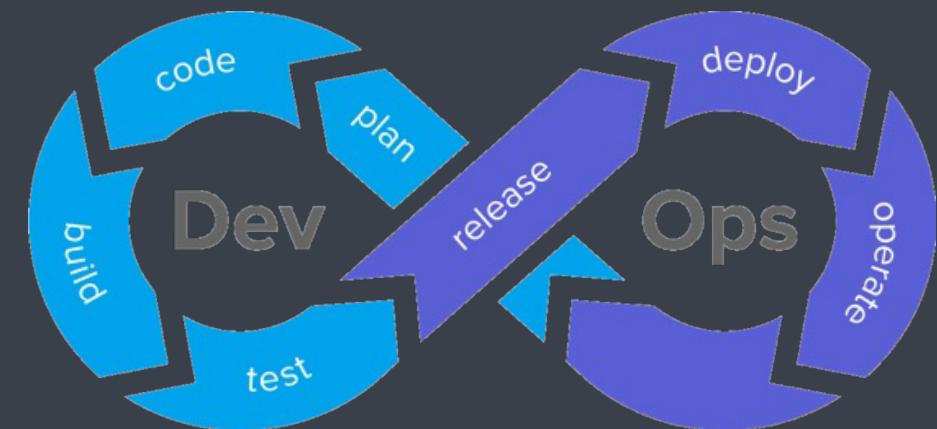
Tools

A



Environment configuration tools

- Ansible
- Puppet
- Chef
- SaltStack



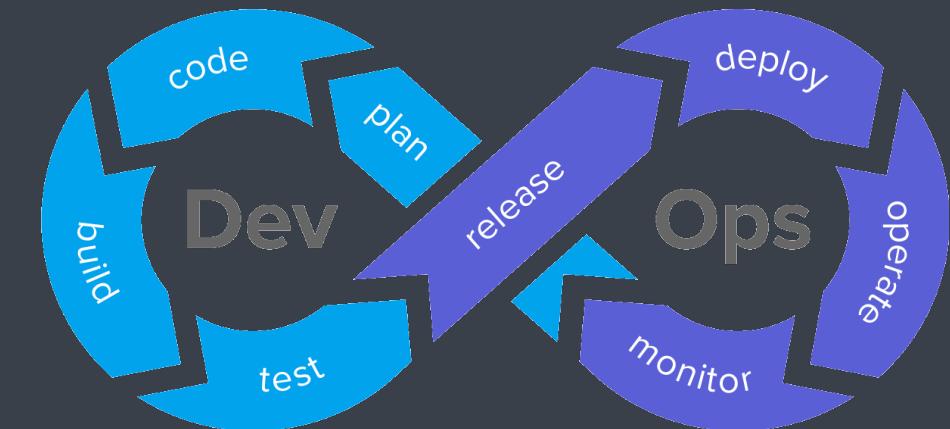


DevOps Lifecycle - Monitor → monitor application

- It ensures that the application is performing as expected and the environment is stable
- It quickly determines when a service is unavailable and understand the underlying causes

tools

- datadog
- nagios
- splunk

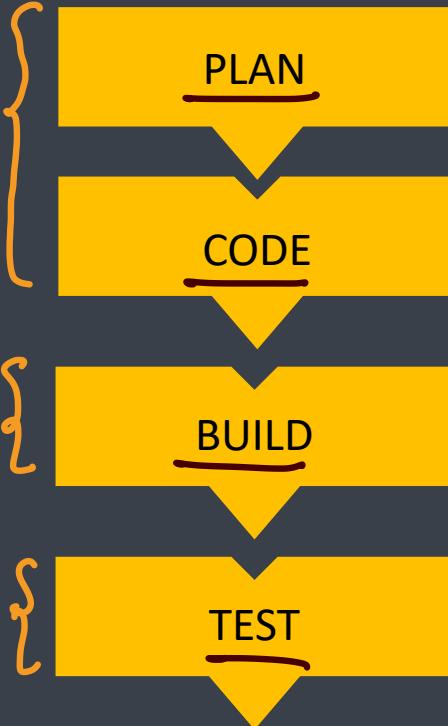


- plan → Jira
- code → VS code
- build → gradle
- test → selenium
- release → Jenkins
 - deploy → docker + k8s
- operate → ansible
- monitor → nagios



DevOps Terminologies

continuous coding



continuous building



continuous testing



Continuous Learning

continuous integration



continuous delivery



continuous deployment

continuous configuration

continuous monitoring



Responsibilities of DevOps Engineer

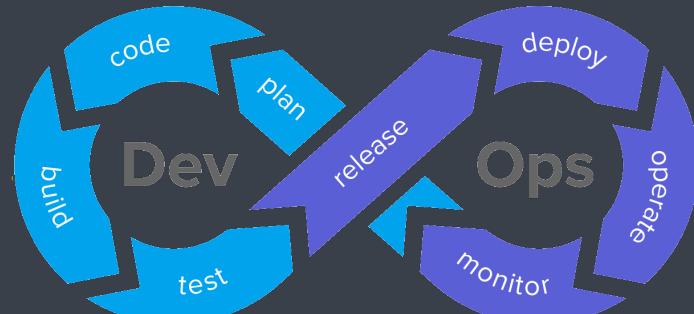
Linux

Be an excellent sysadmin

Deploy Virtualization

Hands-on experience in
network and storage

Introduction to coding



Soft skills

Automation tools

Software Testing
knowledge

IT security

Skills of a DevOps Engineer



Skills	Description
Tools	<ul style="list-style-type: none">• Version Control – Git/SVN• Continuous Integration – Jenkins• Virtualization / Containerization – Docker/Kubernetes• Configuration Management – Puppet/Chef/Ansible• Monitoring – Nagios/Splunk
Network Skills	<ul style="list-style-type: none">• General Networking Skills• Maintaining connections/Port Forwarding
Other Skills	<ul style="list-style-type: none">• Cloud: AWS/Azure/GCP• Soft Skills• People management skill

git

basic workflow

```
# initialize an empty repository
> git init

# get the current status of every file present in working directory
> git status

# get short status
> git status -s

# note: git status -s command returns a status with two characters
# 1st character: shows the status of file with respect to the staging area
# 2nd character: shows the status of file with respect to the working area

# ???: untracked file (the repository does not recognize the file or does
not have any version of this file created yet)
# A : the changes are present in the staging area and will be added to the
repository after committing
# M: the changes are present in the working directory and the file is
modified in the working directory
# M : the changes are moved to the staging area
# UU: the file has conflicts

# add the file/files to staging area
> git add <file name>
> git add .

# create a version of all the files present in staging area
> git commit -m <message>

# get the list of all commits
> git log

# get the list of logs in one line with color and graph
> git log --oneline --graph --color

# get the difference between the latest version and the previous version
> git diff

# get the last version from repository and replace it with current version
# note: even if the file is deleted from the working directory, you can
still bring it (last version of it) back from the repository
> git checkout <file name>

# soft reset:
# - move all the changes from staging area to the working directory
```

```

# - no changes will be lost in the process
> git reset

# hard reset:
# - remove all the changes from staging area or working directory
# - get last version of all the files and replace with current version
# (irrespective of their location)
# - note: please execute this command on your own risk
> git reset --hard

# remove all the metadata or repository
# note: this command will remove all the history
> rm -rf .git

```

branches

- reference to a commit in the history

```

# get the list of branches
# note: the branch which has * in front of the name, is the current branch
> git branch

# to find out the current branch, you can use command git status and look
for the first line to get the branch name

# git uses HEAD to find the current branch
# HEAD: reference to the current branch

# create a new branch
# note: this command does not switch to the newly created branch
> git branch <branch name>

# switch to another branch
> git checkout <branch name>

# create a new branch and switch to it immediately
> git checkout -b <branch name>

# merge a branch in another branch
# (merging a feature branch)

# example: merging a branch named prime-number in main branch (source)
# step1: checkout the source branch (the branch in which you want to merge
another branch)
> git checkout main

# step2: merge the changes from another branch to the source branch
# > git merge <branch name>
> git merge prime-number

```

```
# delete a branch
# note: you CAN NOT delete the current branch
> git branch -d <branch name>

# conflict scenario
# – when a file gets modified by two branches on the same line(s), git
does not understand which changes to keep and which ones to remove, this
scenario is called as a conflict scenario
# – conflict scenario CAN NOT be resolved automatically
# – conflict scenario MUST be resolved manually
# – conflicted file(s) will have a conflict marker (>>>>> or <<<<<<)
# – resolving a conflict means, removing the conflict markers
# – once the conflicts are resolved, committing a new version is mandatory
```

shared repository

```
# types
# – local
#   – present on local machine
# – shared
#   – present on the server
#   – also known as remote repository
#   – to create a shared repository
#     – create a project or repository on shared repository server
#     – connect the local repository to the remote one
#   – e.g. GitHub, GitLab, BitBucket

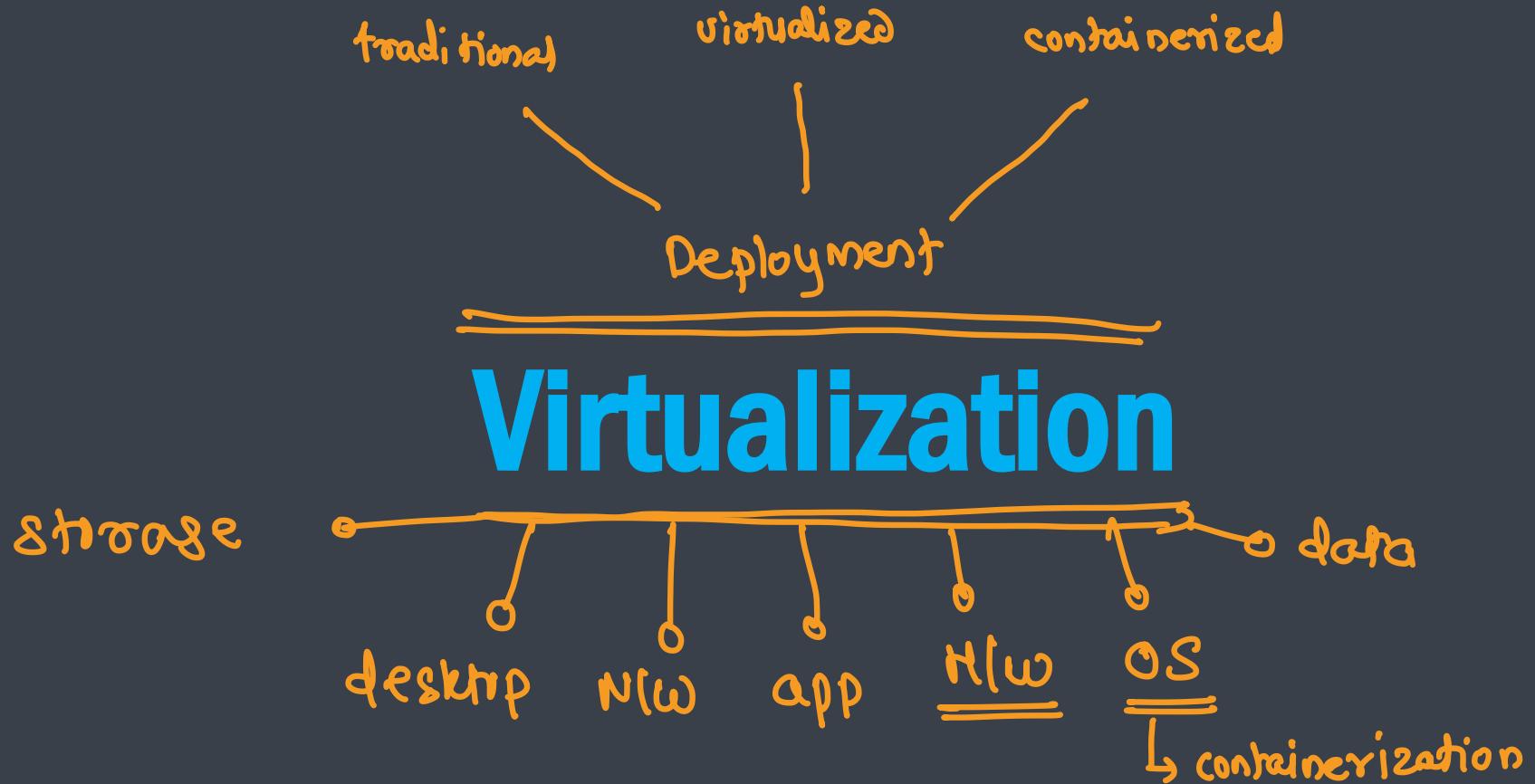
# get the remote repository url
> git remote -v

# add the remote repository
# > git remote add <server alias> <repository url>
> git remote add origin <repository>

# send all the local changes to the remote repository
> git push <alias> <branch name>

# pull all the changes from remote to local repository
> git pull <alias> <branch name>

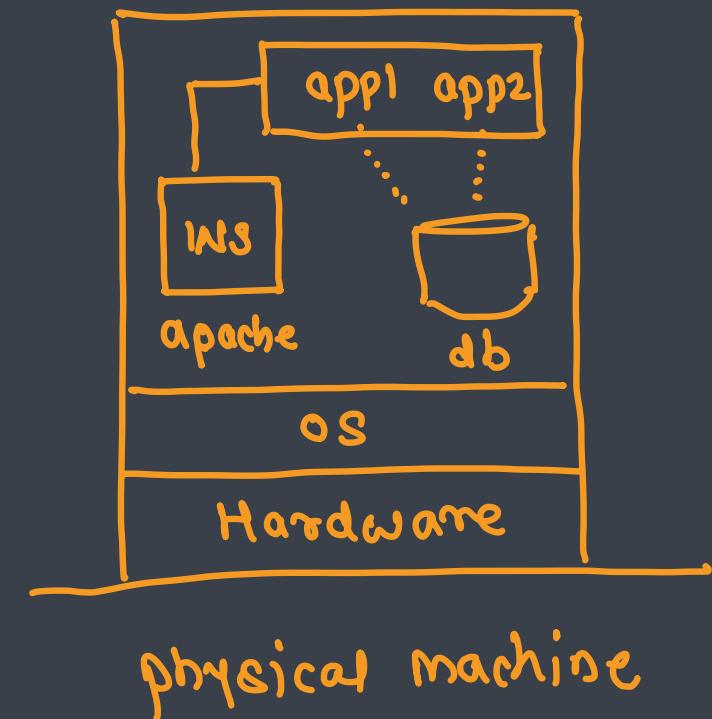
# remove the remote repository from local one
> git remote remove <alias>
```

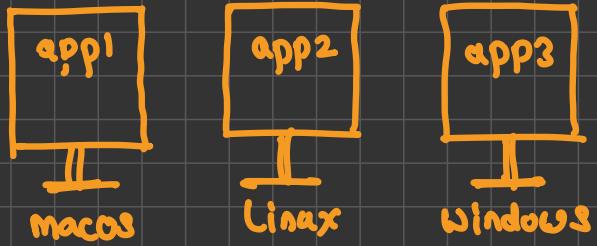


Traditional Deployment → deprecated

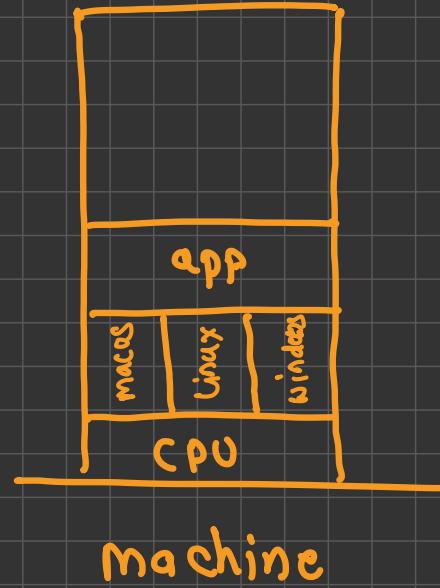
- Early on, organizations ran applications on physical servers
- There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues
- For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform
- A solution for this would be to run each application on a different physical server
- But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers

resource → CPU + memory





+ fastest option
- costliest



multi-booting machine

+ cheapest
- time [one os can run at a time]

What is virtualization

→ No physical existence

+ faster
+ run multiple OSes simultaneously
+ can be easily shared

- Virtualization is the creation of a virtual -- rather than actual -- version of something, such as an operating system (OS), a server, a storage device or network resources

- Virtualization uses software that simulates hardware functionality in order to create a virtual system

→ How virtualization

- This practice allows IT organizations to operate multiple operating systems, more than one virtual system and various applications on a single server

- Types

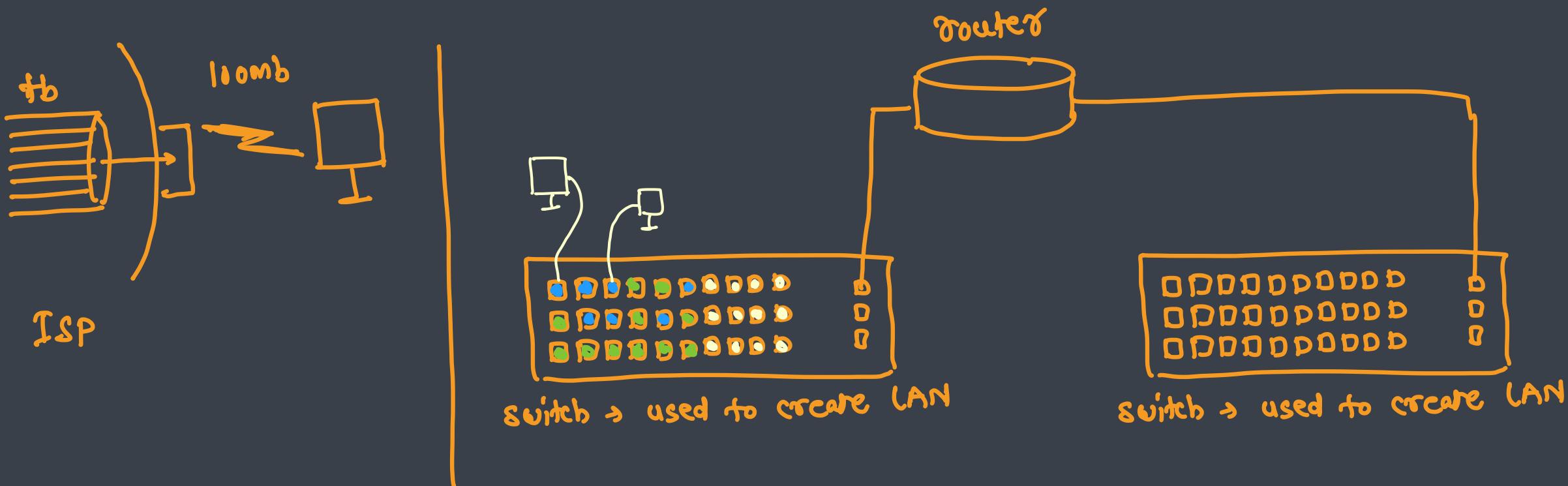
- ① ▪ Network virtualization
- ② ▪ Storage virtualization
- ③ ▪ Data virtualization
- ④ ▪ Desktop virtualization
- ⑤ ▪ Application virtualization
- ⑥ ▪ Hardware virtualization

- ⑦ ▪ OS virtualization

(containerization) * * *

Network Virtualization

- Network virtualization takes the available resources on a network and breaks the bandwidth into discrete channels
- Admins can secure each channel separately, and they can assign and reassign channels to specific devices in real time
- The promise of network virtualization is to improve networks' speed, availability and security, and it's particularly useful for networks that must support unpredictable usage bursts

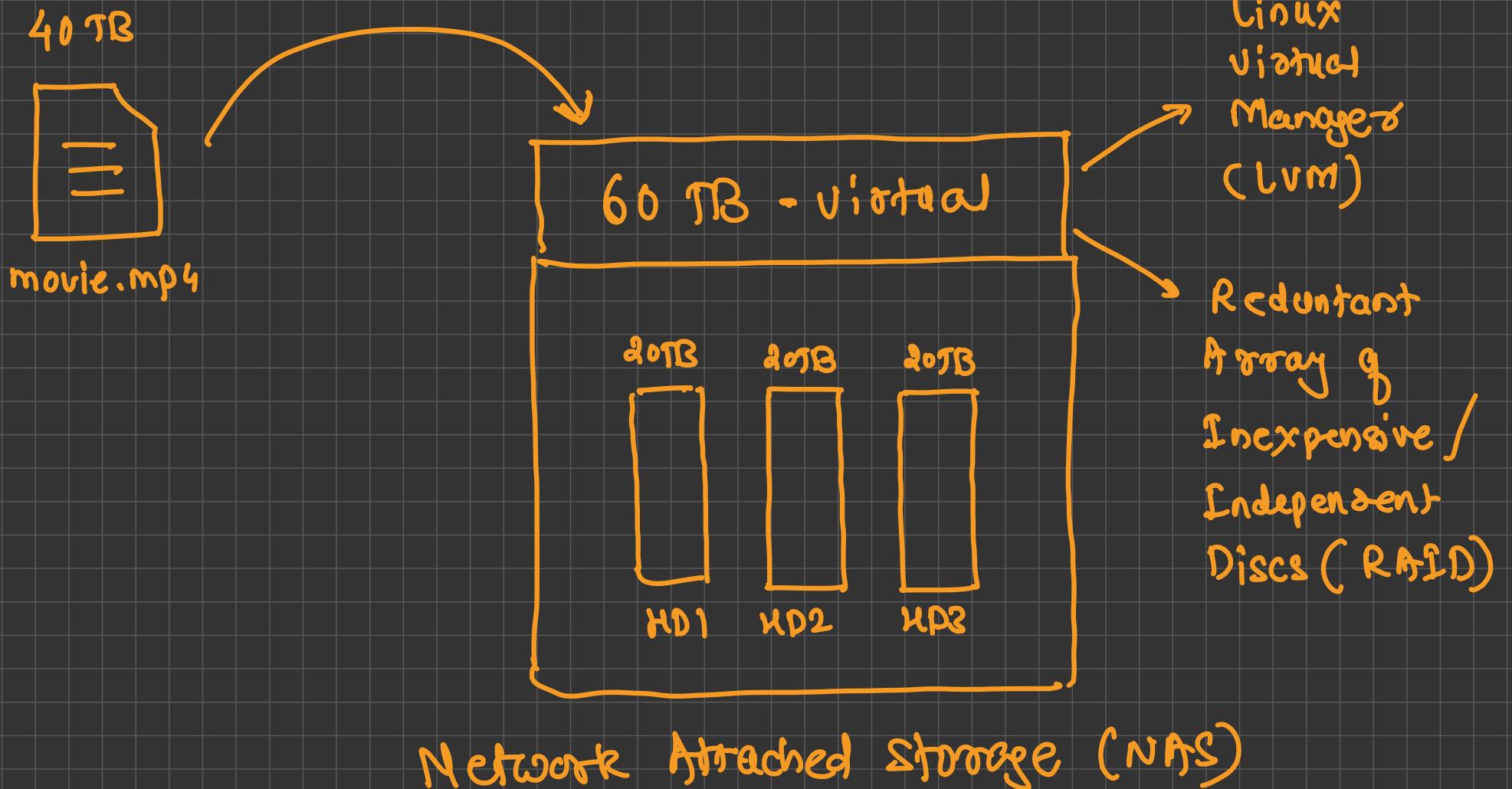


Storage Virtualization

- Storage virtualization is the pooling of physical storage from multiple network storage devices into what appears to be a single storage device that is managed from a central console
- Storage virtualization is commonly used in storage area networks
- Applications can use storage without having any concern for where it resides, what technical interface it provides, how it has been implemented, which platform it uses and how much of it is available
- Benefits
 - Makes the remote storage devices appear local
 - Multiple smaller volumes appear as a single large volume
 - Data is spread over multiple physical disks to improve reliability and performance
 - All operating systems use the same storage device
 - Provided high availability, disaster recovery, improved performance and sharing

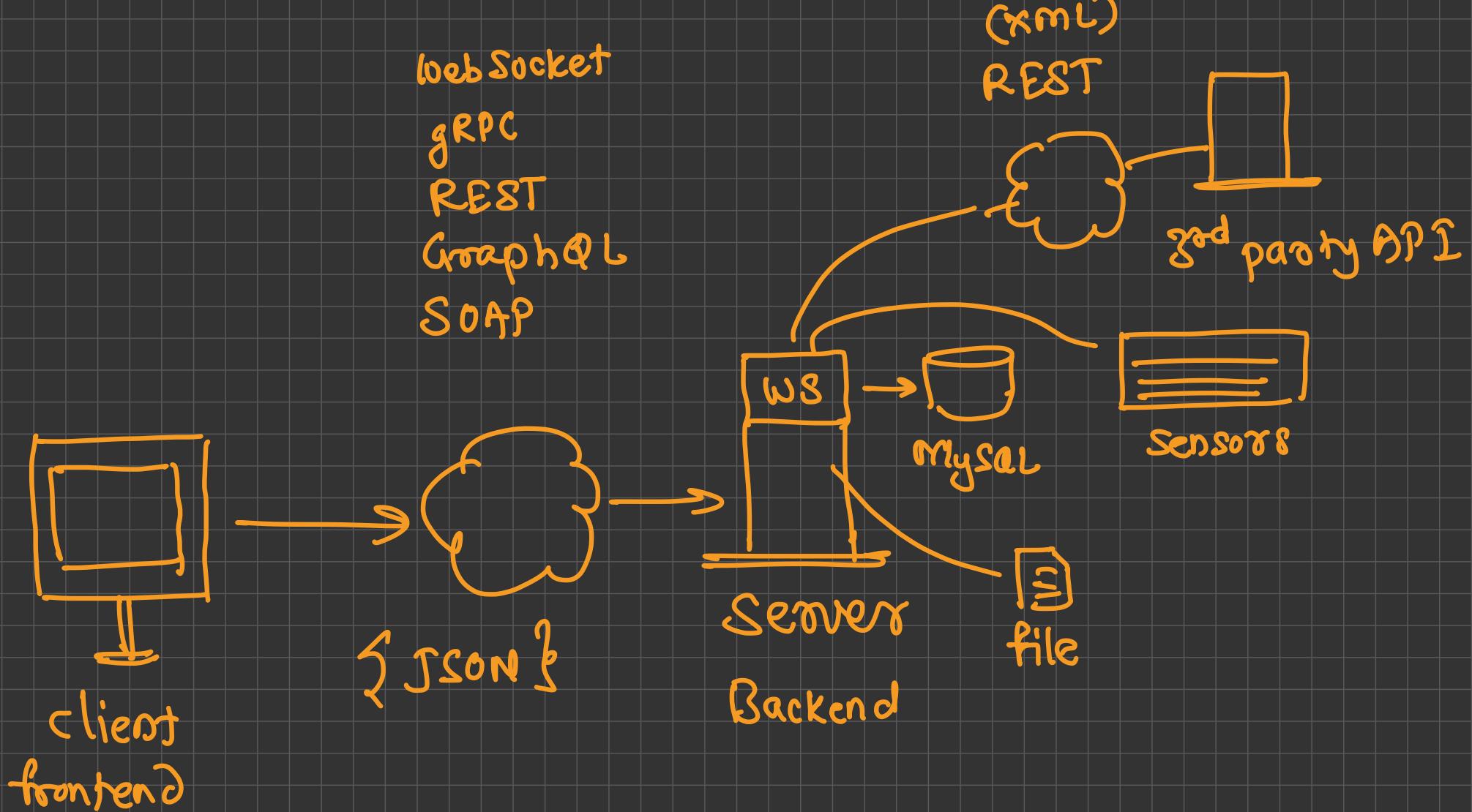


Storage virtualization



Data virtualization

- Data virtualization is the process of **aggregating** data from different sources of information to develop a single, logical and virtual view of information so that it can be accessed by front-end solutions such as applications, dashboards and portals without having to know the data's exact storage location
- The process of data virtualization involves abstracting, transforming, federating and delivering data from disparate sources
- The main goal of data virtualization technology is to provide a single point of access to the data by aggregating it from a wide range of data sources
- Benefits
 - Abstraction of technical aspects of stored data like APIs, Language, Location, Storage structure
 - Provides an ability to connect multiple data sources from a single location
 - Provides an ability to combine the data result sets across multiple sources (also known as data federation)
 - Provides an ability to deliver the data as requested by users

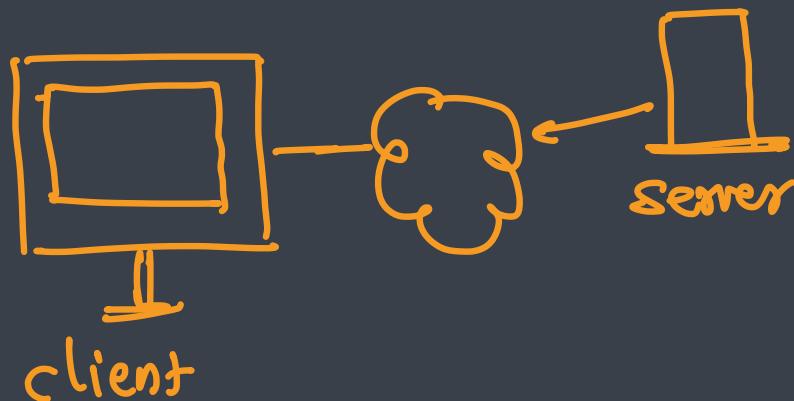


Desktop virtualization → Anydesk

- With desktop virtualization, the goal is to isolate a desktop OS from the endpoint that employees use to access it
- It provides an ability to connect to the desktop from remote site
- When multiple users connect to a shared desktop, as is the case with Microsoft Remote Desktop Services, it's known as shared hosted desktop virtualization

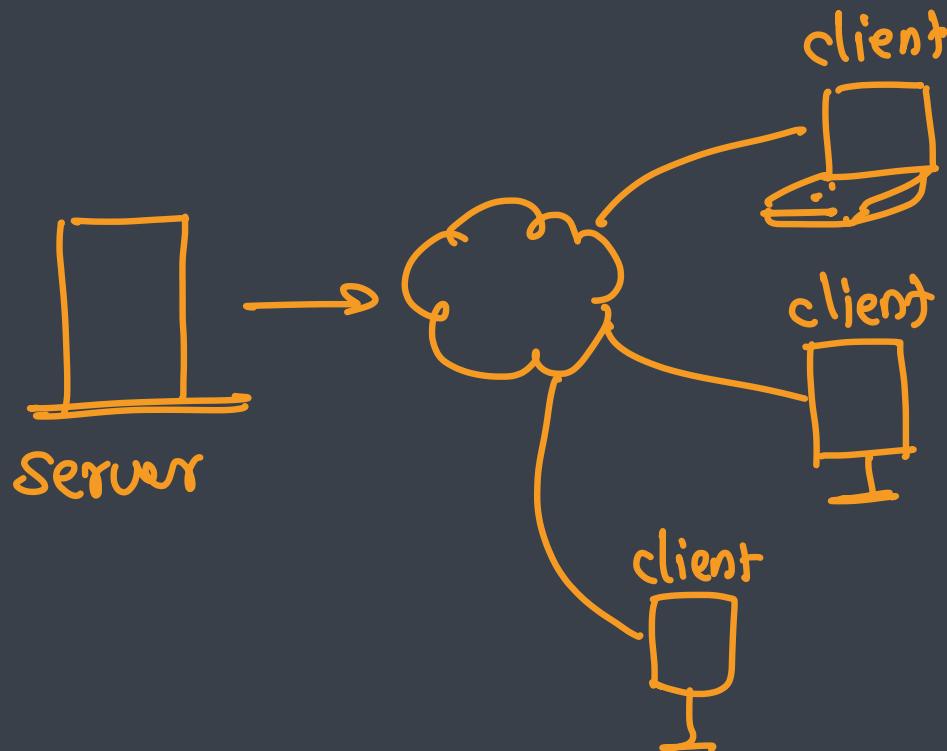
UNC → virtual Network Connection

RDC → Remote Desktop Connection



Application Virtualization

- With application virtualization, an app runs separately from the device that accesses it
- Application virtualization makes it possible for IT admins to install, patch and update only one version of an app rather than performing the same management tasks multiple times

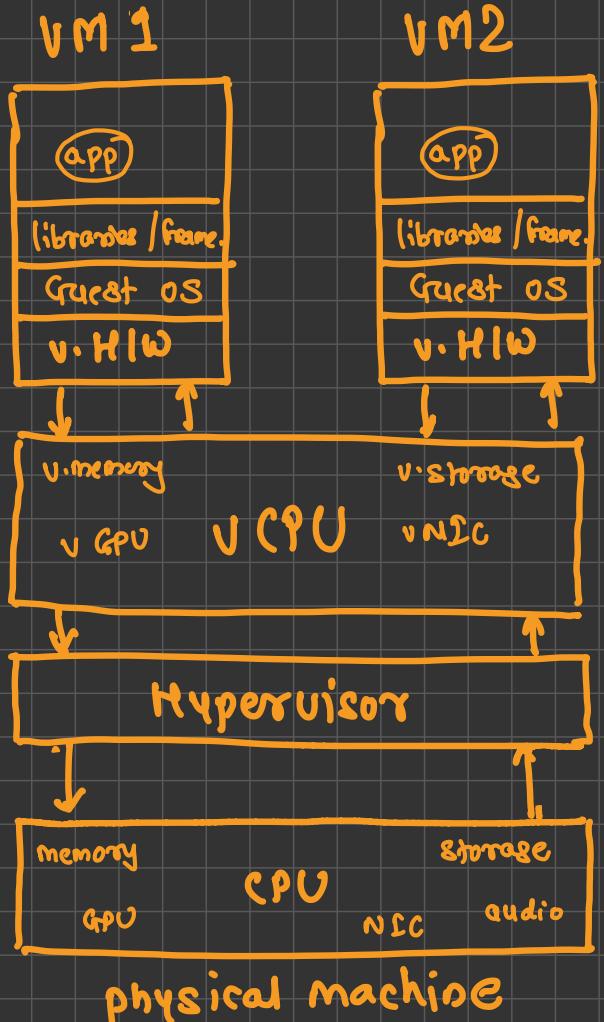


Hardware Virtualization

- Hardware virtualization or platform virtualization refers to the creation of a virtual machine that acts like a real computer with an operating system
- The process of masking the hardware resources like
 - CPU
 - Storage
 - Memory
- For example, a computer that is running Microsoft Windows may host a virtual machine that looks like a computer with the Ubuntu Linux operating system; Ubuntu-based software can be run on the virtual machine
- The process of creating Machines

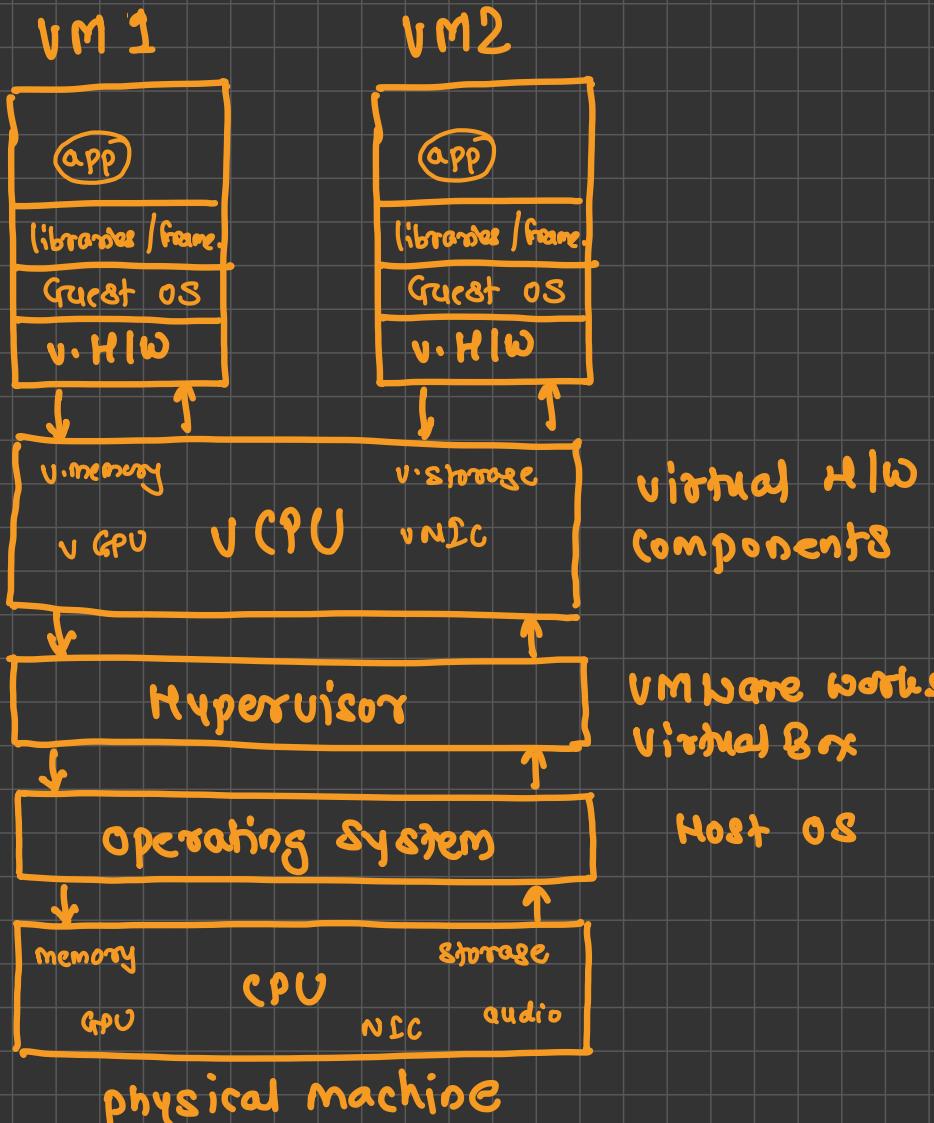
Type I → Bare Metal Hypervisor

- Does Not require Host OS
- faster and utilizes resources in better ways compared with Type II
- used by cloud providers



Type II → Hosted hypervisor

- requires host OS
- requires more resources
- used by developers / testers

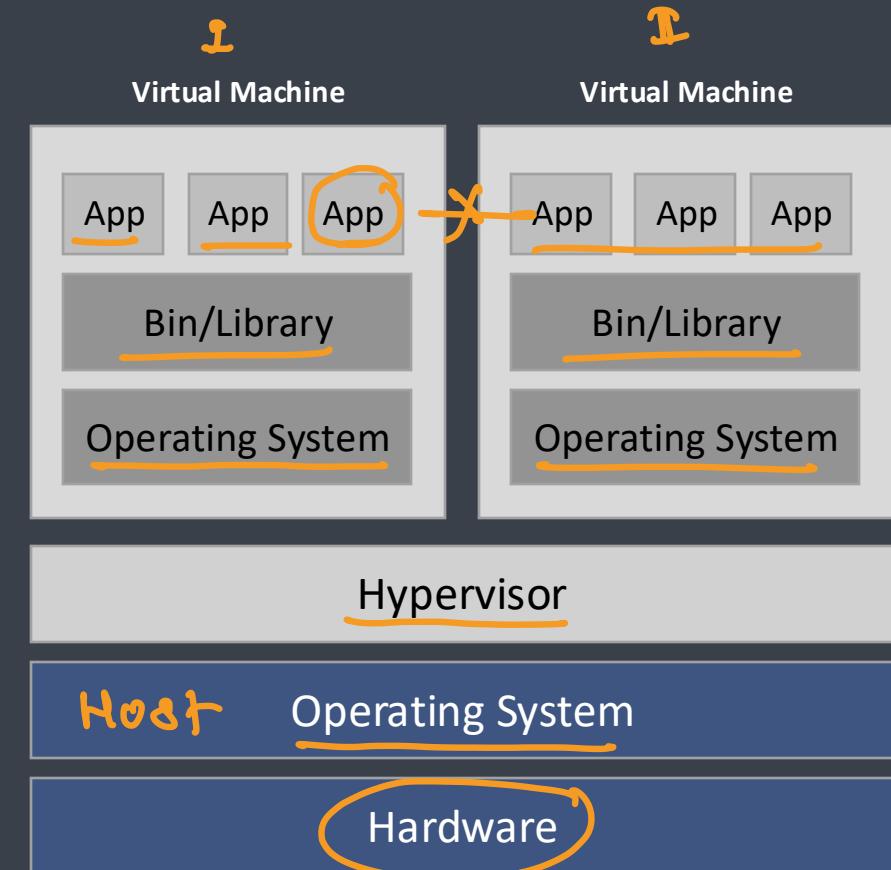


Virtual Machine

- A virtual machine is the emulated equivalent of a computer system that runs on top of another system
- Virtual machines may have access to any number of resources
 - Computing power - through hardware-assisted but limited access to the host machine's CPU
 - Memory - one or more physical or virtual disk devices for storage
 - A virtual or real network interfaces
 - Any devices such as
 - video cards,
 - USB devices,
 - other hardware that are shared with the virtual machine
- If the virtual machine is stored on a virtual disk, this is often referred to as a **disk image**

Virtualized Deployment

- It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU
- Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application
- Virtualization allows better utilization of resources in a physical server and allows better scalability because
 - an application can be added or updated easily
 - reduces hardware costs
- With virtualization you can present a set of physical resources as a cluster of disposable virtual machines
- Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware



Types of hardware virtualization

- **Type I**

- A Type 1 hypervisor runs directly on the host machine's physical hardware, and it's referred to as a bare-metal hypervisor
- It doesn't have to load an underlying OS first
- With direct access to the underlying hardware and no other software, it is more efficient and provides better performance
- It is best suited for enterprise computing or data centers
- E.g. VMware ESXi, Microsoft Hyper-V server and open source KVM

- **Type II**

- A Type 2 hypervisor is typically installed on top of an existing OS, and it's called a hosted hypervisor
- It relies on the host machine's pre-existing OS to manage calls to CPU, memory, storage and network resources
- E.g. VMware Fusion, Oracle VM VirtualBox, Oracle VM Server for x86, Oracle Solaris Zones, Parallels and VMware Workstation

Advantages of virtualization

- **Lower costs**

- Virtualization reduces the amount of hardware servers necessary within a company and data center
- This lowers the overall cost of buying and maintaining large amounts of hardware

- **Easier disaster recovery**

- Disaster recovery is very simple in a virtualized environment
- Regular snapshots provide up-to-date data, allowing virtual machines to be feasibly backed up and recovered
- Even in an emergency, a virtual machine can be migrated to a new location within minutes

- **Easier testing**

- Testing is less complicated in a virtual environment
- Even if a large mistake is made, the test does not need to stop and go back to the beginning
- It can simply return to the previous snapshot and proceed with the test.

- **Quicker backups**

- Backups can be taken of both the virtual server and the virtual machine
- Automatic snapshots are taken throughout the day to guarantee that all data is up-to-date
- Furthermore, the virtual machines can be easily migrated between each other and efficiently redeployed

- **Improved productivity**

- Fewer physical resources results in less time spent managing and maintaining the servers
- Tasks that can take days or weeks in a physical environment can be done in minutes
- This allows staff members to spend the majority of their time on more productive tasks, such as raising revenue and fostering business initiatives

Advantages of virtualization

- **Quicker backups**

- Backups can be taken of both the virtual server and the virtual machine
- Automatic snapshots are taken throughout the day to guarantee that all data is up-to-date
- Furthermore, the virtual machines can be easily migrated between each other and efficiently redeployed

- **Improved productivity**

- Fewer physical resources results in less time spent managing and maintaining the servers
- Tasks that can take days or weeks in a physical environment can be done in minutes
- This allows staff members to spend the majority of their time on more productive tasks, such as raising revenue and fostering business initiatives

frontend and backend

Monolithic Architecture

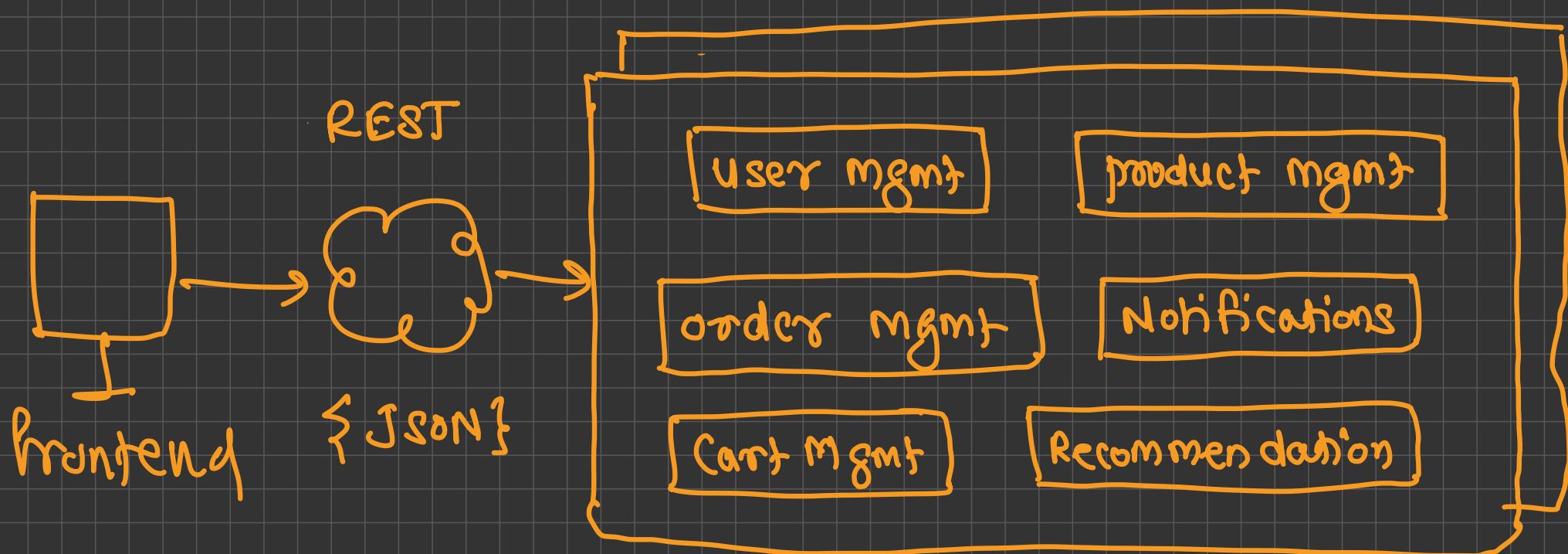
↓
single application with all requirements

E-commerce

- user mgmt
- product mgmt
- order mgmt
- notifications
- recommendation
- cart mgmt

* Technical Stack

- programming language - JavaScript
- frame work - express / fastify / Hapi
- Database - RDBMS → MySQL
- Git Repository - Ecommerce-backend



What is Monolithic Architecture ?

single has all functionality

- A monolithic architecture is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications.
- The word “monolith” is often attributed to something large and glacial, which isn’t far from the truth of a monolith architecture for software design
→ single language
- A monolithic architecture is a singular, large computing network with one code base that couples all of the business concerns together
- To make a change to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface
- Monoliths can be convenient early on in a project's life for ease of code management, cognitive overhead, and deployment
- This allows everything in the monolith to be released at once.

Pros

- **Easy deployment**
 - One executable file or directory makes deployment easier
- **Development**
 - When an application is built with one code base, it is easier to develop
- **Performance**
 - In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices
- **Simplified testing**
 - Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application
- **Easy debugging**
 - With all code located in one place, it's easier to follow a request and find an issue.

Cons

- **Slower development speed**
 - A large, monolithic application makes development more complex and slower
- **Scalability**
 - You can't scale individual components
- **Reliability**
 - If there's an error in any module, it could affect the entire application's availability
- **Barrier to technology adoption**
 - Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming
- **Lack of flexibility**
 - A monolith is constrained by the technologies already used in the monolith
- **Deployment**
 - A small change to a monolithic application requires the redeployment of the entire monolith

Microservices

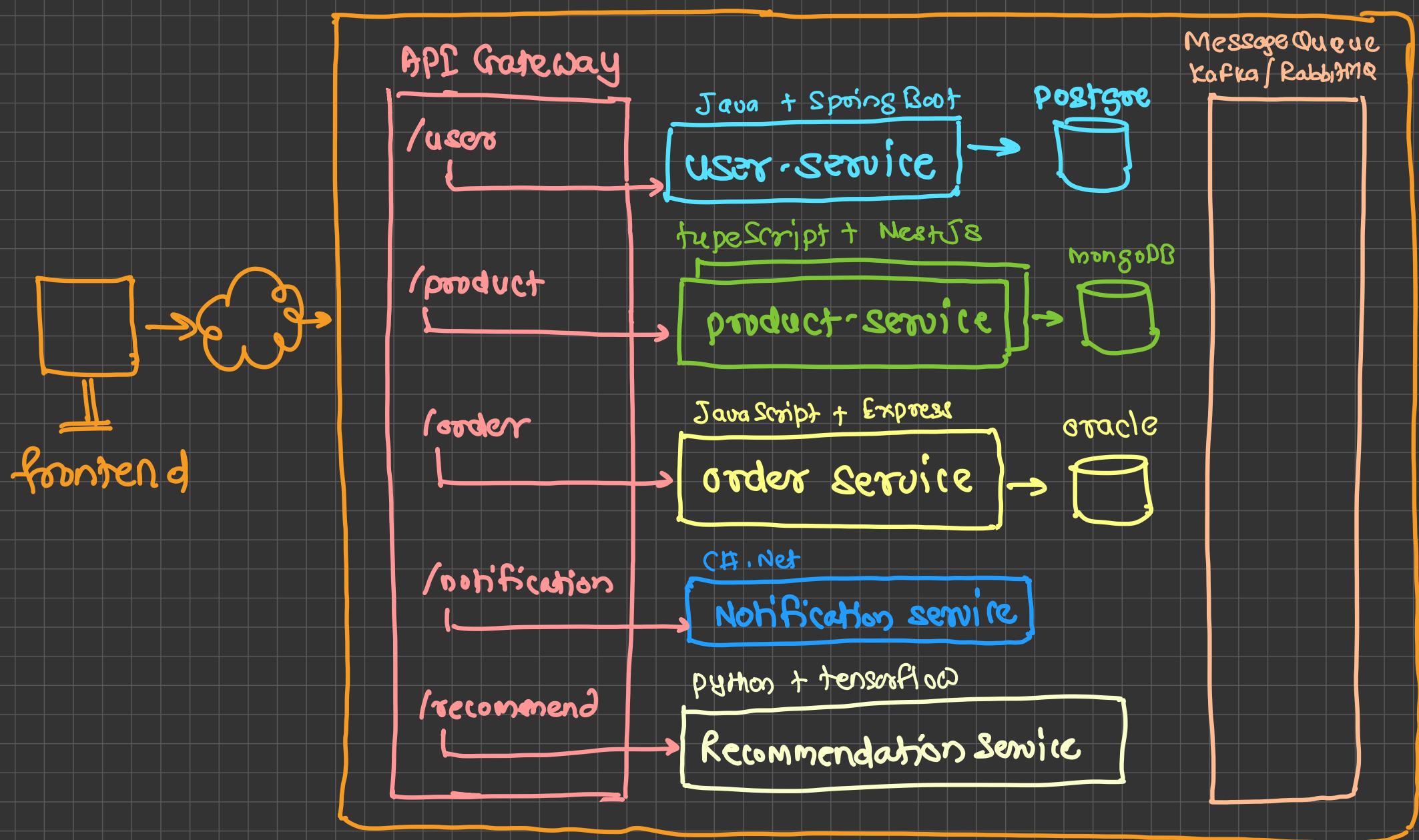
small applications

E-commerce

- user mgmt
 - ↳ Java / Spring Boot + PostgreSQL
- product mgmt
 - ↳ Typescript / NestJS + MongoDB
- order mgmt
 - ↳ JavaScript / Express + Oracle
- notifications
 - ↳ C# / .Net
- recommendation
 - ↳ python / tensorflow / pytorch

microservices

backend application - logical view



What is Microservices Architecture ?

- A microservices architecture, also simply known as microservices, is an architectural method that relies on a series of independently deployable services
- These services have their own business logic and database with a specific goal
- Updating, testing, deployment, and scaling occur within each service
- Microservices decouple major business, domain-specific concerns into separate, independent code bases
- Microservices don't reduce complexity, but they make any complexity visible and more manageable by separating tasks into smaller processes that function independently of each other and contribute to the overall whole
- Adopting microservices often goes hand in hand with DevOps, since they are the basis for continuous delivery practices that allow teams to adapt quickly to user requirements

Pros

- **Agility**
 - Promote agile ways of working with small teams that deploy frequently
- **Flexible scaling**
 - If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure
 - We are now multi-tenant and stateless with customers spread across multiple instances. Now we can support much larger instance sizes
- **Continuous deployment**
 - We now have frequent and faster release cycles. Before we would push out updates once a week and now we can do so about two to three times a day.
- **Highly maintainable and testable**
 - Teams can experiment with new features and roll back if something doesn't work
 - This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services
- **Independently deployable**
 - Since microservices are individual units they allow for fast and easy independent deployment of individual features.
- **Technology flexibility**
 - Microservice architectures allow teams the freedom to select the tools they desire
- **High reliability**
 - You can deploy changes for a specific service, without the threat of bringing down the entire application
- **Happier teams**
 - The Atlassian teams who work with microservices are a lot happier, since they are more autonomous and can build and deploy themselves without waiting weeks for a pull request to be approved

Cons

- **Development sprawl**
 - These add more complexity compared to a monolith architecture, as there are more services in more places created by multiple teams
 - If development sprawl isn't properly managed, it results in slower development speed and poor operational performance
- **Exponential infrastructure costs**
 - Each new microservice can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more
- **Added organizational overhead**
 - Teams need to add another level of communication and collaboration to coordinate updates and interfaces
- **Debugging challenges**
 - Each microservice has its own set of logs, which makes debugging more complicated
 - Plus, a single business process can run across multiple machines, further complicating debugging
- **Lack of standardization**
 - Without a common platform, there can be a proliferation of languages, logging standards, and monitoring
- **Lack of clear ownership**
 - As more services are introduced, so are the number of teams running those services
 - Over time it becomes difficult to know the available services a team can leverage and who to contact for support

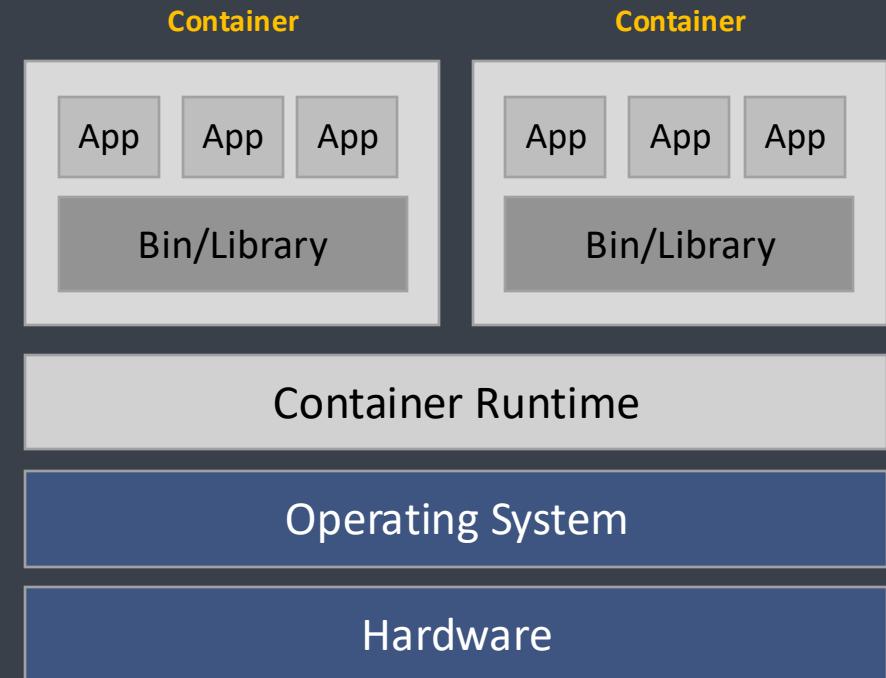
Containerization

What is Containerization

- Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a **container**—that runs consistently on any infrastructure
- More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications
- Containerization allows developers to create and deploy applications faster and more securely
- With traditional methods, code is developed in a specific computing environment which, when transferred to a new location, often results in bugs and errors
- For example, when a developer transfers code from a desktop computer to a VM or from a Linux to a Windows operating system
- Containerization eliminates this problem by bundling the application code together with the related configuration files, libraries, and dependencies required for it to run
- This single package of software or “container” is abstracted away from the host operating system, and hence, it stands alone and becomes portable—able to run across any platform or cloud, free of issues

Container deployment

- Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications
- Therefore, containers are considered lightweight
- Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more
- As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions



Containerization vs Virtualization

Virtual Machine	Container
Hardware level virtualization	OS virtualization
Heavyweight (bigger in size)	Lightweight (smaller in size)
Slow provisioning	Real-time and fast provisioning
Limited Performance	Native performance
Fully isolated	Process-level isolation
More secure	Less secure
Each VM has separate OS	Each container can share OS resources
Boots in minutes	Boots in seconds
Pre-configured VMs are difficult to find and manage	Pre-built containers are readily available
Can be easily moved to new OS	Containers are destroyed and recreated
Creating VM takes longer time	Containers can be created in seconds

Benefits

- **Portability**

- A container creates an executable package of software that is abstracted away from (not tied to or dependent upon) the host operating system, and hence, is portable and able to run uniformly and consistently across any platform or cloud

- **Agility**

- The open source Docker Engine for running containers started the industry standard for containers with simple developer tools and a universal packaging approach that works on both Linux and Windows operating systems
- The container ecosystem has shifted to engines managed by the Open Container Initiative (OCI)
- Software developers can continue using agile or DevOps tools and processes for rapid application development and enhancement

- **Speed**

- Containers are often referred to as “lightweight,” meaning they share the machine’s operating system (OS) kernel and are not bogged down with this extra overhead
- Not only does this drive higher server efficiencies, it also reduces server and licensing costs while speeding up start-times as there is no operating system to boot

- **Fault isolation**

- Each containerized application is isolated and operates independently of others
- The failure of one container does not affect the continued operation of any other containers
- Development teams can identify and correct any technical issues within one container without any downtime in other containers
- Also, the container engine can leverage any OS security isolation techniques—such as SELinux access control—to isolate faults within containers

Benefits

- **Efficiency**

- Software running in containerized environments shares the machine's OS kernel, and application layers within a container can be shared across containers
- Thus, containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM. This drives higher server efficiencies, reducing server and licensing costs

- **Ease of management**

- Container orchestration platform automates the installation, scaling, and management of containerized workloads and services
- Container orchestration platforms can ease management tasks such as scaling containerized apps, rolling out new versions of apps, and providing monitoring, logging and debugging, among other functions. Kubernetes, perhaps the most popular container orchestration system available, is an open source technology (originally open-sourced by Google, based on their internal project called Borg) that automates Linux container functions originally
- Kubernetes works with many container engines, such as Docker, but it also works with any container system that conforms to the Open Container Initiative (OCI) standards for container image formats and runtimes

- **Security**

- The isolation of applications as containers inherently prevents the invasion of malicious code from affecting other containers or the host system
- Additionally, security permissions can be defined to automatically block unwanted components from entering containers or limit communications with unnecessary resources

Docker

What is Docker ?

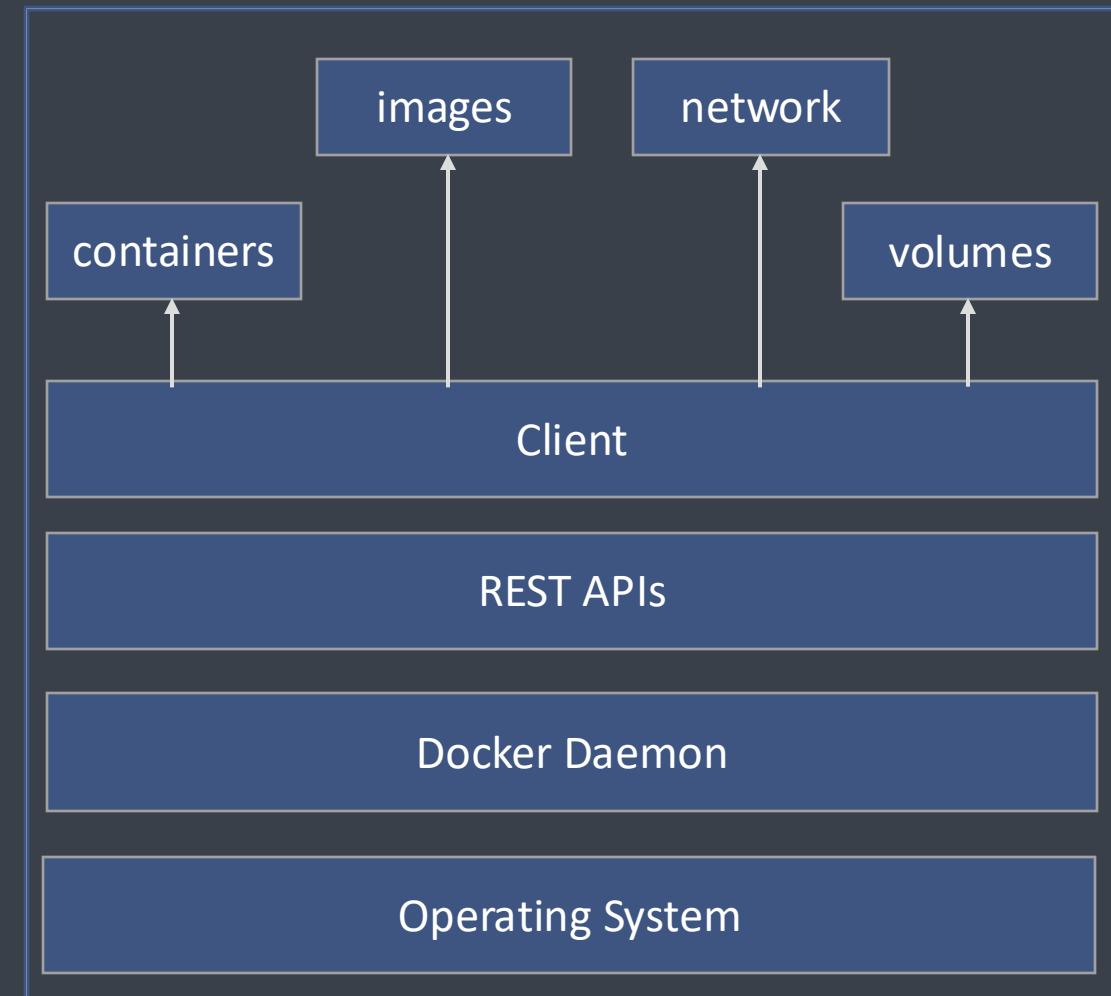
- Docker is an open source platform that enables developers to build, deploy, run, update and manage containers—standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment
- Why docker?
 - It is an easy way to create application deployable packages
 - Developer can create ready-to-run containerized applications
 - It provides consistent computing environment
 - It works equally well in on-prem as well as cloud environments
 - It is light weight compared to VM

Little history about Docker

- Docker Inc, started by Solomon Hykes, is behind the docker tool
- Docker Inc started as Paas provider called as dotCloud
- In 2013, the dotCloud became Docker Inc
- Docker Inc was using LinuX Containers (LXC) before version 0.9
- After 0.9 (2014), Docker replaced LXC with its own library libcontainer which is developed in Go programming language
- Its not the only solution for containerization
 - “FreeBSD Jails”, launched in 2000
 - LXD is next generation system container manager build on top of LXC and REST APIs
 - Google has its own open source container technology lmctfy (Let Me Contain That For You)
 - Rkt is another option for running containers

Docker Architecture

- **Docker daemon (dockerd)**
 - Continuous running process
 - Manages the containers
- **REST APIs**
 - Used to communicate with docker daemon
- **Client (docker)**
 - Provides command line interface
 - Used to perform all the tasks



libcontainer

- Docker has replaced LXC by libcontainer, which is used to manage the containers
- Libcontainer uses
 - Namespaces
 - Creates isolated workspace which limits what container can see
 - Provides a layer of isolation to the container
 - Each container runs in a separate namespace
 - Processes running in a namespace can interact with other processes or use resources which are the part of the same namespace
 - E.g. process ID, network, IPC, Filesystem
 - Control Groups (cgroups)
 - Used to share the available resources to the containers
 - It optionally enforces limits and constraints on resource usage
 - It limits how much a container can use
 - E.g. CPU, Disk space, memory

libcontainer

- Union File System (UnionFS)

- It uses layers
- It is a lightweight and very fast FS
- Docker uses different variants of UnionFS
 - Aufs (advanced multi-layered unification filesystem)
 - Btrfs (B-Tree FS)
 - VFS (Virtual FS)
 - Devicemapper

Docker Objects

- **Images**: read only template with instructions for creating docker containers
- **Container**: running instance of a docker image
- **Network**: network interface used to connect the containers to each other or external networks
- **Volumes**: used to persist the data generated by and used by the containers
- **Registry**: private or public collection of docker images
- **Service**: used to deploy application in a docker multi node cluster

What is Docker image ?

- A Docker image is a file used to execute code in a Docker container
- Docker images act as a set of instructions to build a Docker container, like a template
- Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual machine (VM) environments
- A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container
- Docker images have multiple layers, each one originates from the previous layer but is different from it
- The layers speed up Docker builds while increasing reusability and decreasing disk use
- Image layers are also read-only files
- Once a container is created, a writable layer is added on top of the unchangeable images, allowing a user to make changes

Docker Container

Docker Container

- It is a running aspect of docker image
- Contains one or more running processes
- It is a self-contained environment
- It wraps up an application into its own isolated box (application running inside a container has no knowledge of any other applications or processes that exist outside the container)
- A container can not modify the image from which it is created
- It consists of
 - Your application code
 - Dependencies
 - Networking
 - Volumes
- Containers are stored under /var/lib/docker
- This directory contains images, containers, network volumes etc

Basic Operations

- Creating container
- Starting container
- Running container
- Listing running containers
- Listing all containers
- Getting information of a container
- Stopping container
- Deleting container

Attaching a container

- There are two ways to attach to a container
- Attach
 - Used to attach the container
 - Uses only one input and output stream
 - Task
 - Attach to a running container
- Exec
 - Mainly it is used for running a command inside a container
 - Task
 - Execute a command inside container

Hostname and name of container

- To check the host name
 - Go inside the container
 - Check the hostname by using a command `hostname`
- Docker uses the first 12 characters of container id as hostname
- Docker automatically generates a name at random for each container

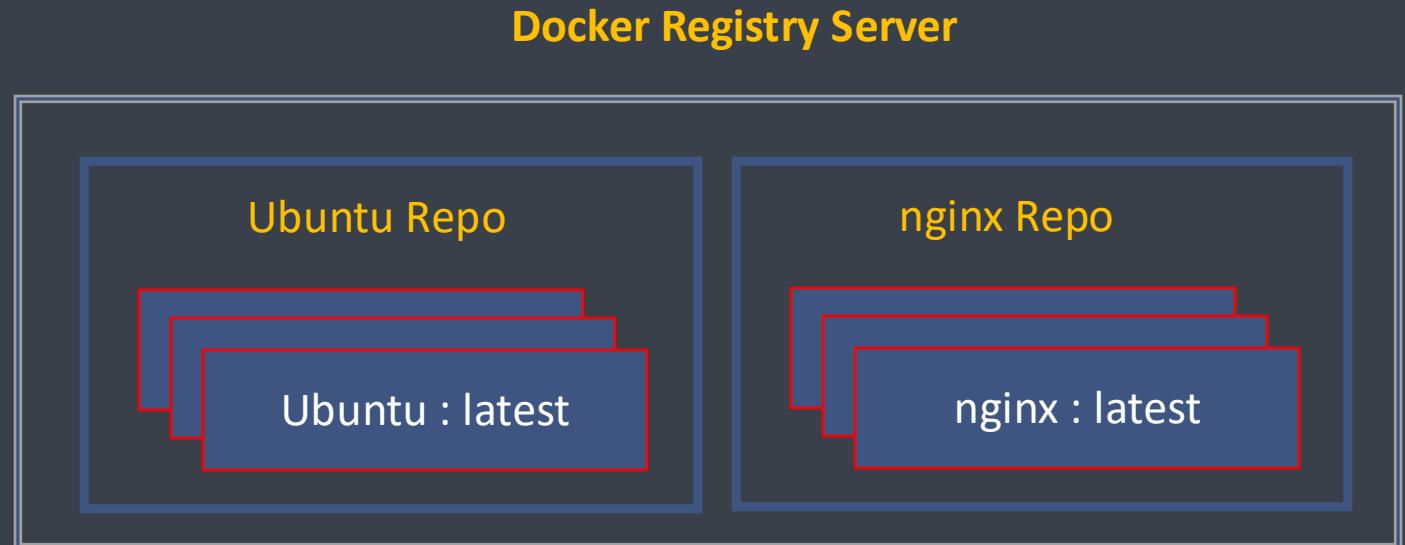
Publishing port on container

- Publishing a port is required to give an external access to your application
- Port can be published only at the time of creating a container
- You can not update the port configuration on running container
- Task
 - Run a httpd container with port 8080 published, to access apache externally

Docker Images (Advanced)

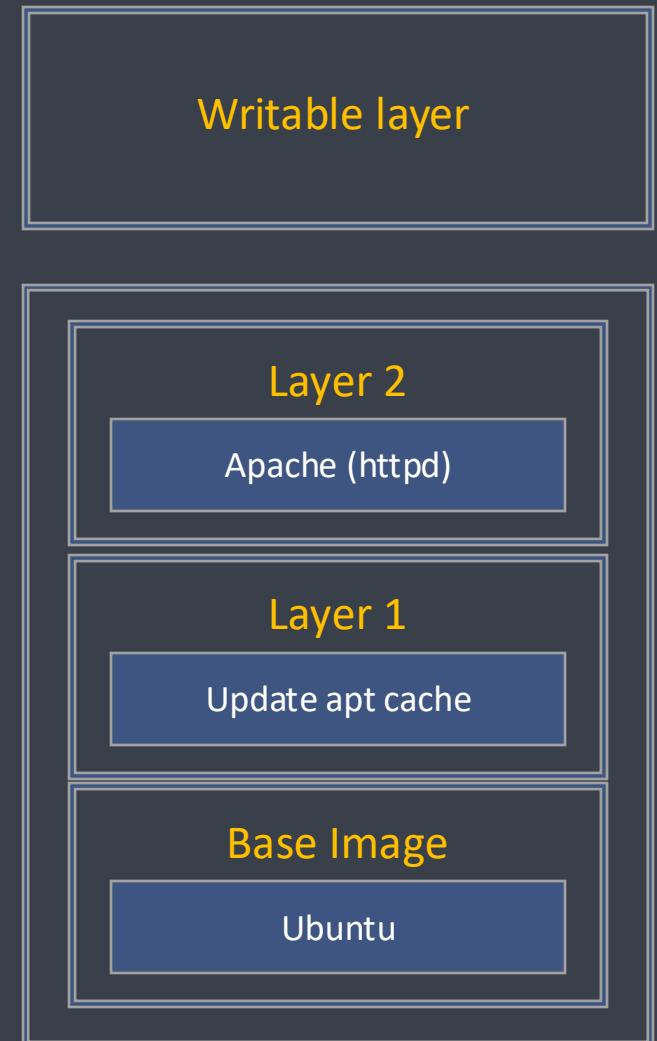
Docker Image

- Read-only instructions to run the containers
- It is made up of different layers
- Repositories hold images
- Docker registry stores repositories
- To create a custom image
 - Commit the running container
 - Use a Dockerfile
- Task
 - Create a container
 - Create a directory and a file within it
 - Commit the container to create a new image



Layered File System

- Docker images are made of layered FS
- Docker uses UnionFS for implementing the layered docker images
- Any update on the image adds a new layer
- All changes made to the running container are written inside a writable layer



Dockerfile

- The Dockerfile contains a series of instructions paired with arguments
- Each instruction should be in upper-case and be followed by an argument
- Instructions are processed from top to bottom
- Each instruction adds a new layer to the image and then commits the image
- Upon running, changes made by an instruction make it to the container

Dockerfile instructions

- FROM
- ENV
- RUN
- CMD
- EXPOSE
- WORKDIR
- ADD
- COPY
- LABEL
- MAINTAINER
- ENTRYPOINT

Orchestration

Container Orchestration

- Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments
- Software teams use container orchestration to control and automate many tasks
 - Provisioning and deployment of containers
 - Redundancy and availability of containers
 - Scaling up or removing containers to spread application load evenly across host infrastructure
 - Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
 - Allocation of resources between containers
 - External exposure of services running in a container with the outside world
 - Load balancing of service discovery between containers
 - Health monitoring of containers and hosts
 - Configuration of an application in relation to the containers running it
- Orchestration Tools
 - Docker Swarm
 - Kubernetes
 - Mesos
 - Marathon

Docker Swarm

- Docker Swarm is a container orchestration engine
- It takes multiple Docker Engines running on different hosts and lets you use them together
- The usage is simple: declare your applications as stacks of services, and let Docker handle the rest
- It is secure by default
- It is built using Swarmkit

What is a swarm?

- A swarm consists of multiple Docker hosts which run in **swarm mode**
- A given Docker host can be a manager, a worker, or perform both roles
- When you create a service, you define its optimal state
- Docker works to maintain that desired state
 - For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes
- A *task* is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container
- When Docker is running in swarm mode, you can still run standalone containers on any of the Docker hosts participating in the swarm, as well as swarm services
- A key difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any daemon

Features

- Cluster management integrated with Docker Engine
- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing
- Secure by default
- Rolling updates

Nodes

- A **node** is an instance of the Docker engine participating in the swarm
- You can run one or more nodes on a single physical computer or cloud server
- To deploy your application to a swarm, you submit a service definition to a **manager node**
- **Manager Node**
 - The manager node dispatches units of work called **tasks** to worker nodes
 - Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm
 - Manager nodes elect a single leader to conduct orchestration tasks
- **Worker nodes**
 - Worker nodes receive and execute tasks dispatched from manager nodes
 - An agent runs on each worker node and reports on the tasks assigned to it
 - The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker

Services and tasks

- **Service**

- A service is the definition of the tasks to execute on the manager or worker nodes
- It is the central structure of the swarm system and the primary root of user interaction with the swarm
- When you create a service, you specify which container image to use and which commands to execute inside running containers

- **Task**

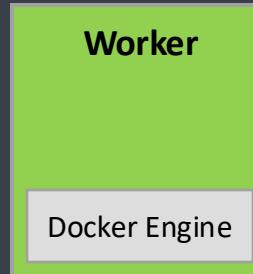
- A task carries a Docker container and the commands to run inside the container
- It is the atomic scheduling unit of swarm
- Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale
- Once a task is assigned to a node, it cannot move to another node
- It can only run on the assigned node or fail

Create a Swarm and add workers to swarm

- In our swarm we are going to use 3 nodes (one manager and two workers)
- Every node must have Docker Engine 1.12 or newer installed
- Following ports are used in node communication
 - TCP port 2377 is used for cluster management communication
 - TCP and UDP port 7946 is used for node communication
 - UDP port 4789 is used for overlay network traffic



```
docker swarm init --advertise-addr <ip>
```



```
docker swarm join --token <token>
```

Swarm Setup

- **Create swarm**

```
> docker swarm init --advertise-addr <MANAGER-IP>
```

- **Get current status of swarm**

```
> docker info
```

- **Get the list of nodes**

```
> docker node ls
```

Swarm Setup

- **Get token (on manager node)**

> `docker swarm join-token worker`

- **Add node (on worker node)**

> `docker swarm join --token <token>`

Overlay Network

- It is a computer network built on top of another network
- Sits on top of the host-specific networks and allows container, connected to it, to communicate securely
- When you initialize a swarm or join a host to swarm, two networks are created
 - An overlay network called as ingress network
 - A bridge network called as docker_gwbridge
- Ingress network facilitates load balancing among services nodes
- Docker_gwbridge is a bridge network that connect overlay networks to individual docker daemon's physical network

Service

- Definition of tasks to execute on Manager or Worker nodes
- Declarative Model for Services
- Scaling
- Desired state reconciliation
- Service discovery
- Rolling updates
- Load balancing
- Internal DNS component

Swarm Service

- **Deploy a service**

> `docker service create --replicas <no> --name <name> -p <ports> <image> <command>`

- **Get running services**

> `docker service ls`

- **Inspect service**

> `docker service inspect <service>`

- **Get the nodes running service**

> `docker service ps <service>`

Swarm Service

- **Scale service**

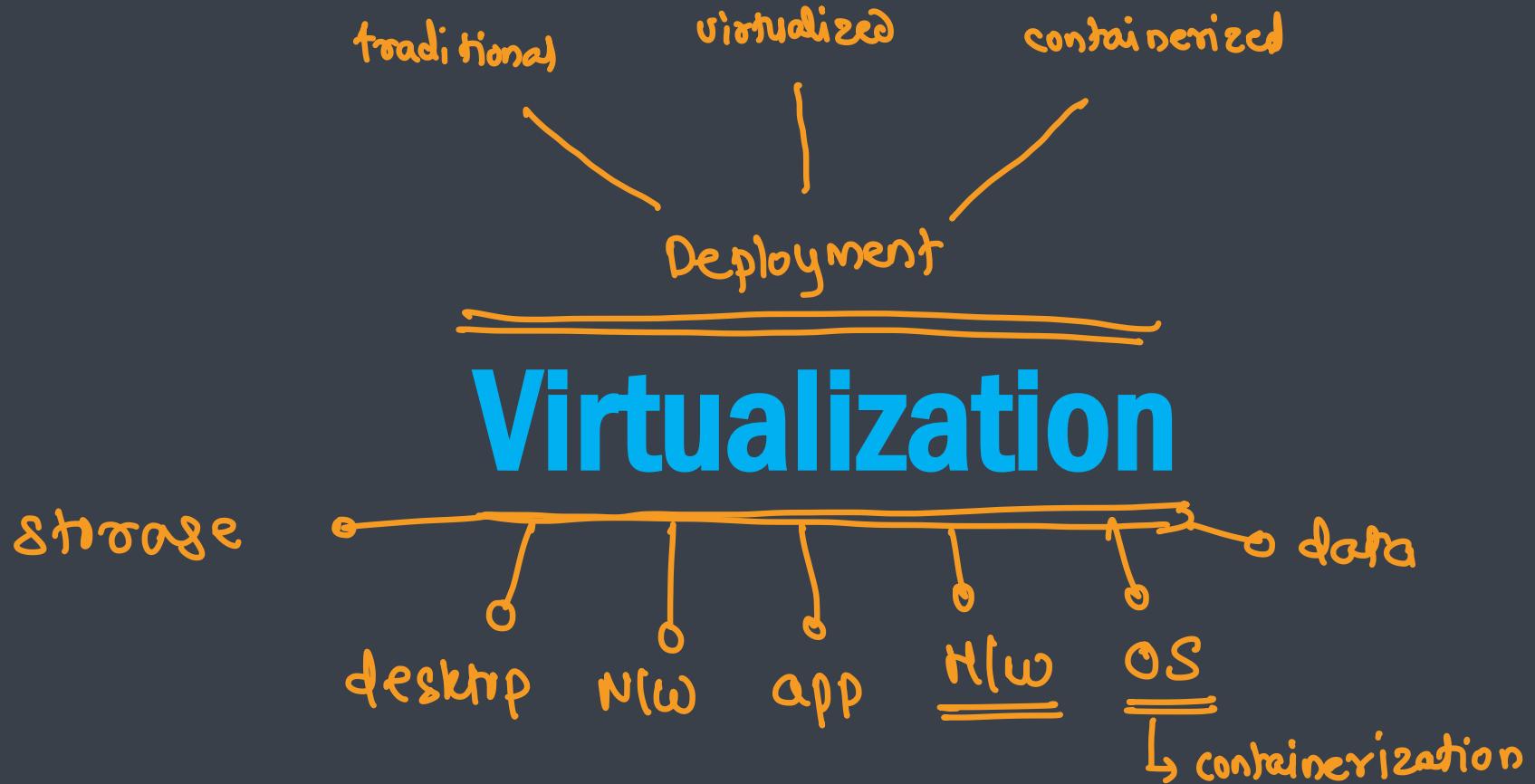
> `docker service scale <service>=<scale>`

- **Update service**

> `docker service update --image <image> <service>`

- **Delete service**

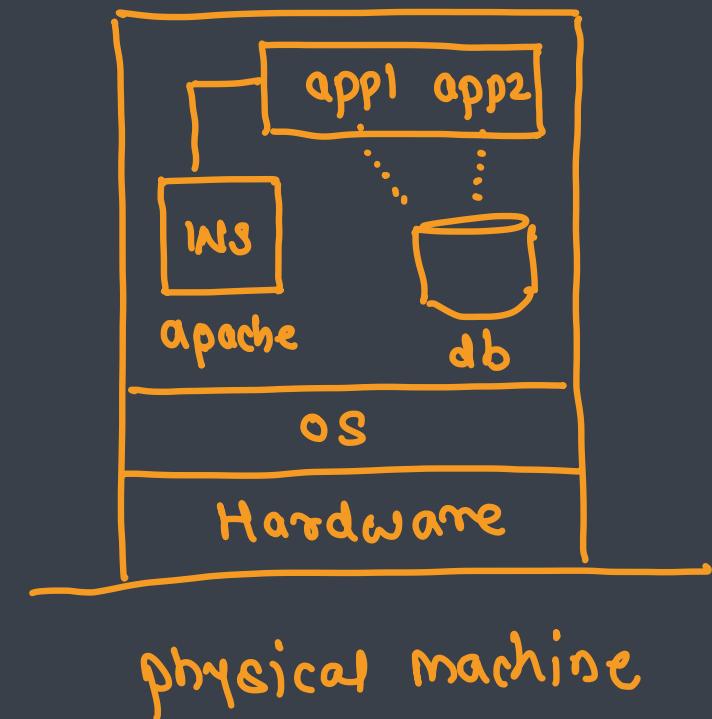
> `docker service rm <service>`

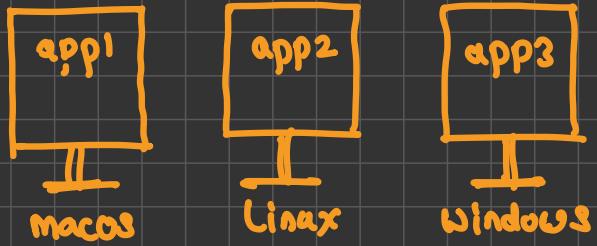


Traditional Deployment → deprecated

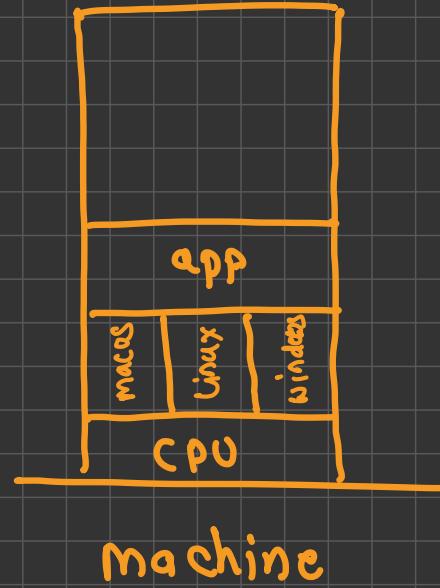
- Early on, organizations ran applications on physical servers
- There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues
- For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform
- A solution for this would be to run each application on a different physical server
- But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers

resource → CPU + memory





+ fastest option
- costliest



multi-booting machine

+ cheapest
- time [one os can run at a time]

What is virtualization

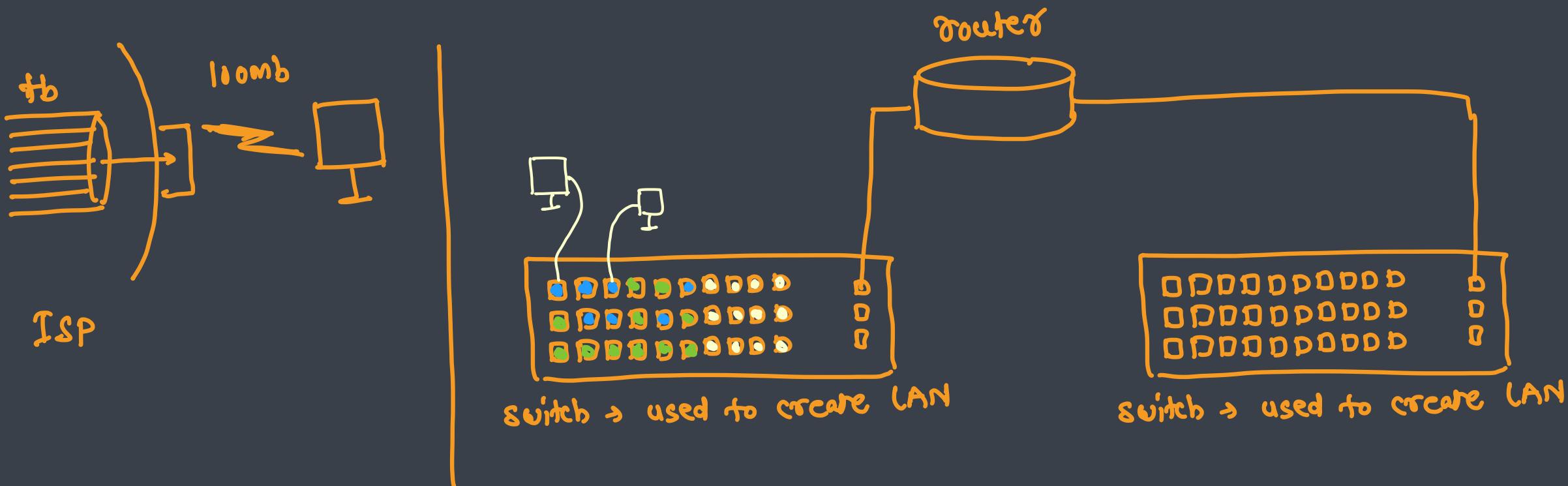
→ No physical existence

+ faster
+ run multiple OSes simultaneously
+ can be easily shared

- Virtualization is the creation of a virtual -- rather than actual -- version of something, such as an operating system (OS), a server, a storage device or network resources
- Virtualization uses software that simulates hardware functionality in order to create a virtual system
- This practice allows IT organizations to operate multiple operating systems, more than one virtual system and various applications on a single server
- Types
 - ① ▪ Network virtualization
 - ② ▪ Storage virtualization
 - ③ ▪ Data virtualization
 - ④ ▪ Desktop virtualization
 - ⑤ ▪ Application virtualization
 - ⑥ ▪ Hardware virtualization
- ⑦ OS virtualization
(containerization) * * *

Network Virtualization

- Network virtualization takes the available resources on a network and breaks the bandwidth into discrete channels
- Admins can secure each channel separately, and they can assign and reassign channels to specific devices in real time
- The promise of network virtualization is to improve networks' speed, availability and security, and it's particularly useful for networks that must support unpredictable usage bursts

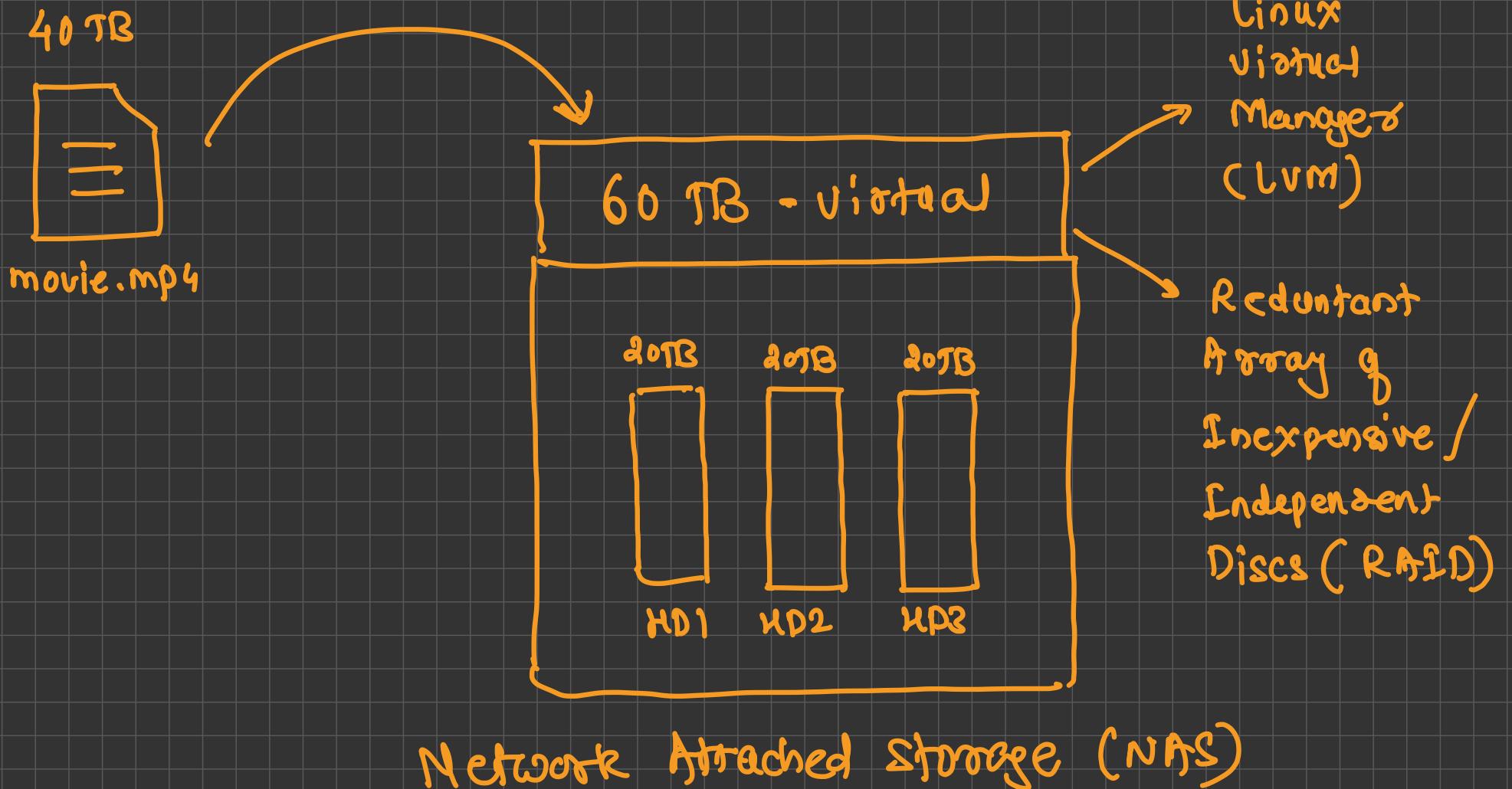


Storage Virtualization

- Storage virtualization is the pooling of physical storage from multiple network storage devices into what appears to be a single storage device that is managed from a central console
- Storage virtualization is commonly used in storage area networks
- Applications can use storage without having any concern for where it resides, what technical interface it provides, how it has been implemented, which platform it uses and how much of it is available
- Benefits
 - Makes the remote storage devices appear local
 - Multiple smaller volumes appear as a single large volume
 - Data is spread over multiple physical disks to improve reliability and performance
 - All operating systems use the same storage device
 - Provided high availability, disaster recovery, improved performance and sharing

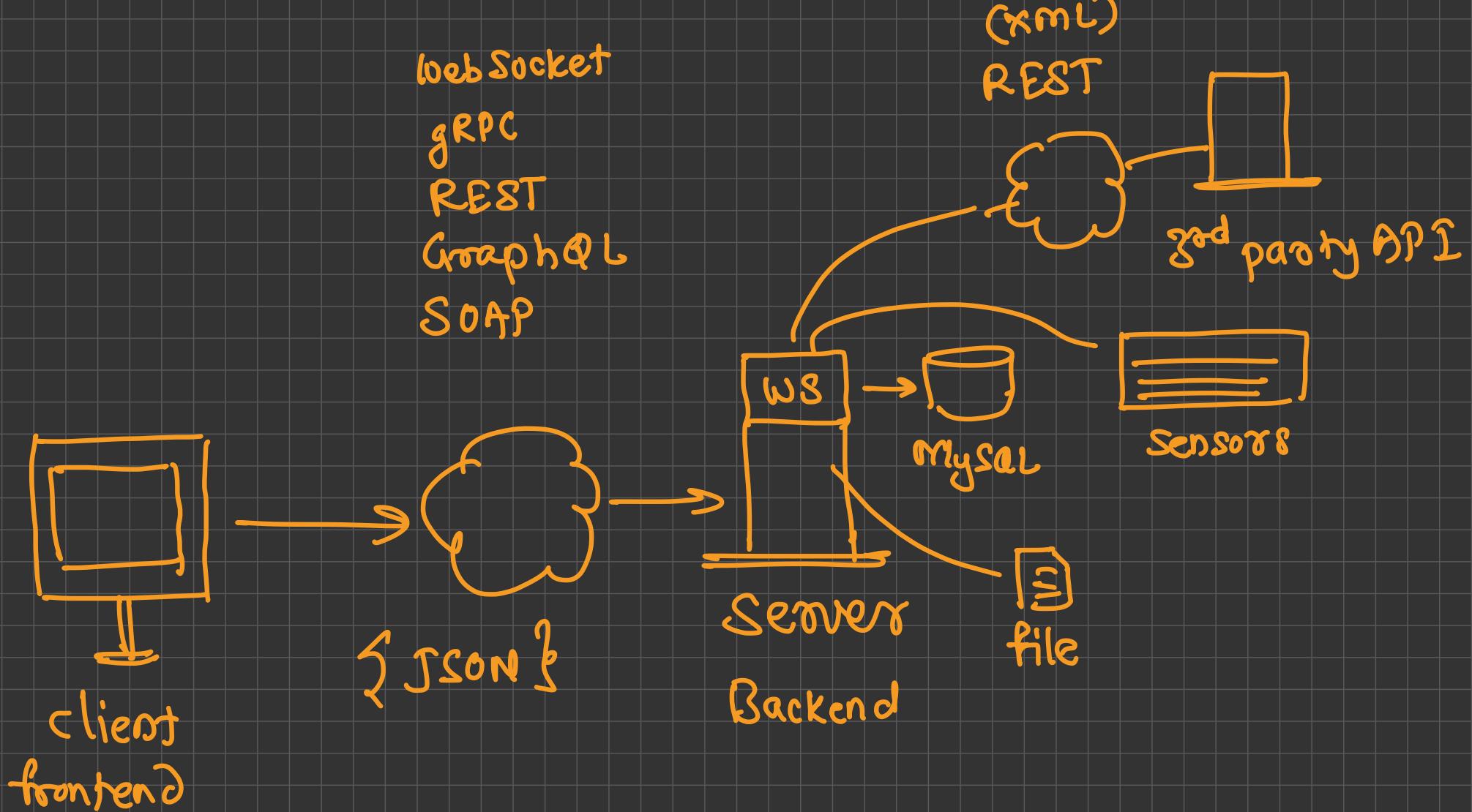


Storage virtualization



Data virtualization

- Data virtualization is the process of aggregating data from different sources of information to develop a single, logical and virtual view of information so that it can be accessed by front-end solutions such as applications, dashboards and portals without having to know the data's exact storage location
- The process of data virtualization involves abstracting, transforming, federating and delivering data from disparate sources
- The main goal of data virtualization technology is to provide a single point of access to the data by aggregating it from a wide range of data sources
- Benefits
 - Abstraction of technical aspects of stored data like APIs, Language, Location, Storage structure
 - Provides an ability to connect multiple data sources from a single location
 - Provides an ability to combine the data result sets across multiple sources (also known as data federation)
 - Provides an ability to deliver the data as requested by users

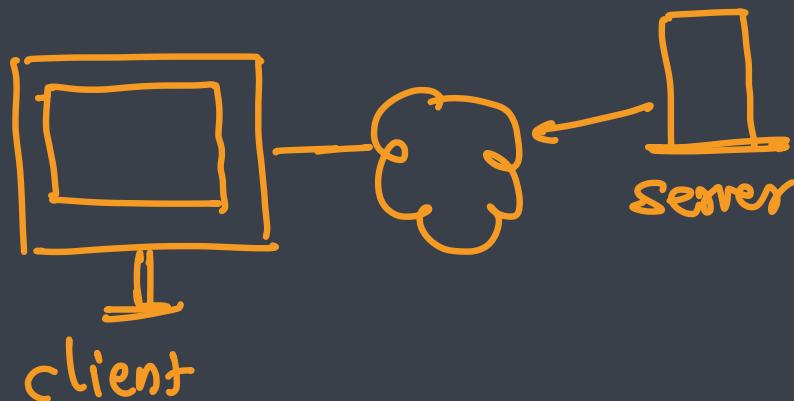


Desktop virtualization → Anydesk

- With desktop virtualization, the goal is to isolate a desktop OS from the endpoint that employees use to access it
- It provides an ability to connect to the desktop from remote site
- When multiple users connect to a shared desktop, as is the case with Microsoft Remote Desktop Services, it's known as shared hosted desktop virtualization

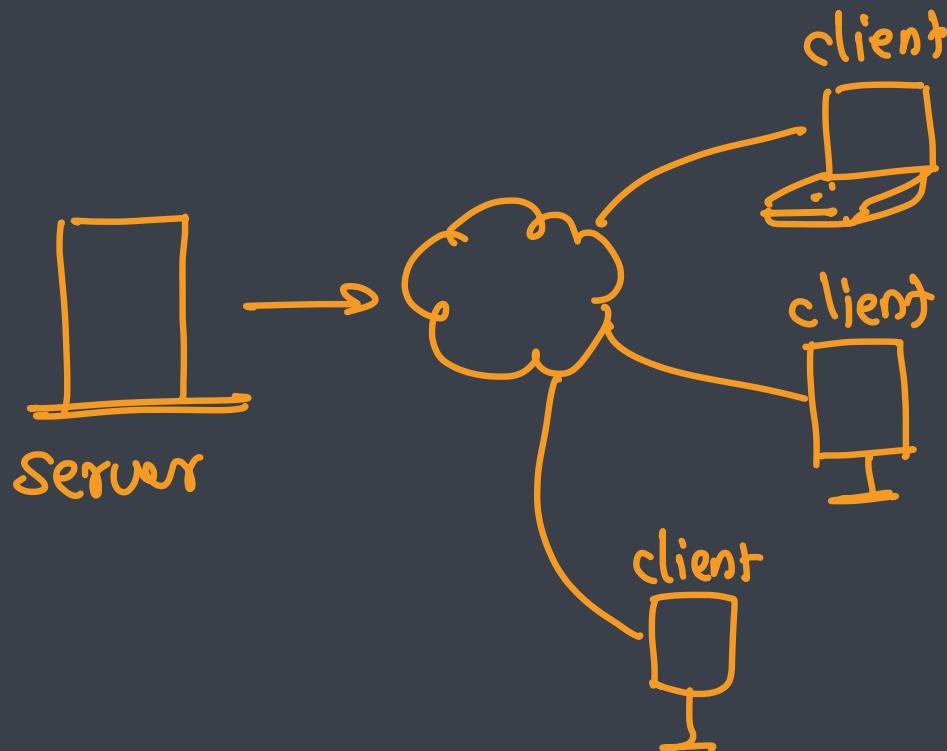
UNC → virtual Network Connection

RDC → Remote Desktop Connection



Application Virtualization

- With application virtualization, an app runs separately from the device that accesses it
- Application virtualization makes it possible for IT admins to install, patch and update only one version of an app rather than performing the same management tasks multiple times

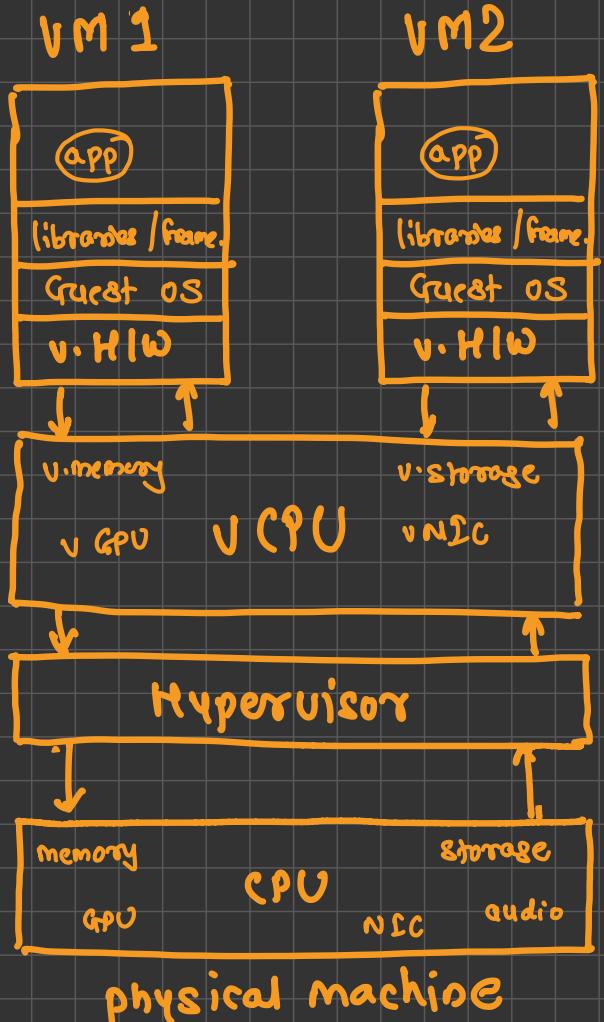


Hardware Virtualization

- Hardware virtualization or platform virtualization refers to the creation of a virtual machine that acts like a real computer with an operating system
- The process of masking the hardware resources like
 - CPU
 - Storage
 - Memory
- For example, a computer that is running Microsoft Windows may host a virtual machine that looks like a computer with the Ubuntu Linux operating system; Ubuntu-based software can be run on the virtual machine
- The process of creating Machines

Type I → Bare Metal Hypervisor

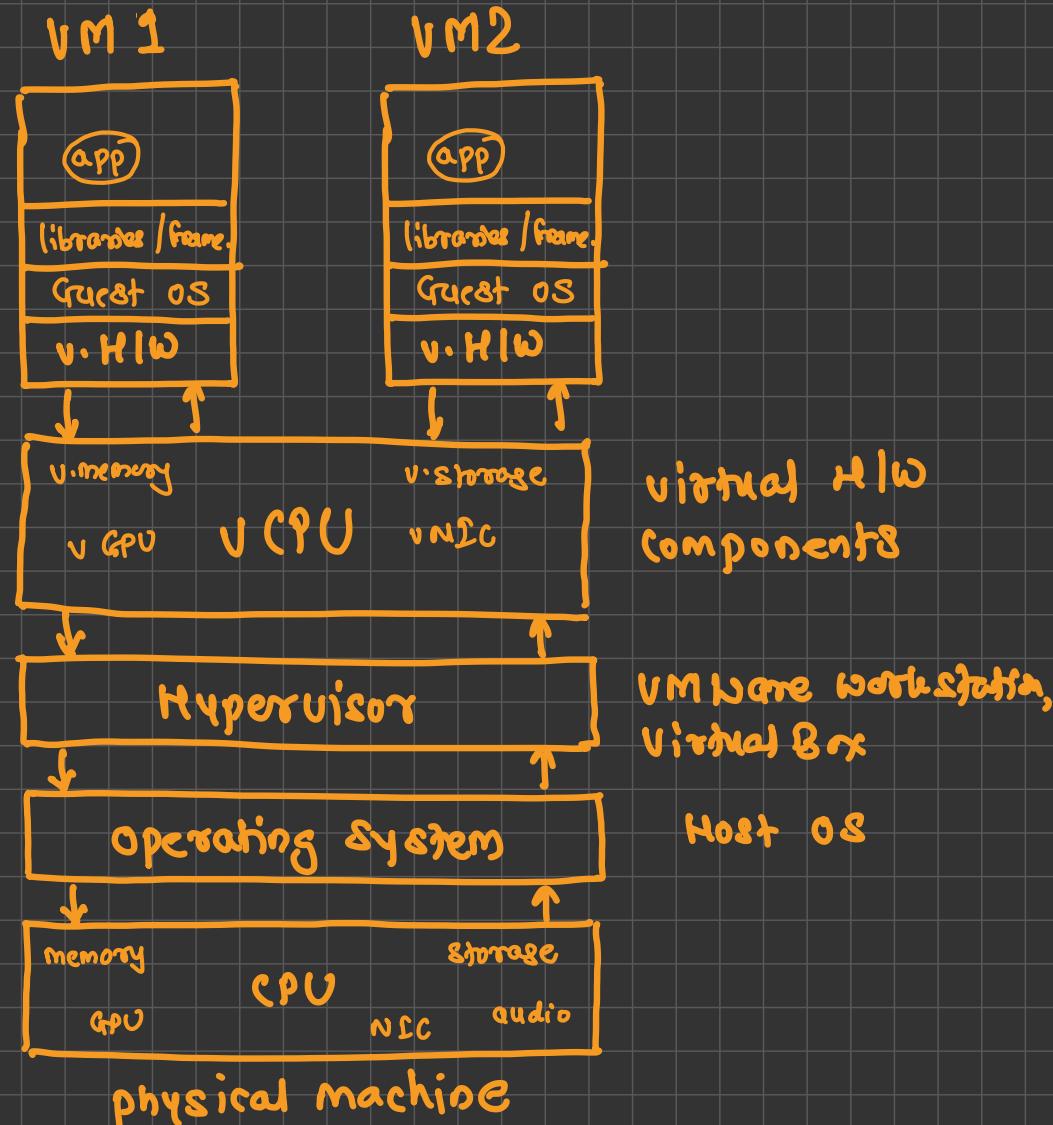
- Does Not require Host OS
- faster and utilizes resources in better ways compared with Type II
- used by cloud providers



VMware ESXi,
Xen, KVM

Type II → Hosted Hypervisor

- requires host OS
- requires more resources
- used by developers / testers



Host OS

virtual HW
components

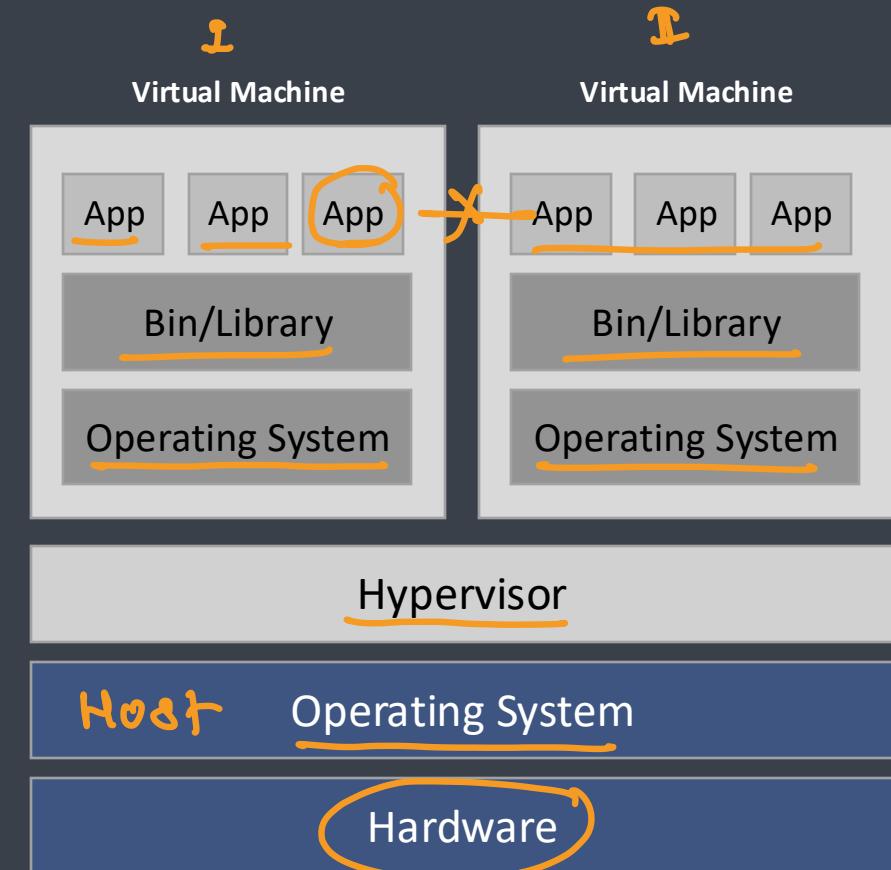
VMWare workstation,
Virtual Box

Virtual Machine

- A virtual machine is the emulated equivalent of a computer system that runs on top of another system
- Virtual machines may have access to any number of resources
 - Computing power - through hardware-assisted but limited access to the host machine's CPU
 - Memory - one or more physical or virtual disk devices for storage
 - A virtual or real network interfaces
 - Any devices such as
 - video cards,
 - USB devices,
 - other hardware that are shared with the virtual machine
- If the virtual machine is stored on a virtual disk, this is often referred to as a **disk image**

Virtualized Deployment

- It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU
- Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application
- Virtualization allows better utilization of resources in a physical server and allows better scalability because
 - an application can be added or updated easily
 - reduces hardware costs
- With virtualization you can present a set of physical resources as a cluster of disposable virtual machines
- Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware



Types of hardware virtualization

- **Type I**

- A Type 1 hypervisor runs directly on the host machine's physical hardware, and it's referred to as a bare-metal hypervisor
- It doesn't have to load an underlying OS first
- With direct access to the underlying hardware and no other software, it is more efficient and provides better performance
- It is best suited for enterprise computing or data centers
- E.g. VMware ESXi, Microsoft Hyper-V server and open source KVM

- **Type II**

- A Type 2 hypervisor is typically installed on top of an existing OS, and it's called a hosted hypervisor
- It relies on the host machine's pre-existing OS to manage calls to CPU, memory, storage and network resources
- E.g. VMware Fusion, Oracle VM VirtualBox, Oracle VM Server for x86, Oracle Solaris Zones, Parallels and VMware Workstation

Advantages of virtualization

- **Lower costs**

- Virtualization reduces the amount of hardware servers necessary within a company and data center
- This lowers the overall cost of buying and maintaining large amounts of hardware

- **Easier disaster recovery**

- Disaster recovery is very simple in a virtualized environment
- Regular snapshots provide up-to-date data, allowing virtual machines to be feasibly backed up and recovered
- Even in an emergency, a virtual machine can be migrated to a new location within minutes

- **Easier testing**

- Testing is less complicated in a virtual environment
- Even if a large mistake is made, the test does not need to stop and go back to the beginning
- It can simply return to the previous snapshot and proceed with the test.

- **Quicker backups**

- Backups can be taken of both the virtual server and the virtual machine
- Automatic snapshots are taken throughout the day to guarantee that all data is up-to-date
- Furthermore, the virtual machines can be easily migrated between each other and efficiently redeployed

- **Improved productivity**

- Fewer physical resources results in less time spent managing and maintaining the servers
- Tasks that can take days or weeks in a physical environment can be done in minutes
- This allows staff members to spend the majority of their time on more productive tasks, such as raising revenue and fostering business initiatives

Advantages of virtualization

- **Quicker backups**

- Backups can be taken of both the virtual server and the virtual machine
- Automatic snapshots are taken throughout the day to guarantee that all data is up-to-date
- Furthermore, the virtual machines can be easily migrated between each other and efficiently redeployed

- **Improved productivity**

- Fewer physical resources results in less time spent managing and maintaining the servers
- Tasks that can take days or weeks in a physical environment can be done in minutes
- This allows staff members to spend the majority of their time on more productive tasks, such as raising revenue and fostering business initiatives

frontend and backend

Monolithic Architecture



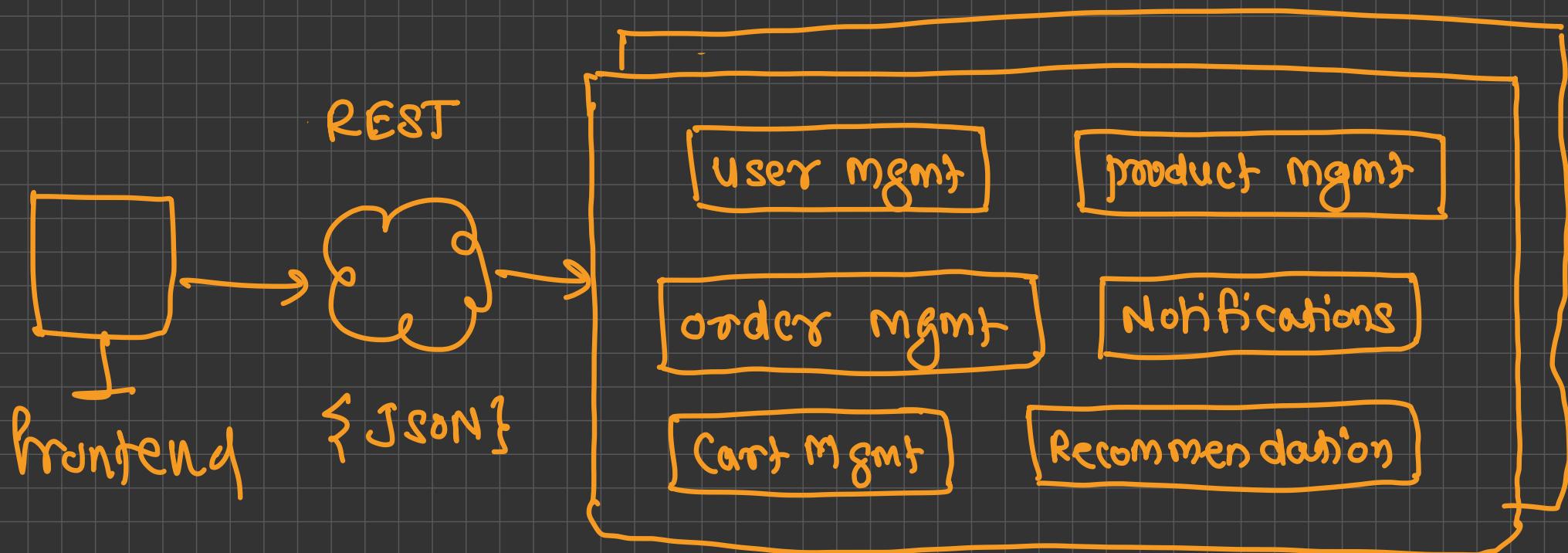
single application with all requirements

E-commerce

- user mgmt
- product mgmt
- order mgmt
- notifications
- recommendation
- cart mgmt

* Technical Stack

- programming language - JavaScript
- frame work - express / fastify / Hapi
- Database - RDBMS → MySQL
- Git Repository - Ecommerce-backend



What is Monolithic Architecture ?

single has all functionality

- A monolithic architecture is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications.
- The word “monolith” is often attributed to something large and glacial, which isn’t far from the truth of a monolith architecture for software design
→ single language
- A monolithic architecture is a singular, large computing network with one code base that couples all of the business concerns together
- To make a change to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface
- Monoliths can be convenient early on in a project's life for ease of code management, cognitive overhead, and deployment
- This allows everything in the monolith to be released at once.

Pros

- **Easy deployment**
 - One executable file or directory makes deployment easier
- **Development**
 - When an application is built with one code base, it is easier to develop
- **Performance**
 - In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices
- **Simplified testing**
 - Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application
- **Easy debugging**
 - With all code located in one place, it's easier to follow a request and find an issue.

Cons

- **Slower development speed**
 - A large, monolithic application makes development more complex and slower
- **Scalability**
 - You can't scale individual components
- **Reliability**
 - If there's an error in any module, it could affect the entire application's availability
- **Barrier to technology adoption**
 - Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming
- **Lack of flexibility**
 - A monolith is constrained by the technologies already used in the monolith
- **Deployment**
 - A small change to a monolithic application requires the redeployment of the entire monolith

Microservices

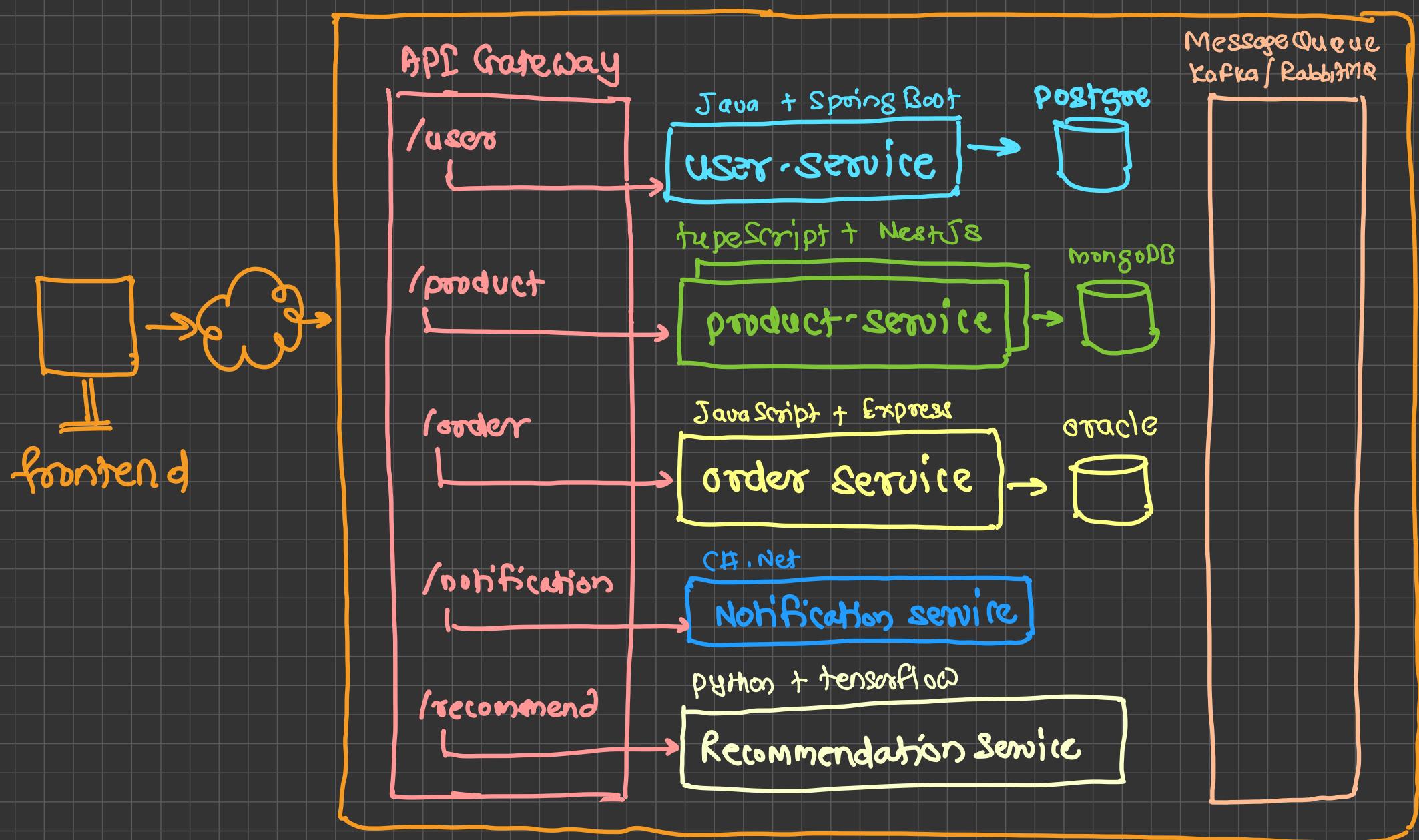
small applications

E-commerce

- user mgmt
 - ↳ Java / Spring Boot + PostgreSQL
- product mgmt
 - ↳ Typescript / NestJS + MongoDB
- order mgmt
 - ↳ JavaScript / Express + Oracle
- notifications
 - ↳ C# / .Net
- recommendation
 - ↳ python / tensorflow / pytorch

microservices

backend application - logical view



What is Microservices Architecture ?

- A microservices architecture, also simply known as microservices, is an architectural method that relies on a series of independently deployable services
- These services have their own business logic and database with a specific goal
- Updating, testing, deployment, and scaling occur within each service
- Microservices decouple major business, domain-specific concerns into separate, independent code bases
- Microservices don't reduce complexity, but they make any complexity visible and more manageable by separating tasks into smaller processes that function independently of each other and contribute to the overall whole
- Adopting microservices often goes hand in hand with DevOps, since they are the basis for continuous delivery practices that allow teams to adapt quickly to user requirements

Pros

- **Agility**
 - Promote agile ways of working with small teams that deploy frequently
- **Flexible scaling**
 - If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure
 - We are now multi-tenant and stateless with customers spread across multiple instances. Now we can support much larger instance sizes
- **Continuous deployment**
 - We now have frequent and faster release cycles. Before we would push out updates once a week and now we can do so about two to three times a day.
- **Highly maintainable and testable**
 - Teams can experiment with new features and roll back if something doesn't work
 - This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services
- **Independently deployable**
 - Since microservices are individual units they allow for fast and easy independent deployment of individual features.
- **Technology flexibility**
 - Microservice architectures allow teams the freedom to select the tools they desire
- **High reliability**
 - You can deploy changes for a specific service, without the threat of bringing down the entire application
- **Happier teams**
 - The Atlassian teams who work with microservices are a lot happier, since they are more autonomous and can build and deploy themselves without waiting weeks for a pull request to be approved

Cons

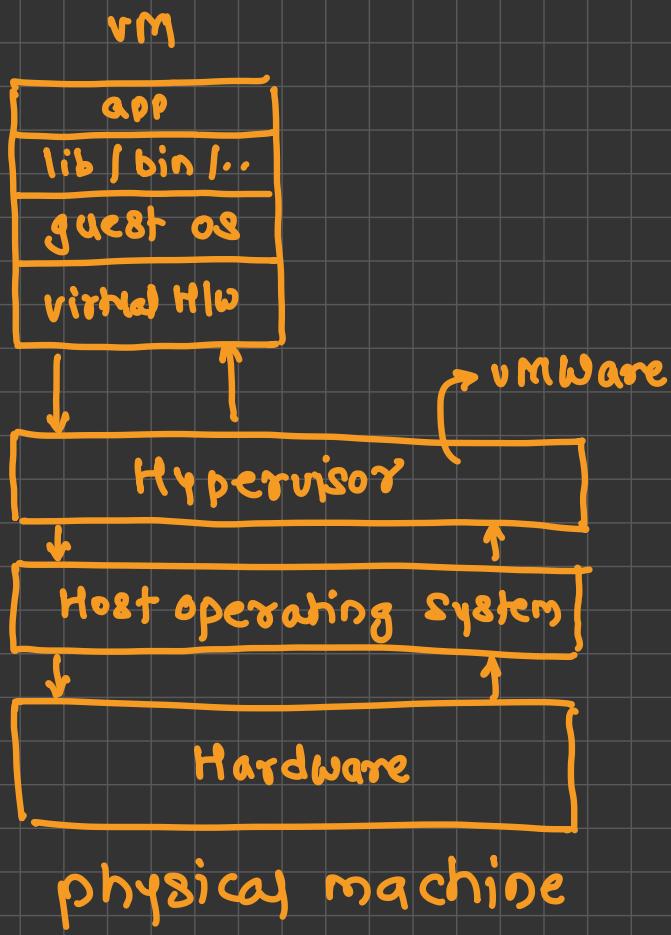
- **Development sprawl**
 - These add more complexity compared to a monolith architecture, as there are more services in more places created by multiple teams
 - If development sprawl isn't properly managed, it results in slower development speed and poor operational performance
- **Exponential infrastructure costs**
 - Each new microservice can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more
- **Added organizational overhead**
 - Teams need to add another level of communication and collaboration to coordinate updates and interfaces
- **Debugging challenges**
 - Each microservice has its own set of logs, which makes debugging more complicated
 - Plus, a single business process can run across multiple machines, further complicating debugging
- **Lack of standardization**
 - Without a common platform, there can be a proliferation of languages, logging standards, and monitoring
- **Lack of clear ownership**
 - As more services are introduced, so are the number of teams running those services
 - Over time it becomes difficult to know the available services a team can leverage and who to contact for support

Containerization

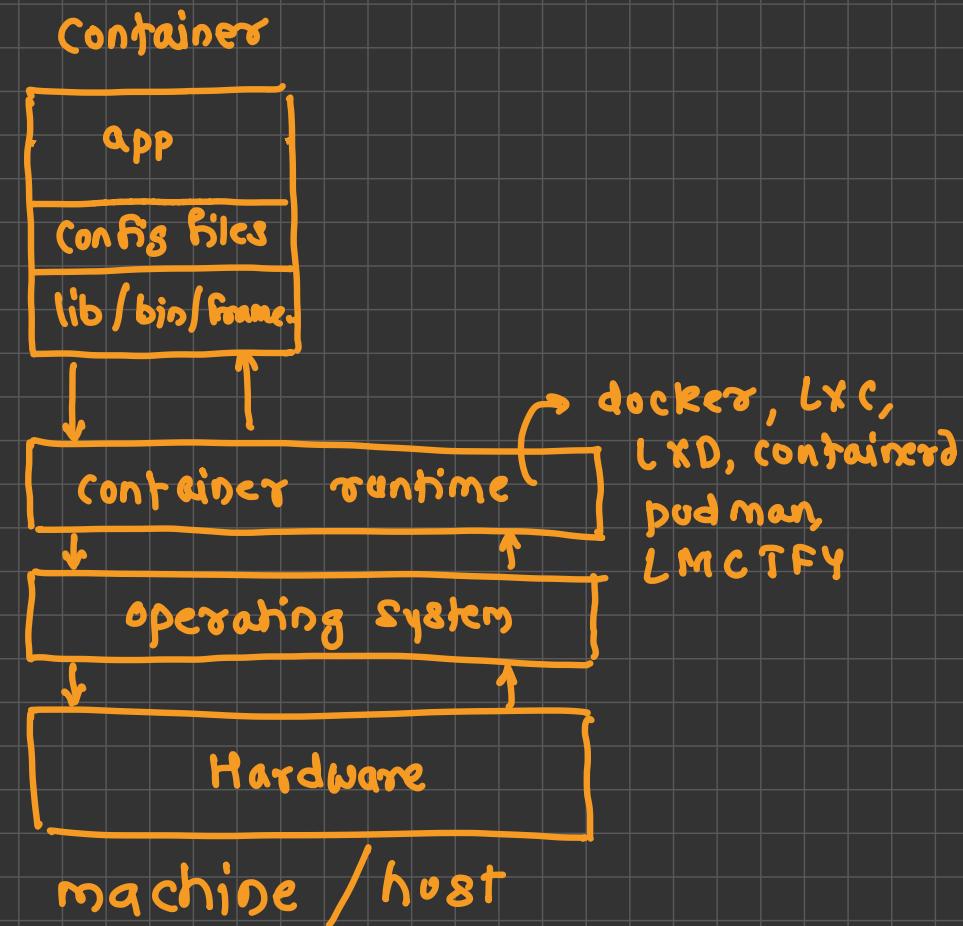
OS virtualization

OS gets shared with containers
lite weight VM

HW virtualization



OS virtualization



What is Containerization

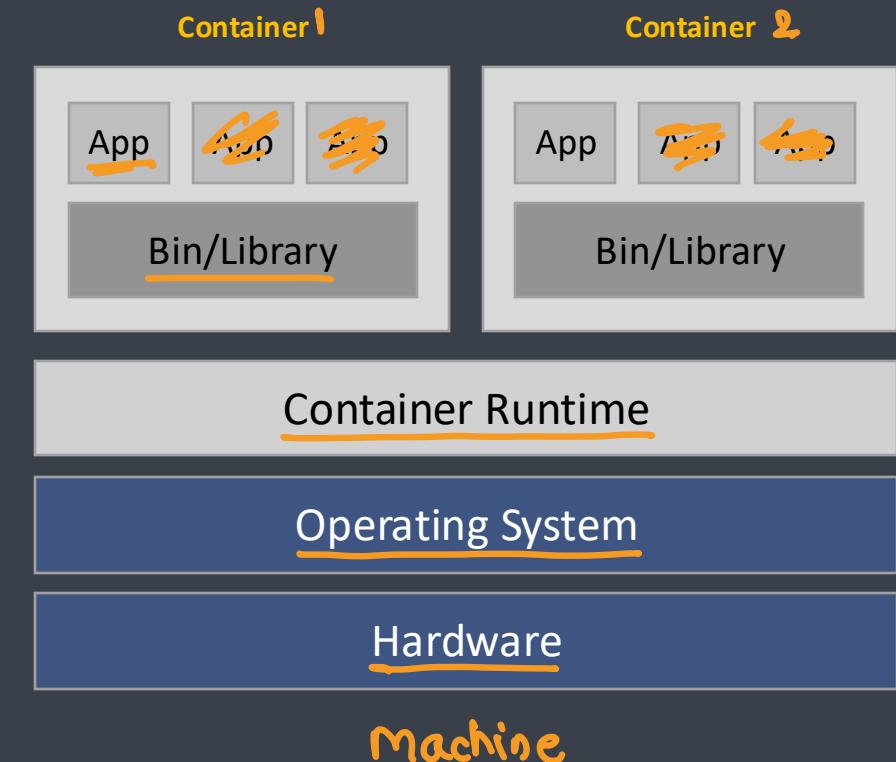
minimilistic OS

libraries
frameworks
config files
Resources

- Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure
- More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications
- Containerization allows developers to create and deploy applications faster and more securely
- With traditional methods, code is developed in a specific computing environment which, when transferred to a new location, often results in bugs and errors
- For example, when a developer transfers code from a desktop computer to a VM or from a Linux to a Windows operating system
- Containerization eliminates this problem by bundling the application code together with the related configuration files, libraries, and dependencies required for it to run
- This single package of software or “container” is abstracted away from the host operating system, and hence, it stands alone and becomes portable—able to run across any platform or cloud, free of issues

Container deployment

- Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications
- Therefore, containers are considered lightweight
- Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more
- As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions



Containerization vs Virtualization (HW)

Virtual Machine	Container
<u>Hardware level virtualization</u> VMs share HW	<u>OS virtualization</u> containers share OS
<u>Heavyweight (bigger in size)</u> because of OS	<u>Lightweight (smaller in size)</u> does not contain OS
<u>Slow provisioning</u> (creation)	<u>Real-time and fast provisioning</u> No booting required, real H/W
<u>Limited Performance</u> slower than container	<u>Native performance</u> faster than VM
<u>Fully isolated</u>	<u>Process-level isolation</u>
<u>More secure</u>	<u>Less secure</u>
<u>Each VM has separate OS</u>	<u>Each container can share OS resources</u>
<u>Boots in minutes</u>	<u>Starts</u> <u>Boots</u> in seconds
<u>Pre-configured VMs are difficult to find and manage</u>	<u>Pre-built containers are readily available</u>
<u>Can be easily moved to new OS</u> VM is mutable	<u>Containers are destroyed and recreated</u>
<u>Creating VM takes longer time</u>	<u>Containers can be created in seconds</u>

Containers are
immutable

Benefits

- Portability

- A container creates an executable package of software that is abstracted away from (not tied to or dependent upon) the host operating system, and hence, is portable and able to run uniformly and consistently across any platform or cloud

- Agility

- The open source Docker Engine for running containers started the industry standard for containers with simple developer tools and a universal packaging approach that works on both Linux and Windows operating systems
- The container ecosystem has shifted to engines managed by the Open Container Initiative (OCI)
- Software developers can continue using agile or DevOps tools and processes for rapid application development and enhancement

- Speed

- Containers are often referred to as “lightweight,” meaning they share the machine’s operating system (OS) kernel and are not bogged down with this extra overhead
- Not only does this drive higher server efficiencies, it also reduces server and licensing costs while speeding up start-times as there is no operating system to boot

- Fault isolation

- Each containerized application is isolated and operates independently of others
- The failure of one container does not affect the continued operation of any other containers
- Development teams can identify and correct any technical issues within one container without any downtime in other containers
- Also, the container engine can leverage any OS security isolation techniques—such as SELinux access control—to isolate faults within containers

Benefits

- Efficiency

- Software running in containerized environments shares the machine's OS kernel, and application layers within a container can be shared across containers
- Thus, containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM. This drives higher server efficiencies, reducing server and licensing costs

- Ease of management

- Container orchestration platform automates the installation, scaling, and management of containerized workloads and services
- Container orchestration platforms can ease management tasks such as scaling containerized apps, rolling out new versions of apps, and providing monitoring, logging and debugging, among other functions. Kubernetes, perhaps the most popular container orchestration system available, is an open source technology (originally open-sourced by Google, based on their internal project called Borg) that automates Linux container functions originally
- Kubernetes works with many container engines, such as Docker, but it also works with any container system that conforms to the Open Container Initiative (OCI) standards for container image formats and runtimes

- Security

- The isolation of applications as containers inherently prevents the invasion of malicious code from affecting other containers or the host system
- Additionally, security permissions can be defined to automatically block unwanted components from entering containers or limit communications with unnecessary resources

Docker

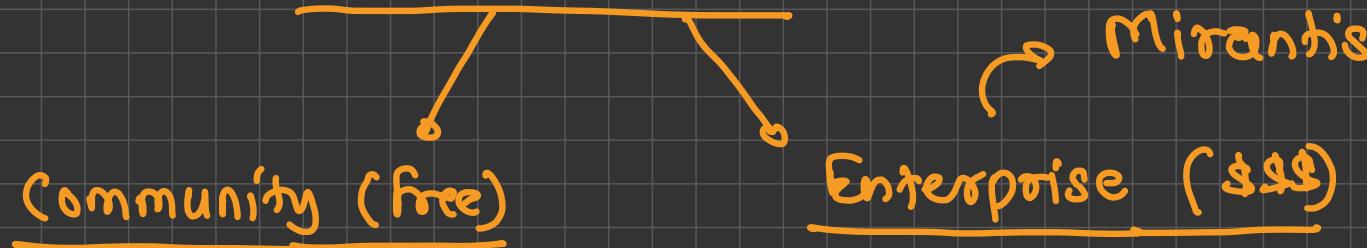
What is Docker? (Container Runtime / platform)

- Docker is an open source **platform** that enables developers to build, deploy, run, update and manage containers—standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment
 - Why docker?**
 - It is an easy way to create application **deployable packages**.
 - Developer can create ready-to-run containerized applications
 - It provides consistent computing environment
 - It works equally well in on-prem as well as cloud environments
 - It is light weight compared to VM
-
- The diagram illustrates the concept of Docker images. In the center, the word "images" is written in orange. Two arrows point from this central term to the right, each leading to a text label: "on premise" above and "on cloud" below. Both labels are written in orange.

Little history about Docker

- Docker Inc, started by Solomon Hykes, is behind the docker tool
- Docker Inc started as Paas provider called as dotCloud
- In 2013, the dotCloud became Docker Inc
- Docker Inc was using LinuX Containers (LXC) before version 0.9
- After 0.9 (2014), Docker replaced LXC with its own library libcontainer which is developed in Go programming language
- Its not the only solution for containerization
 - “FreeBSD Jails”, launched in 2000
 - LXD is next generation system container manager build on top of LXC and REST APIs
 - Google has its own open source container technology lmctfy (Let Me Contain That For You)
 - Rkt is another option for running containers
 - RedHat has its own tool podman

Docker Versions

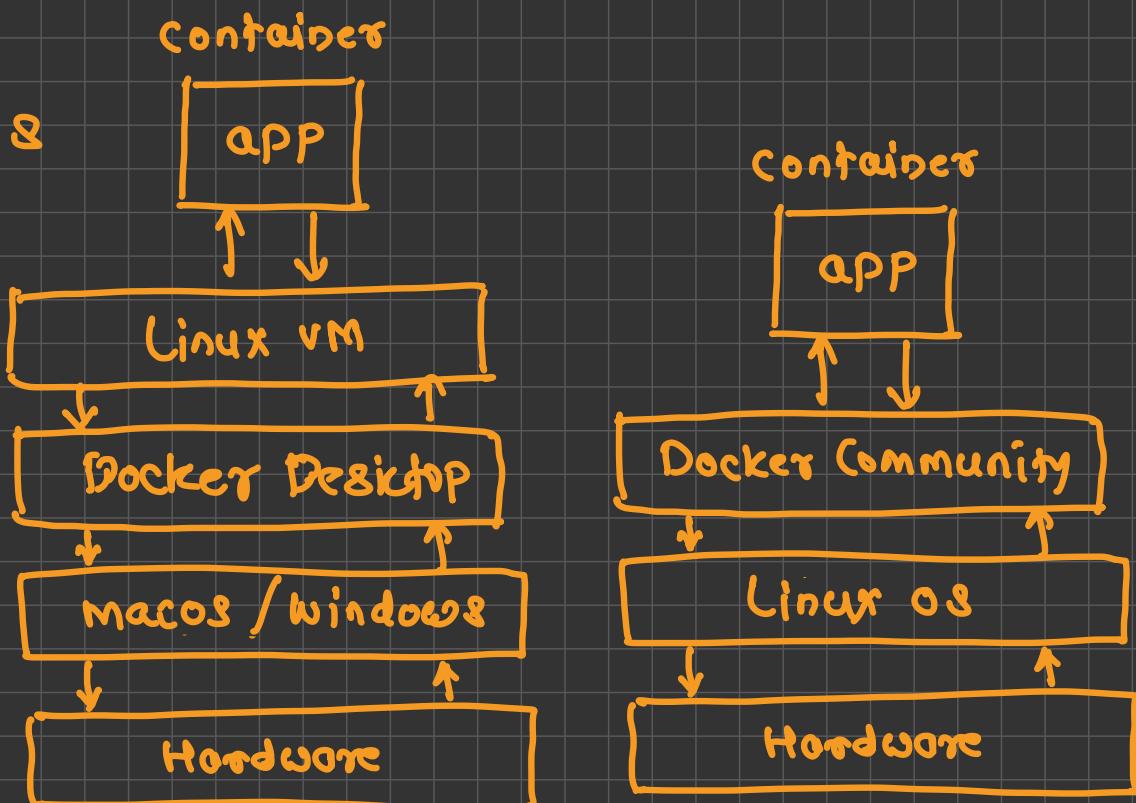


→ Linux community Edition

↳ Linux os has Linux kernel

→ Docker Desktop

↳ macos and windows



Docker Architecture → client-server architecture

→ server

Docker daemon (dockerd)

- Continuous running process
- Manages the containers

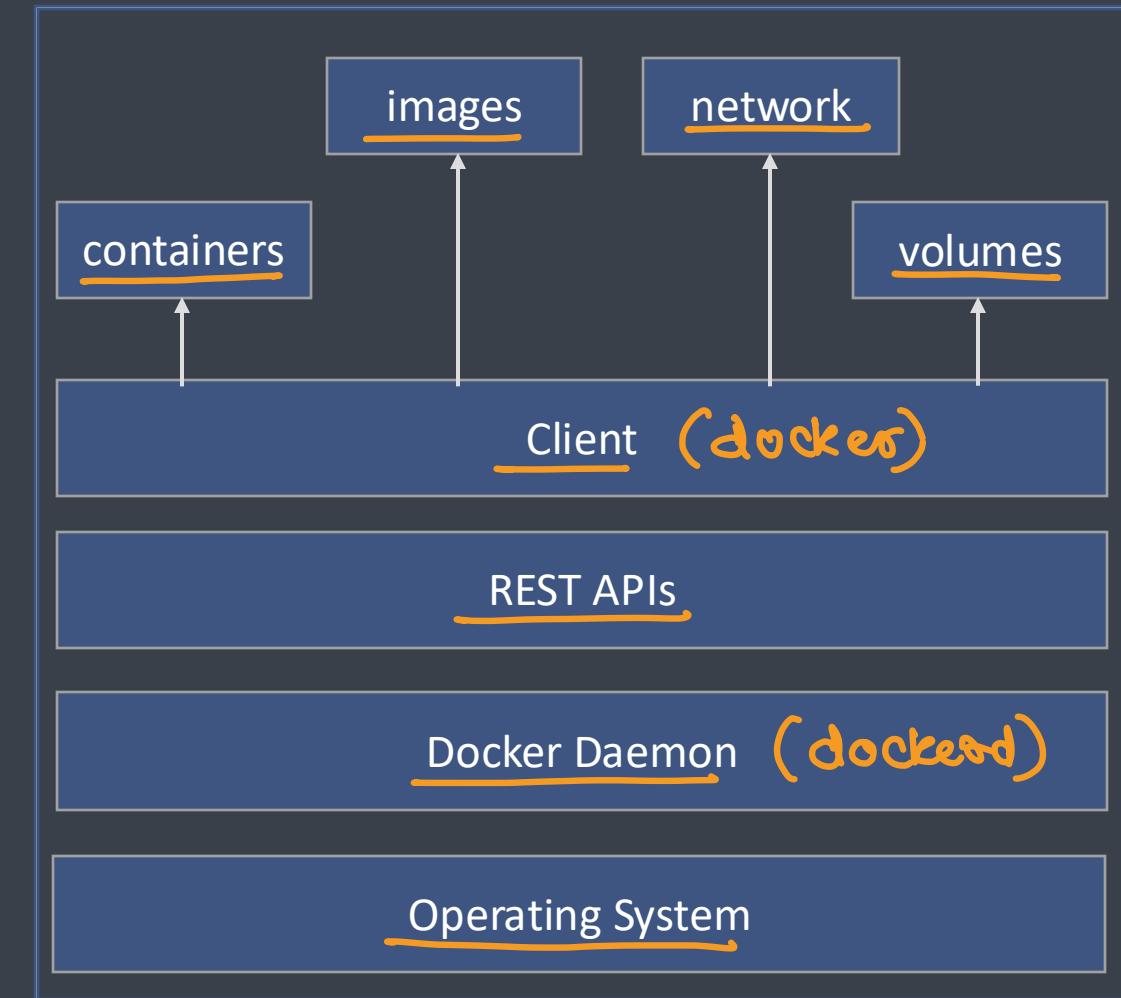
REST APIs

- Used to communicate with docker daemon

Client (docker)

- Provides command line interface
- Used to perform all the tasks

background processes
↳ no UI



libcontainer

- Docker has replaced LXC by libcontainer, which is used to manage the containers
- Libcontainer uses
 - Namespaces
 - Creates isolated workspace which limits what container can see
 - Provides a layer of isolation to the container
 - Each container runs in a separate namespace
 - Processes running in a namespace can interact with other processes or use resources which are the part of the same namespace
 - E.g. process ID, network, IPC, Filesystem
 - Control Groups (cgroups)
 - Used to share the available resources to the containers
 - It optionally enforces limits and constraints on resource usage
 - It limits how much a container can use
 - E.g. CPU, Disk space, memory

libcontainer

- Union File System (UnionFS)

- It uses layers
- It is a lightweight and very fast FS
- Docker uses different variants of UnionFS
 - Aufs (advanced multi-layered unification filesystem)
 - Btrfs (B-Tree FS)
 - VFS (Virtual FS)
 - Devicemapper

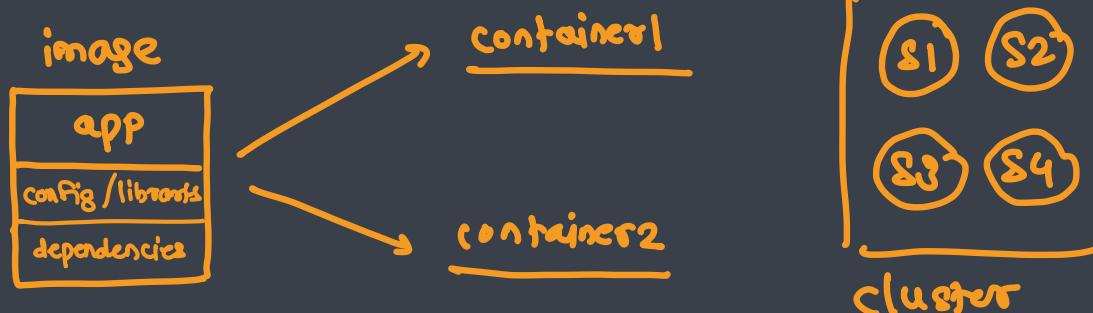
Docker Objects (modules)

- ✓ Images: read only template with instructions for creating docker containers
- ✓ Container: running instance of a docker image
- Network: network interface used to connect the containers to each other or external networks
- Volumes: used to persist the data generated by and used by the containers
- Registry: private or public collection of docker images
- Service: used to deploy application in a docker multi node cluster

What is Docker image ?

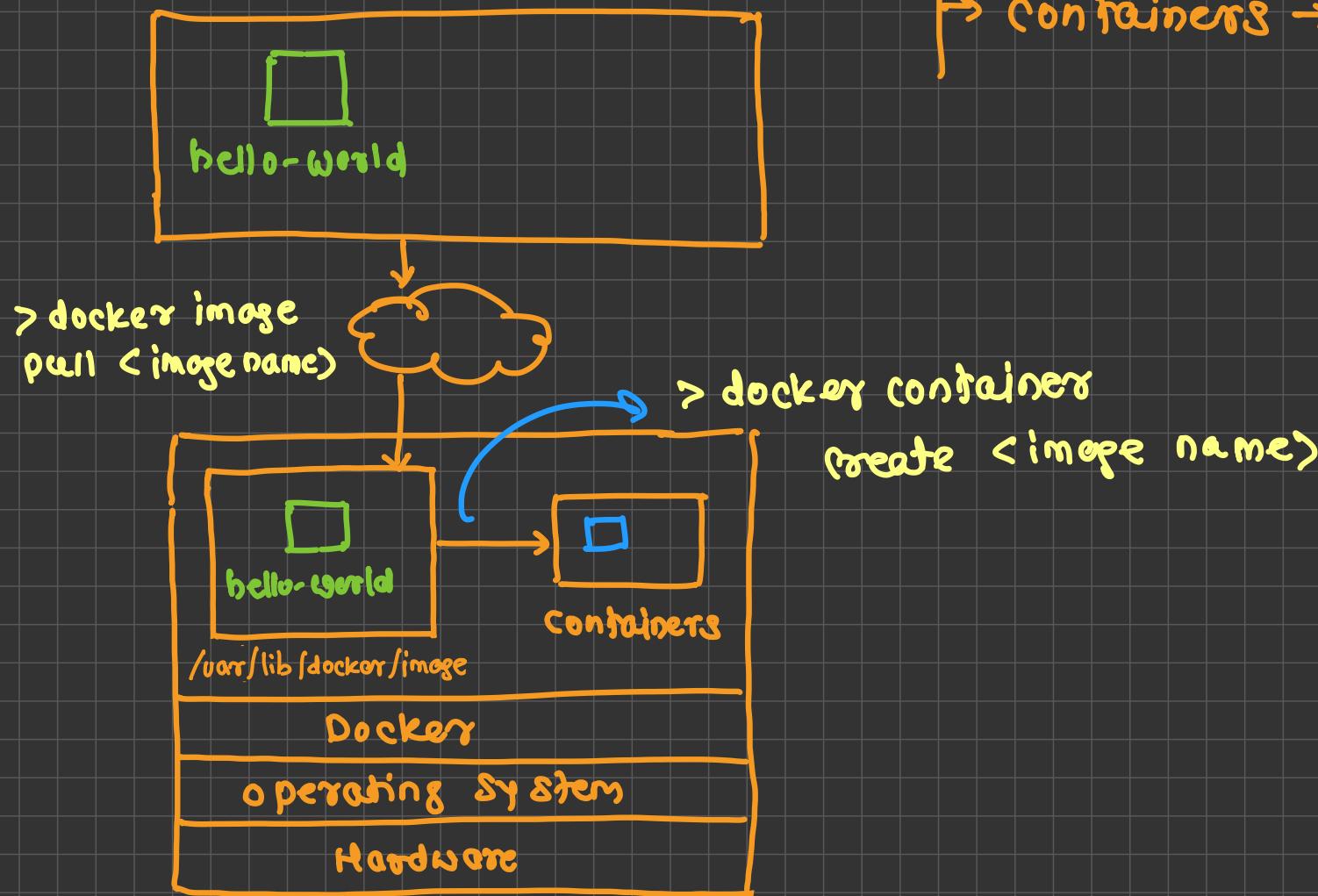
image (class) → container (object) → immutable

- A Docker image is a file used to execute code in a Docker container
- Docker images act as a set of instructions to build a Docker container, like a template
- Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual machine (VM) environments
- A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container
- Docker images have multiple layers, each one originates from the previous layer but is different from it
- The layers speed up Docker builds while increasing reusability and decreasing disk use
- Image layers are also read-only files
- Once a container is created, a writable layer is added on top of the unchangeable images, allowing a user to make changes



Docker Workflow

(hub.docker.com)
Docker image Registry



docker Home = /var/lib/docker

- image → contains images
- volumes → contains volumes
- networks → contains n/w objects
- containers → contains containers

→ app
→ libraries / frameworks
→ config files
→ dependencies

unit which contains

Docker Container

↳ created using image

Docker Container (process)

- It is a running **aspect of docker image**
- Contains one or more running processes
- It is a **self-contained** environment
- It wraps up an application into its own **isolated box** (application running inside a container has no knowledge of any other applications or processes that exist outside the container)
- A container can not modify the image from which it is created
- **It consists of**
 - Your application code
 - Dependencies
 - Networking
 - Volumes
- Containers are stored under /var/lib/docker /containers
- This directory contains images, containers, network volumes etc

→ **sandbox**

Basic Operations

- Creating container
- Starting container
- Running container
- Listing running containers
- Listing all containers
- Getting information of a container
- Stopping container
- Deleting container

Attaching a container

- There are two ways to attach to a container
- Attach
 - Used to attach the container
 - Uses only one input and output stream
 - Task
 - Attach to a running container
- Exec
 - Mainly it is used for running a command inside a container
 - Task
 - Execute a command inside container

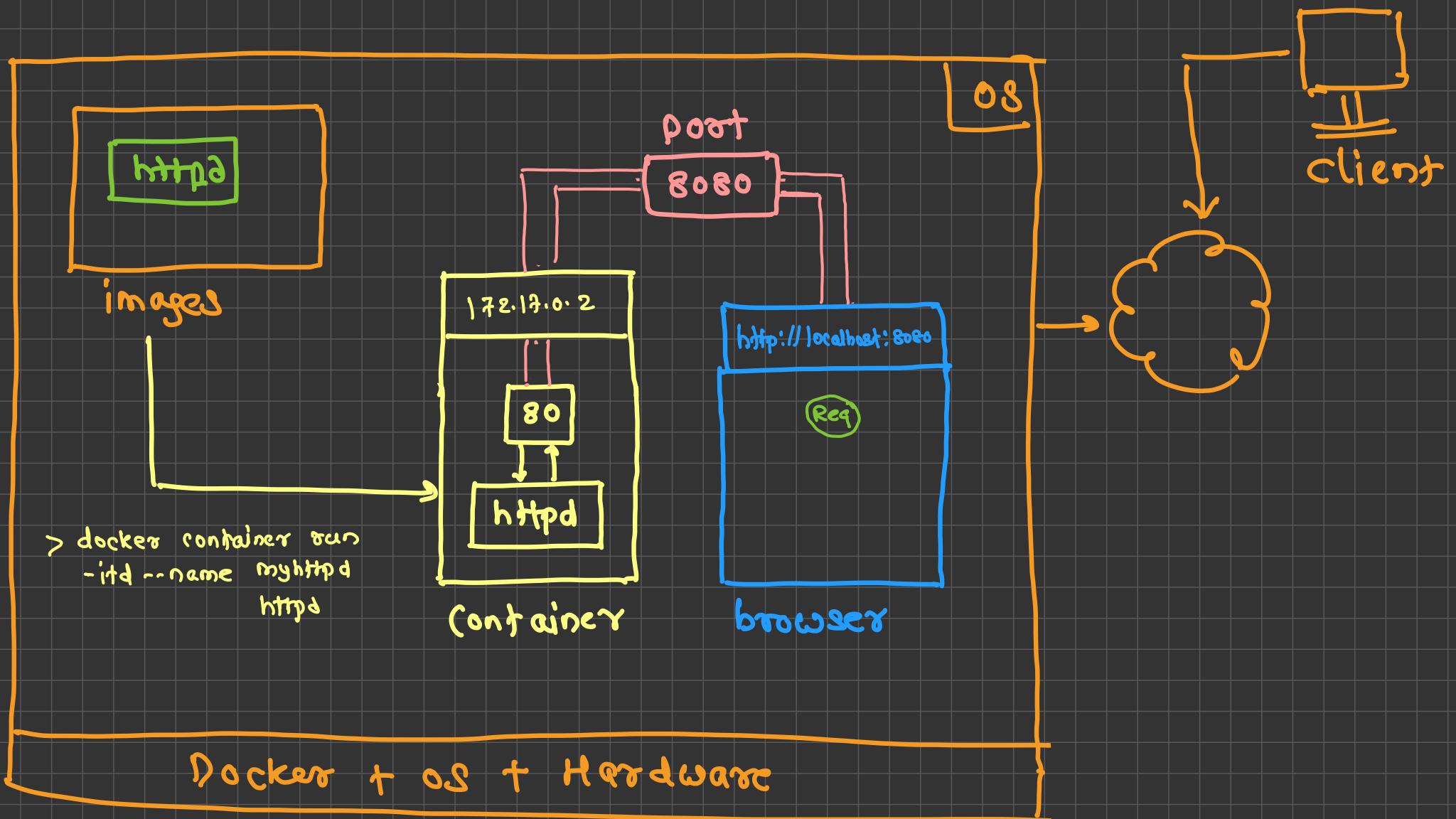
Hostname and name of container

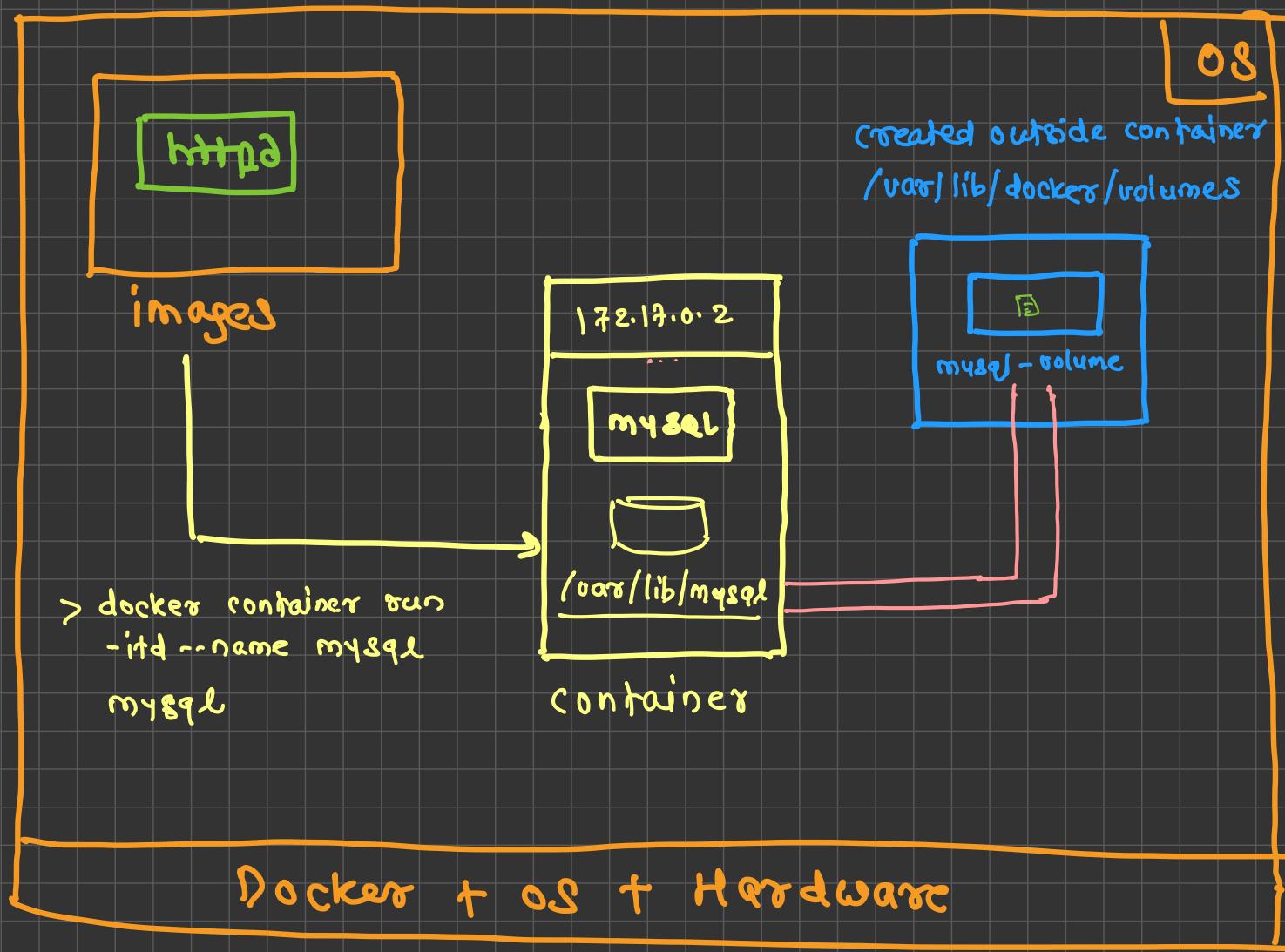
- To check the host name
 - Go inside the container
 - Check the hostname by using a command `hostname`
- Docker uses the first 12 characters of container id as hostname
- Docker automatically generates a name at random for each container

Publishing port on container

- Publishing a port is required to give an external access to your application *running inside container*
- Port can be published only at the time of creating a container
- You can not update the port configuration on running container
- Task
 - Run a httpd container with port 8080 published, to access apache externally

Port forwarding: forwarding request from one port to another

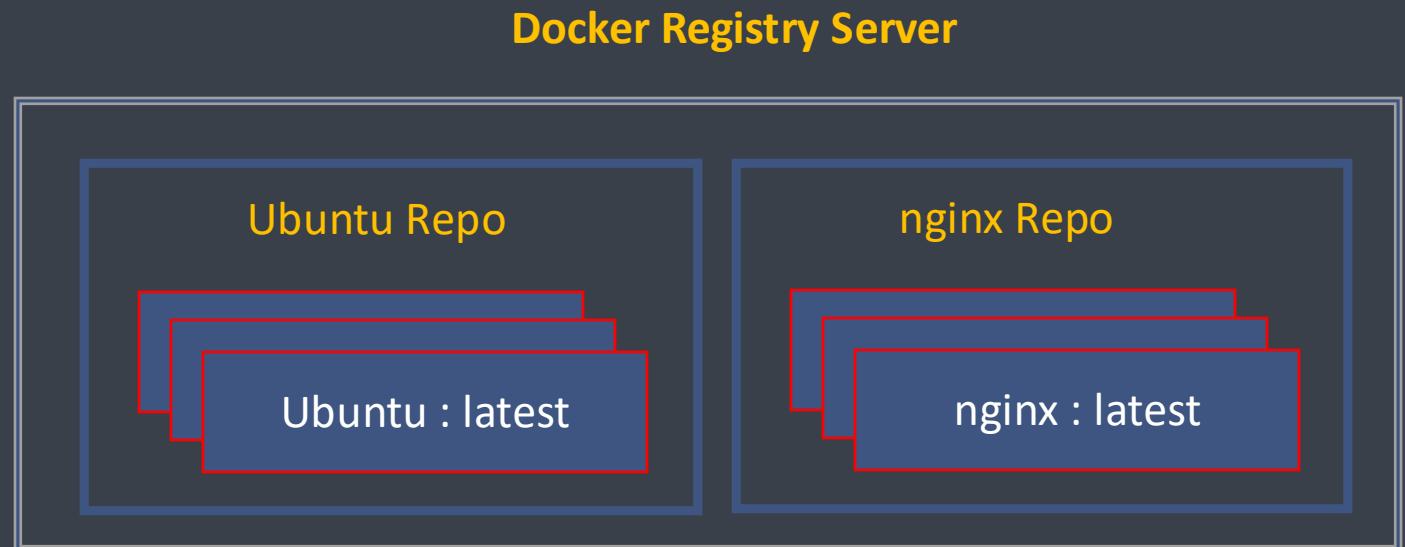




Docker Images (Advanced)

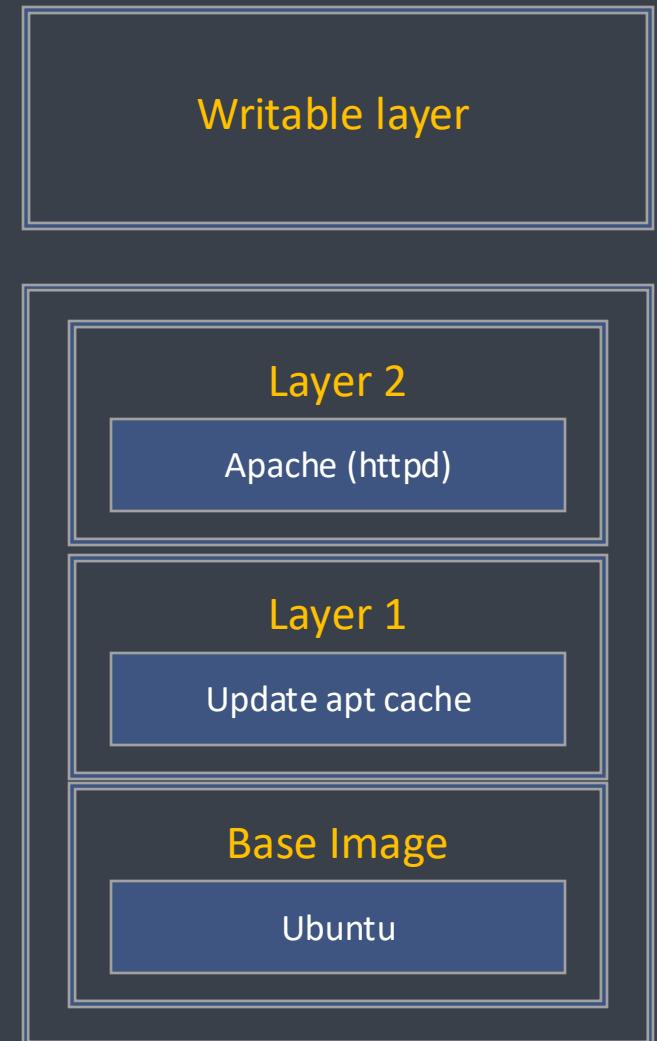
Docker Image

- Read-only instructions to run the containers
- It is made up of different layers
- Repositories hold images
- Docker registry stores repositories
- To create a custom image
 - Commit the running container
 - Use a Dockerfile
- Task
 - Create a container
 - Create a directory and a file within it
 - Commit the container to create a new image



Layered File System

- Docker images are made of layered FS
- Docker uses UnionFS for implementing the layered docker images
- Any update on the image adds a new layer
- All changes made to the running container are written inside a writable layer



Dockerfile

- The Dockerfile contains a series of instructions paired with arguments
- Each instruction should be in upper-case and be followed by an argument
- Instructions are processed from top to bottom
- Each instruction adds a new layer to the image and then commits the image
- Upon running, changes made by an instruction make it to the container

Dockerfile instructions

- FROM
- ENV
- RUN
- CMD
- EXPOSE
- WORKDIR
- ADD
- COPY
- LABEL
- MAINTAINER
- ENTRYPOINT

Orchestration

Container Orchestration

- Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments
- Software teams use container orchestration to control and automate many tasks
 - Provisioning and deployment of containers
 - Redundancy and availability of containers
 - Scaling up or removing containers to spread application load evenly across host infrastructure
 - Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
 - Allocation of resources between containers
 - External exposure of services running in a container with the outside world
 - Load balancing of service discovery between containers
 - Health monitoring of containers and hosts
 - Configuration of an application in relation to the containers running it
- Orchestration Tools
 - Docker Swarm
 - Kubernetes
 - Mesos
 - Marathon

Docker Swarm

- Docker Swarm is a container orchestration engine
- It takes multiple Docker Engines running on different hosts and lets you use them together
- The usage is simple: declare your applications as stacks of services, and let Docker handle the rest
- It is secure by default
- It is built using Swarmkit

What is a swarm?

- A swarm consists of multiple Docker hosts which run in **swarm mode**
- A given Docker host can be a manager, a worker, or perform both roles
- When you create a service, you define its optimal state
- Docker works to maintain that desired state
 - For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes
- A *task* is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container
- When Docker is running in swarm mode, you can still run standalone containers on any of the Docker hosts participating in the swarm, as well as swarm services
- A key difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any daemon

Features

- Cluster management integrated with Docker Engine
- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing
- Secure by default
- Rolling updates

Nodes

- A **node** is an instance of the Docker engine participating in the swarm
- You can run one or more nodes on a single physical computer or cloud server
- To deploy your application to a swarm, you submit a service definition to a **manager node**
- **Manager Node**
 - The manager node dispatches units of work called **tasks** to worker nodes
 - Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm
 - Manager nodes elect a single leader to conduct orchestration tasks
- **Worker nodes**
 - Worker nodes receive and execute tasks dispatched from manager nodes
 - An agent runs on each worker node and reports on the tasks assigned to it
 - The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker

Services and tasks

■ Service

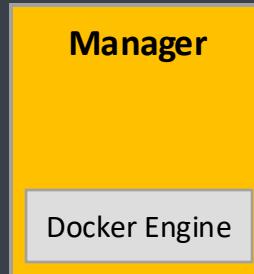
- A service is the definition of the tasks to execute on the manager or worker nodes
- It is the central structure of the swarm system and the primary root of user interaction with the swarm
- When you create a service, you specify which container image to use and which commands to execute inside running containers

■ Task

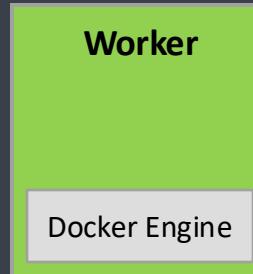
- A task carries a Docker container and the commands to run inside the container
- It is the atomic scheduling unit of swarm
- Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale
- Once a task is assigned to a node, it cannot move to another node
- It can only run on the assigned node or fail

Create a Swarm and add workers to swarm

- In our swarm we are going to use 3 nodes (one manager and two workers)
- Every node must have Docker Engine 1.12 or newer installed
- Following ports are used in node communication
 - TCP port 2377 is used for cluster management communication
 - TCP and UDP port 7946 is used for node communication
 - UDP port 4789 is used for overlay network traffic



```
docker swarm init --advertise-addr <ip>
```



```
docker swarm join --token <token>
```

Swarm Setup

- **Create swarm**

```
> docker swarm init --advertise-addr <MANAGER-IP>
```

- **Get current status of swarm**

```
> docker info
```

- **Get the list of nodes**

```
> docker node ls
```

Swarm Setup

- **Get token (on manager node)**

> `docker swarm join-token worker`

- **Add node (on worker node)**

> `docker swarm join --token <token>`

Overlay Network

- It is a computer network built on top of another network
- Sits on top of the host-specific networks and allows container, connected to it, to communicate securely
- When you initialize a swarm or join a host to swarm, two networks are created
 - An overlay network called as ingress network
 - A bridge network called as docker_gwbridge
- Ingress network facilitates load balancing among services nodes
- Docker_gwbridge is a bridge network that connect overlay networks to individual docker daemon's physical network

Service

- Definition of tasks to execute on Manager or Worker nodes
- Declarative Model for Services
- Scaling
- Desired state reconciliation
- Service discovery
- Rolling updates
- Load balancing
- Internal DNS component

Swarm Service

- **Deploy a service**

> `docker service create --replicas <no> --name <name> -p <ports> <image> <command>`

- **Get running services**

> `docker service ls`

- **Inspect service**

> `docker service inspect <service>`

- **Get the nodes running service**

> `docker service ps <service>`

Swarm Service

- **Scale service**

> `docker service scale <service>=<scale>`

- **Update service**

> `docker service update --image <image> <service>`

- **Delete service**

> `docker service rm <service>`

docker

docker generic commands

```
# get a list of objects  
# > docker <object> ls  
  
# get information about selected object  
# > docker <object> inspect <object name or id>  
  
# remove selected object  
# > docker <object> rm <object name or id>  
  
# remove unused objects  
# > docker <object> prune
```

docker images

- image registry
 - collection of docker images
 - types
 - custom or private
 - can be hosted by organizations or companies
 - public
 - hosted by docker at hub.docker.com

```
# list available images  
> docker image ls  
  
# download an image from docker image registry  
# > docker image pull <image name>  
> docker image pull hello-world  
  
# get the information about the selected image  
# > docker image inspect <image name>  
> docker image inspect hello-world  
  
# delete selected image  
# > docker image rm <image name or id>  
> docker image rm hello-world  
  
# delete unused images  
> docker image prune
```

docker containers

- a container can run only one application at a time
- when the application stops (either because the application is finished with its execution or got terminated because of any error), the container gets exited
- containers are not meant to persist the data (as the data will be stored inside the container till it is alive)
- if the container gets removed, all the data stored inside the container will be removed

```
# get the list of running containers
> docker container ls

# get the list of all containers
# statuses: Created, Running (Up), Exited
> docker container ls -a

# create a new container
# every new container gets
# - a unique id assigned
# - a random but unique name assigned
# > docker container create <image name or id>
> docker container create hello-world

# start a container
# > docker container start <container name or id>

# stop a running container
# > docker container stop <container name or id>

# get the logs from a container
# > docker container logs <container name or id>

# get the information about the container
# > docker container inspect <container name or id>

# delete exited container
# > docker container rm <container name or id>

# delete a running container
# > docker container rm --force <container name or id>
# > docker container rm -f <container name or id>

# run a container in attached mode (in foreground)
# - run = create + start
# > docker container run <image name or id>
> docker container run httpd

# run a container
# params
```

```
# - -d: run the container in detached mode (in background)
# - -i: run the container in interacting mode
# - -t: enable the container to get the terminal from it
# - --name: name of the container
# - -p: used to publish a port on the container
# - -e: used to set an environment variable
# - -v: used to attach a volume on the container
# > docker container run -d -i -t --name <container name> -p <source
# port>:<destination port> -v <volume name>:<container mount path> <image
# name or id>
# > docker container run -d -i -t --name myhttpd -p 8080:80 httpd
> docker container d-itd --name myhttpd -p 8080:80 httpd

# create a mysql container
> docker container run -itd --name mysql -p 3306:3306 -e
MySQL_ROOT_PASSWORD=root -v mysql-volume:/var/lib/mysql mysql

# execute a command inside a container
# > docker container exec <container name or id> <command>
> docker container exec myhttpd date

# get a terminal from the container
# > docker container exec -it <container name or id> <bash or sh>
> docker container exec -it myhttpd bash

# delete unused containers
> docker container prune
```

docker volumes

```
# get the list of volumes
> docker volume ls

# create a new volume
# > docker volume create <volume name>

# delete a volume
# > docker volume rm <volume name>

# delete all unused volumes
> docker volume prune
```

docker

docker generic commands

```
# get a list of objects  
# > docker <object> ls  
  
# get information about selected object  
# > docker <object> inspect <object name or id>  
  
# remove selected object  
# > docker <object> rm <object name or id>  
  
# remove unused objects  
# > docker <object> prune
```

docker images

- image registry
 - collection of docker images
 - types
 - custom or private
 - can be hosted by organizations or companies
 - public
 - hosted by docker at hub.docker.com

```
# list available images  
> docker image ls  
  
# download an image from docker image registry  
# > docker image pull <image name>  
> docker image pull hello-world  
  
# get the information about the selected image  
# > docker image inspect <image name>  
> docker image inspect hello-world  
  
# delete selected image  
# > docker image rm <image name or id>  
> docker image rm hello-world  
  
# delete unused images  
> docker image prune
```

custom images

- to create a custom image, create a file named Dockerfile
- the custom image must be created using an existing image (it must have one base image)
- commands
 - FROM: used to specify the base image
 - COPY: used to copy resource(s) from local machine to image
 - syntax: COPY
 - EXPOSE: used to expose a port for container (used later in port forwarding)
 - RUN: used to execute command(s) while building the image
 - CMD: used to execute command when a new container starts
 - WORKDIR: used to set (or create if it does not exist) working directory
 - ENV: used to set an environment variable
- to push your image to docker hub
 - signup with docker hub (free)
 - rename the image with the following pattern
 - name: /
 - login with your name and password
 - push the image to the docker hub

```
# build the image
# > docker image build -t <image name> <context>
> docker image build -t mywebsite .

# create a reference to an image with different name
# > docker image tag <old image name> <new image name>
> docker image tag mywebsite amitksunbeam/mywebsite

# login with docker hub user name and password
> docker login

# push image to docker hub
# note: make sure the image name is in the pattern: <docker user
name>/<image name>
> docker image push <image name>
```

docker containers

- a container can run only one application at a time
- when the application stops (either because the application is finished with its execution or got terminated because of any error), the container gets exited

- containers are not meant to persist the data (as the data will be stored inside the container till it is alive)
- if the container gets removed, all the data stored inside the container will be removed

```
# get the list of running containers
> docker container ls

# get the list of all containers
# statuses: Created, Running (Up), Exited
> docker container ls -a

# create a new container
# every new container gets
# - a unique id assigned
# - a random but unique name assigned
# > docker container create <image name or id>
> docker container create hello-world

# start a container
# > docker container start <container name or id>

# stop a running container
# > docker container stop <container name or id>

# get the logs from a container
# > docker container logs <container name or id>

# get the information about the container
# > docker container inspect <container name or id>

# delete exited container
# > docker container rm <container name or id>

# delete a running container
# > docker container rm --force <container name or id>
# > docker container rm -f <container name or id>

# run a container in attached mode (in foreground)
# - run = create + start
# > docker container run <image name or id>
> docker container run httpd

# run a container
# params
# - -d: run the container in detached mode (in background)
# - -i: run the container in interacting mode
# - -t: enable the container to get the terminal from it
# - --name: name of the container
# - -p: used to publish a port on the container
# - -e: used to set an environment variable
# - -v: used to attach a volume on the container
```

```
# > docker container run -d -i -t --name <container name> -p <source
port>:<destination port> -v <volume name>:<container mount path> <image
name or id>
# > docker container run -d -i -t --name myhttpd -p 8080:80 httpd
> docker container d-itd --name myhttpd -p 8080:80 httpd

# create a mysql container
> docker container run -itd --name mysql -p 3306:3306 -e
MYSQL_ROOT_PASSWORD=root -v mysql-volume:/var/lib/mysql mysql

# execute a command inside a container
# > docker container exec <container name or id> <command>
> docker container exec myhttpd date

# get a terminal from the container
# > docker container exec -it <container name or id> <bash or sh>
> docker container exec -it myhttpd bash

# delete unused containers
> docker container prune
```

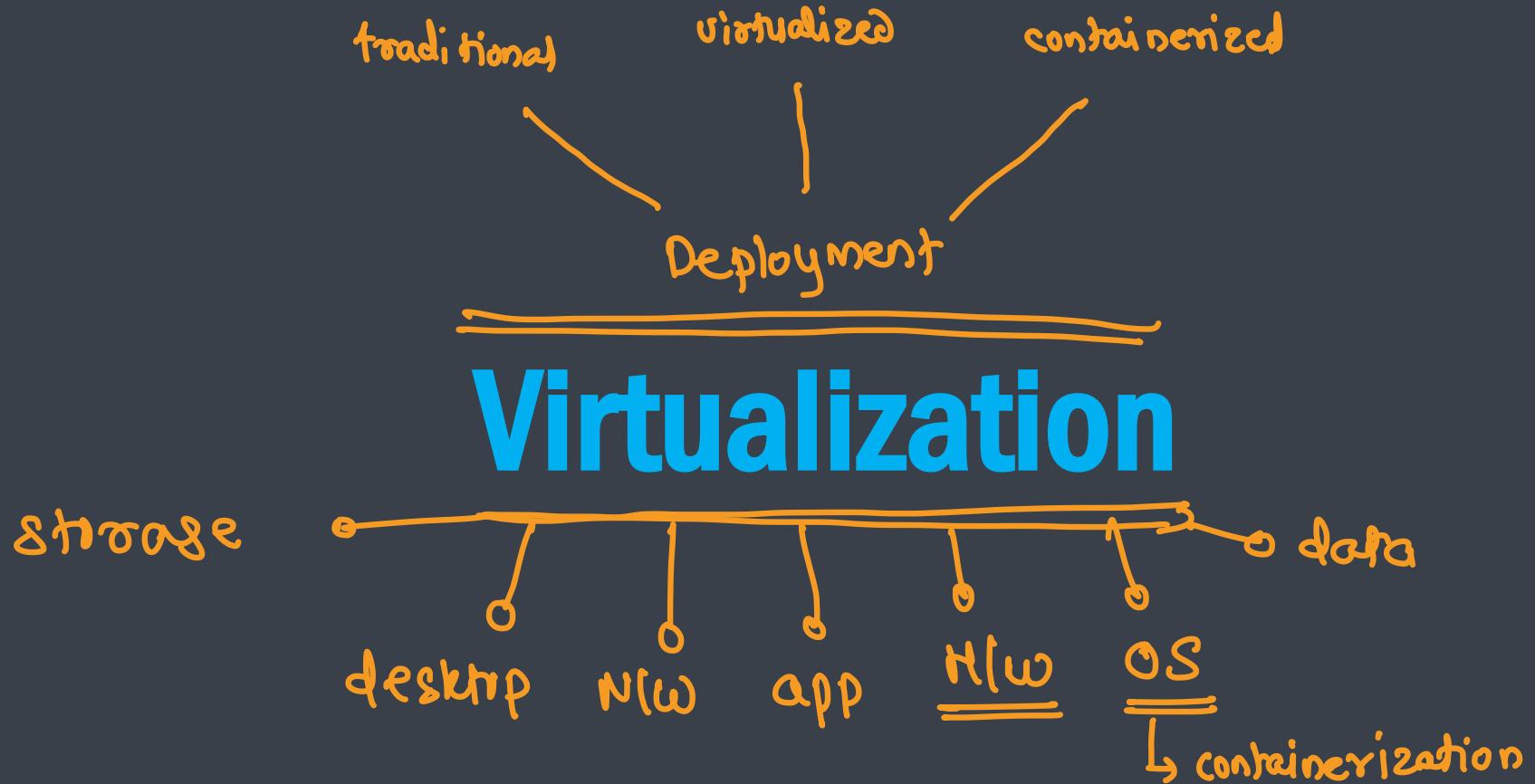
docker volumes

```
# get the list of volumes
> docker volume ls

# create a new volume
# > docker volume create <volume name>

# delete a volume
# > docker volume rm <volume name>

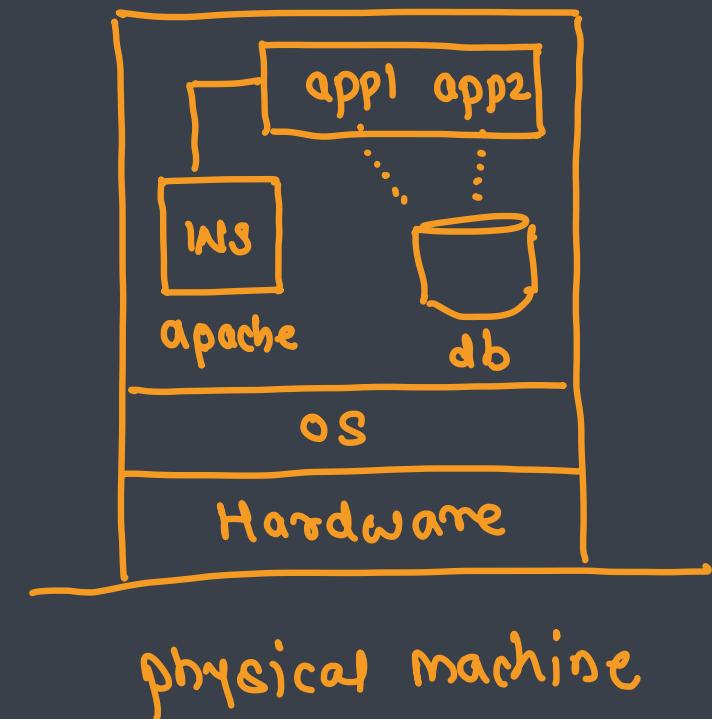
# delete all unused volumes
> docker volume prune
```

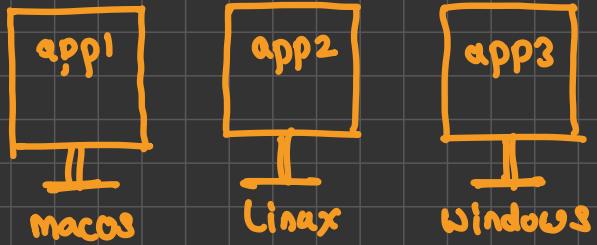


Traditional Deployment → deprecated

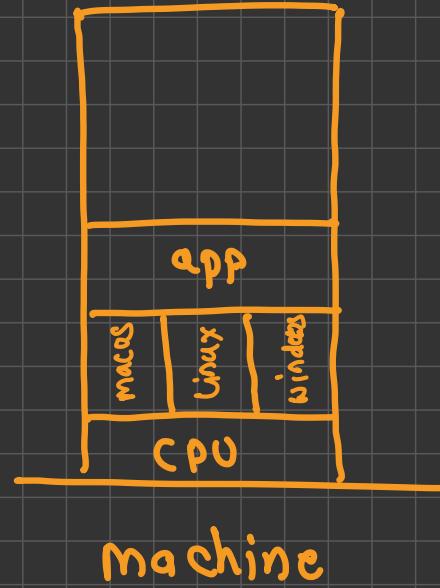
- Early on, organizations ran applications on physical servers
- There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues
- For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform
- A solution for this would be to run each application on a different physical server
- But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers

resource → CPU + memory





+ fastest option
- costliest



multi-booting machine

+ cheapest
- time [one os can run at a time]

What is virtualization

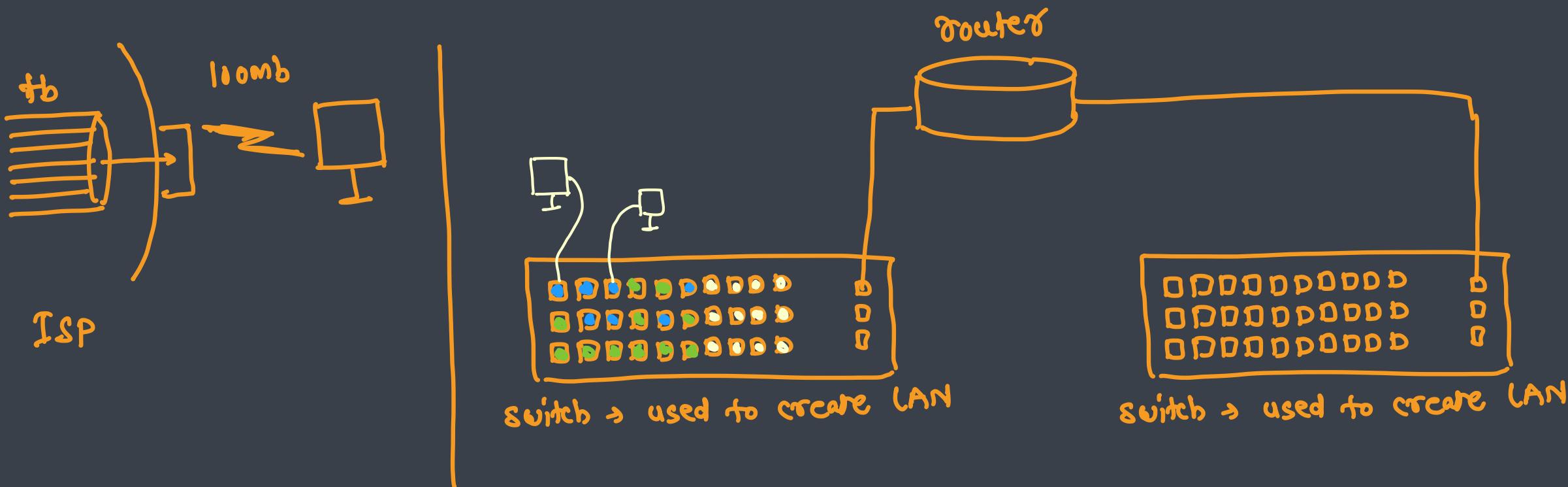
→ No physical existence

+ faster
+ run multiple OSes simultaneously
+ can be easily shared

- Virtualization is the creation of a virtual -- rather than actual -- version of something, such as an operating system (OS), a server, a storage device or network resources
- Virtualization uses software that simulates hardware functionality in order to create a virtual system
- This practice allows IT organizations to operate multiple operating systems, more than one virtual system and various applications on a single server
- Types
 - ① ▪ Network virtualization
 - ② ▪ Storage virtualization
 - ③ ▪ Data virtualization
 - ④ ▪ Desktop virtualization
 - ⑤ ▪ Application virtualization
 - ⑥ ▪ Hardware virtualization
- ⑦ OS virtualization
(containerization) * * *

Network Virtualization

- Network virtualization takes the available resources on a network and breaks the bandwidth into discrete channels
- Admins can secure each channel separately, and they can assign and reassign channels to specific devices in real time
- The promise of network virtualization is to improve networks' speed, availability and security, and it's particularly useful for networks that must support unpredictable usage bursts

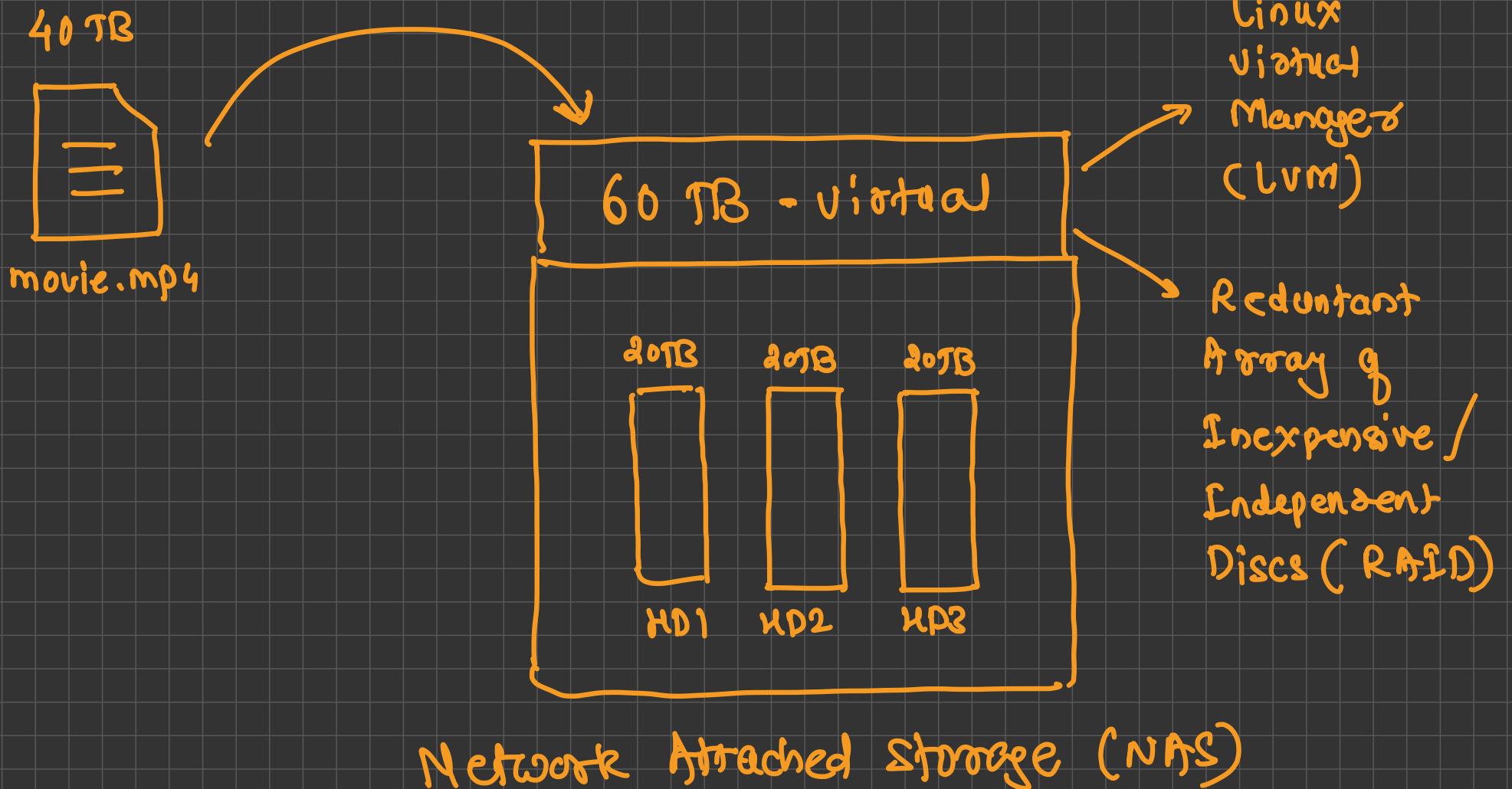


Storage Virtualization

- Storage virtualization is the pooling of physical storage from multiple network storage devices into what appears to be a single storage device that is managed from a central console
- Storage virtualization is commonly used in storage area networks
- Applications can use storage without having any concern for where it resides, what technical interface it provides, how it has been implemented, which platform it uses and how much of it is available
- Benefits
 - Makes the remote storage devices appear local
 - Multiple smaller volumes appear as a single large volume
 - Data is spread over multiple physical disks to improve reliability and performance
 - All operating systems use the same storage device
 - Provided high availability, disaster recovery, improved performance and sharing

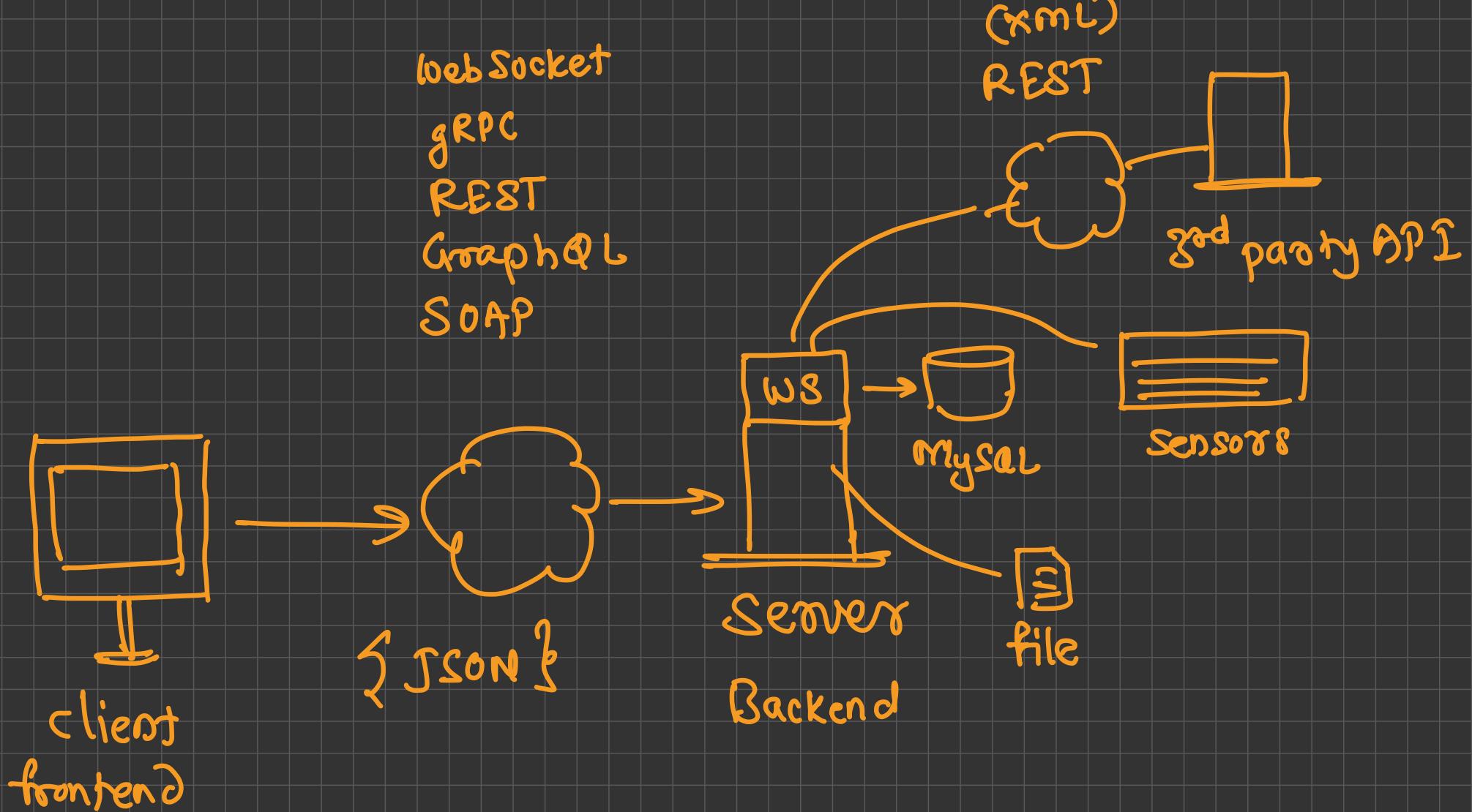


Storage virtualization



Data virtualization

- Data virtualization is the process of aggregating data from different sources of information to develop a single, logical and virtual view of information so that it can be accessed by front-end solutions such as applications, dashboards and portals without having to know the data's exact storage location
- The process of data virtualization involves abstracting, transforming, federating and delivering data from disparate sources
- The main goal of data virtualization technology is to provide a single point of access to the data by aggregating it from a wide range of data sources
- Benefits
 - Abstraction of technical aspects of stored data like APIs, Language, Location, Storage structure
 - Provides an ability to connect multiple data sources from a single location
 - Provides an ability to combine the data result sets across multiple sources (also known as data federation)
 - Provides an ability to deliver the data as requested by users

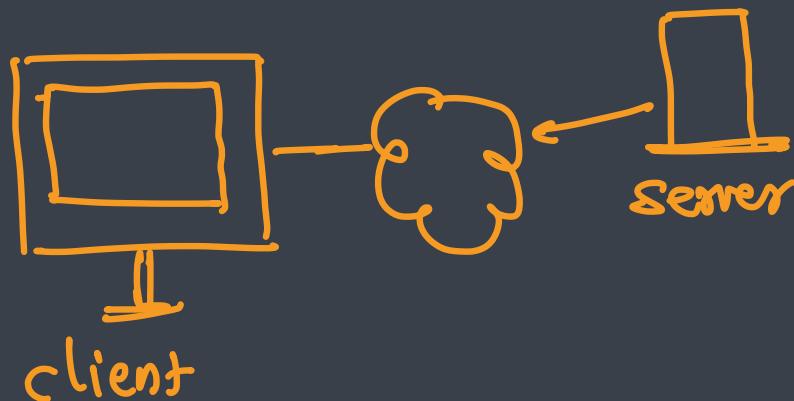


Desktop virtualization → Anydesk

- With desktop virtualization, the goal is to isolate a desktop OS from the endpoint that employees use to access it
- It provides an ability to connect to the desktop from remote site
- When multiple users connect to a shared desktop, as is the case with Microsoft Remote Desktop Services, it's known as shared hosted desktop virtualization

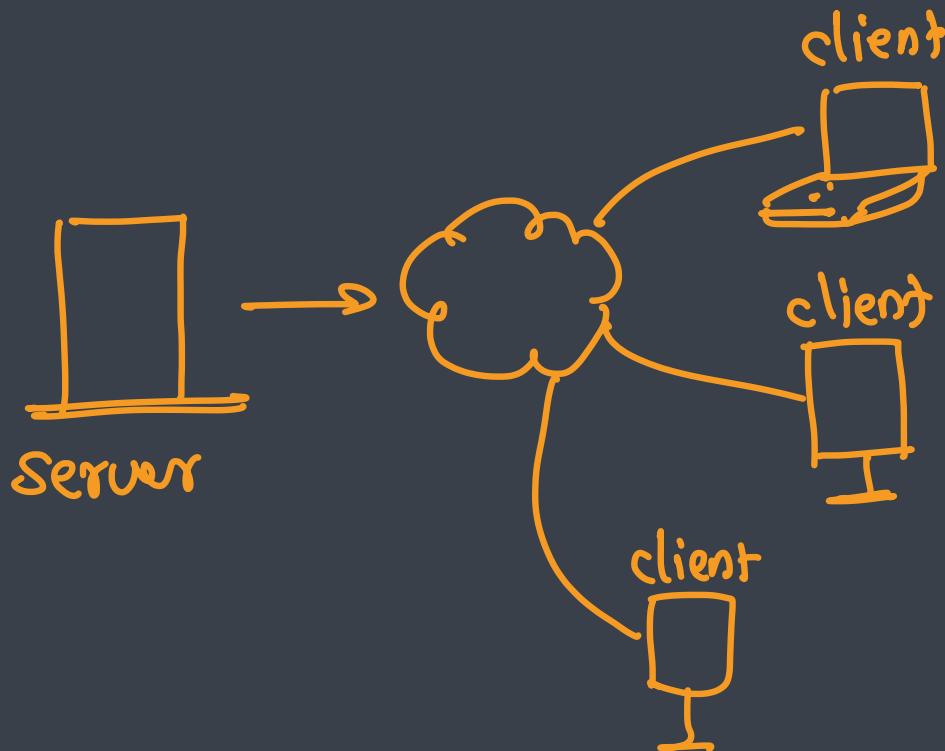
UNC → virtual Network Connection

RDC → Remote Desktop Connection



Application Virtualization

- With application virtualization, an app runs separately from the device that accesses it
- Application virtualization makes it possible for IT admins to install, patch and update only one version of an app rather than performing the same management tasks multiple times

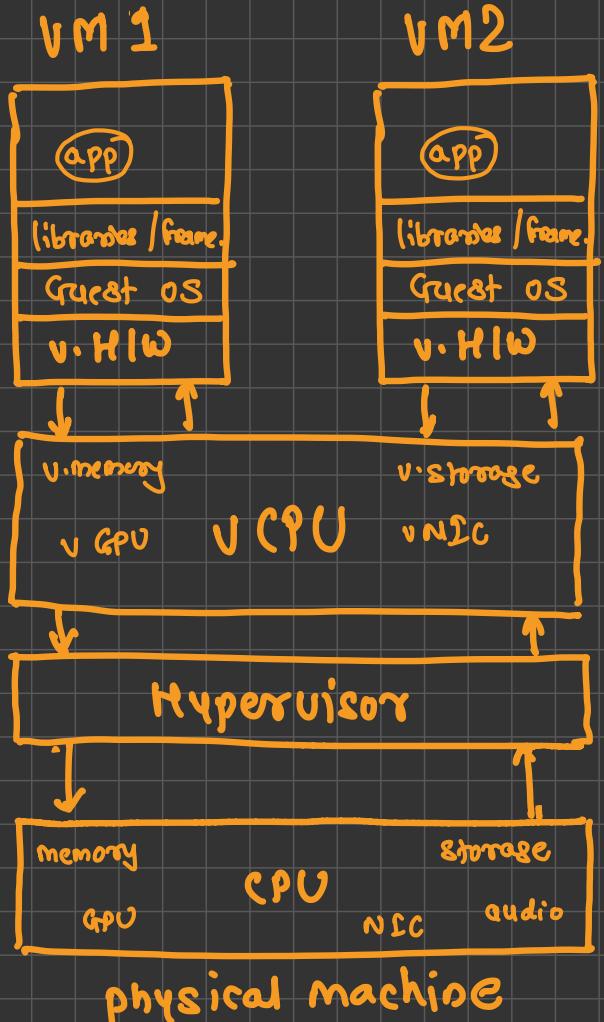


Hardware Virtualization

- Hardware virtualization or platform virtualization refers to the creation of a virtual machine that acts like a real computer with an operating system
- The process of masking the hardware resources like
 - CPU
 - Storage
 - Memory
- For example, a computer that is running Microsoft Windows may host a virtual machine that looks like a computer with the Ubuntu Linux operating system; Ubuntu-based software can be run on the virtual machine
- The process of creating Machines

Type I → Bare Metal Hypervisor

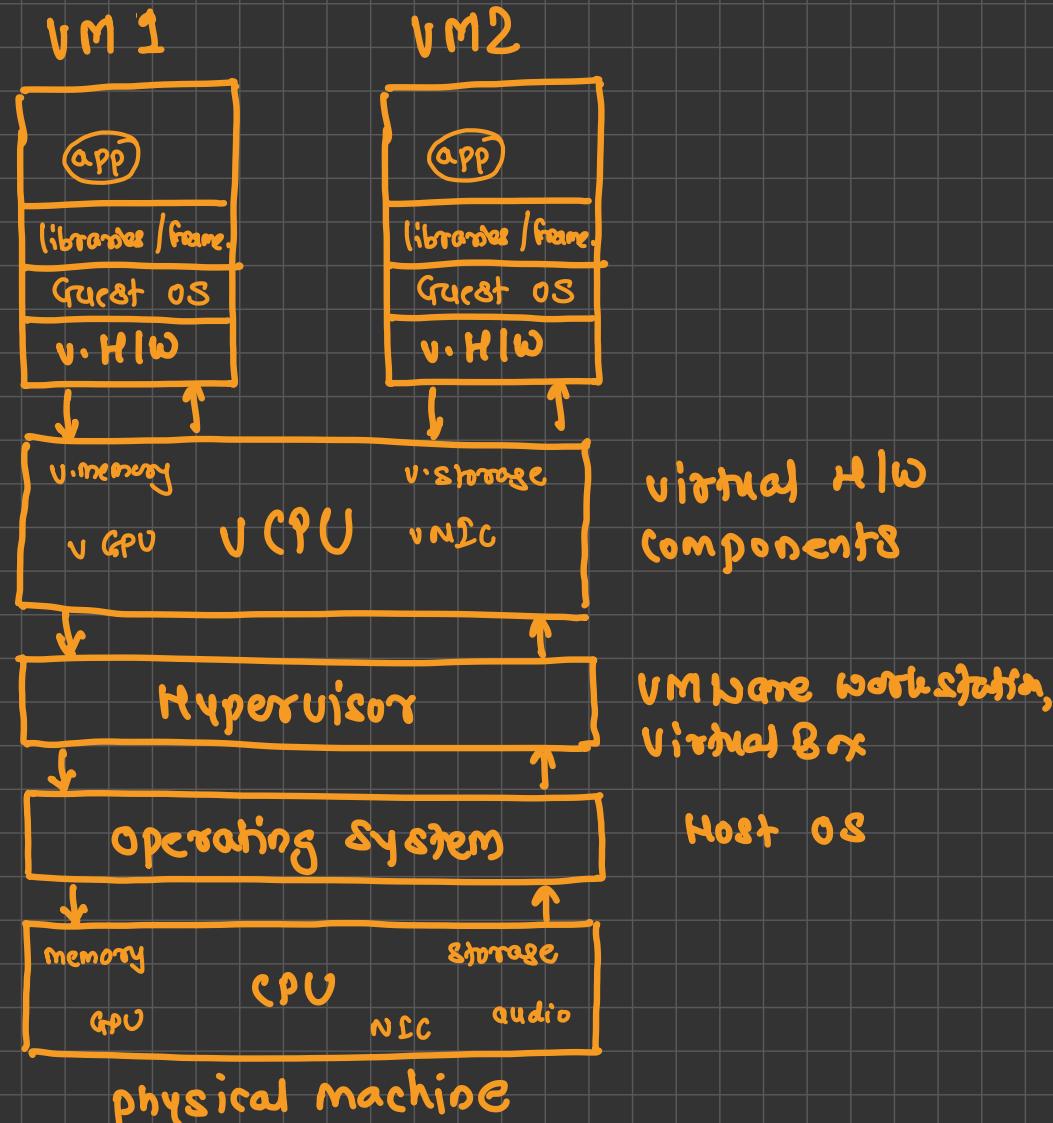
- Does Not require Host OS
- faster and utilizes resources in better ways compared with Type II
- used by cloud providers



VMware ESXi,
Xen, KVM

Type II → Hosted Hypervisor

- requires host OS
- requires more resources
- used by developers / testers



Host OS

virtual HW
components

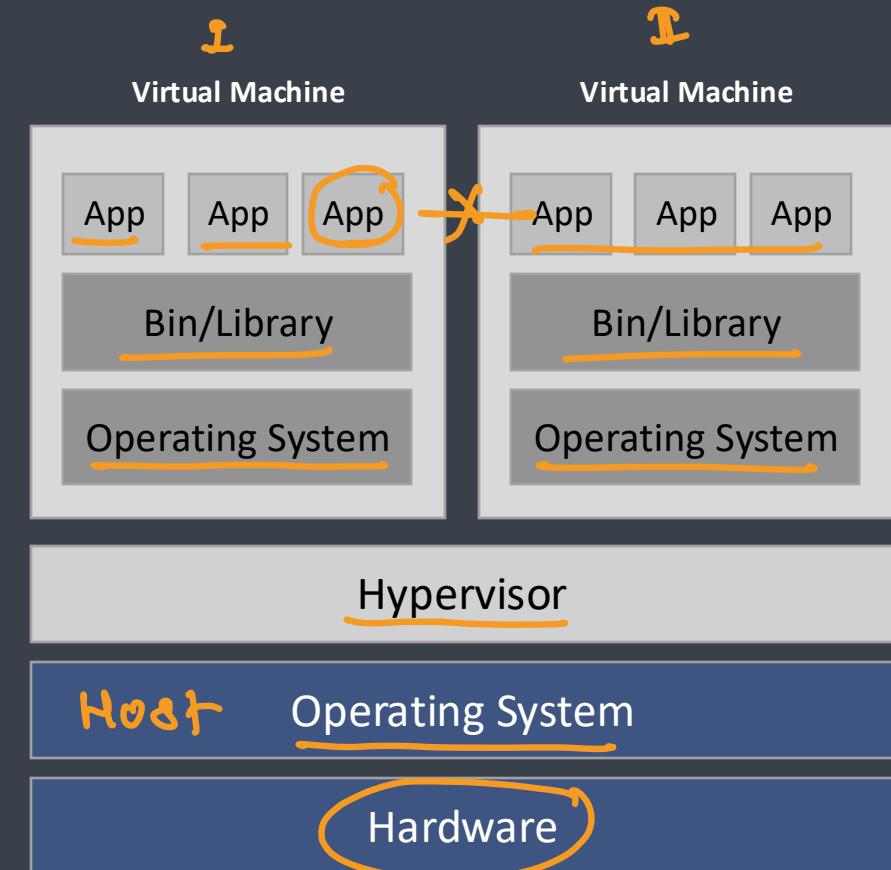
VMWare workstation,
Virtual Box

Virtual Machine

- A virtual machine is the emulated equivalent of a computer system that runs on top of another system
- Virtual machines may have access to any number of resources
 - Computing power - through hardware-assisted but limited access to the host machine's CPU
 - Memory - one or more physical or virtual disk devices for storage
 - A virtual or real network interfaces
 - Any devices such as
 - video cards,
 - USB devices,
 - other hardware that are shared with the virtual machine
- If the virtual machine is stored on a virtual disk, this is often referred to as a **disk image**

Virtualized Deployment

- It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU
- Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application
- Virtualization allows better utilization of resources in a physical server and allows better scalability because
 - an application can be added or updated easily
 - reduces hardware costs
- With virtualization you can present a set of physical resources as a cluster of disposable virtual machines
- Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware



Types of hardware virtualization

- **Type I**

- A Type 1 hypervisor runs directly on the host machine's physical hardware, and it's referred to as a bare-metal hypervisor
- It doesn't have to load an underlying OS first
- With direct access to the underlying hardware and no other software, it is more efficient and provides better performance
- It is best suited for enterprise computing or data centers
- E.g. VMware ESXi, Microsoft Hyper-V server and open source KVM

- **Type II**

- A Type 2 hypervisor is typically installed on top of an existing OS, and it's called a hosted hypervisor
- It relies on the host machine's pre-existing OS to manage calls to CPU, memory, storage and network resources
- E.g. VMware Fusion, Oracle VM VirtualBox, Oracle VM Server for x86, Oracle Solaris Zones, Parallels and VMware Workstation

Advantages of virtualization

- **Lower costs**

- Virtualization reduces the amount of hardware servers necessary within a company and data center
- This lowers the overall cost of buying and maintaining large amounts of hardware

- **Easier disaster recovery**

- Disaster recovery is very simple in a virtualized environment
- Regular snapshots provide up-to-date data, allowing virtual machines to be feasibly backed up and recovered
- Even in an emergency, a virtual machine can be migrated to a new location within minutes

- **Easier testing**

- Testing is less complicated in a virtual environment
- Even if a large mistake is made, the test does not need to stop and go back to the beginning
- It can simply return to the previous snapshot and proceed with the test.

- **Quicker backups**

- Backups can be taken of both the virtual server and the virtual machine
- Automatic snapshots are taken throughout the day to guarantee that all data is up-to-date
- Furthermore, the virtual machines can be easily migrated between each other and efficiently redeployed

- **Improved productivity**

- Fewer physical resources results in less time spent managing and maintaining the servers
- Tasks that can take days or weeks in a physical environment can be done in minutes
- This allows staff members to spend the majority of their time on more productive tasks, such as raising revenue and fostering business initiatives

Advantages of virtualization

- **Quicker backups**

- Backups can be taken of both the virtual server and the virtual machine
- Automatic snapshots are taken throughout the day to guarantee that all data is up-to-date
- Furthermore, the virtual machines can be easily migrated between each other and efficiently redeployed

- **Improved productivity**

- Fewer physical resources results in less time spent managing and maintaining the servers
- Tasks that can take days or weeks in a physical environment can be done in minutes
- This allows staff members to spend the majority of their time on more productive tasks, such as raising revenue and fostering business initiatives

frontend and backend

Monolithic Architecture



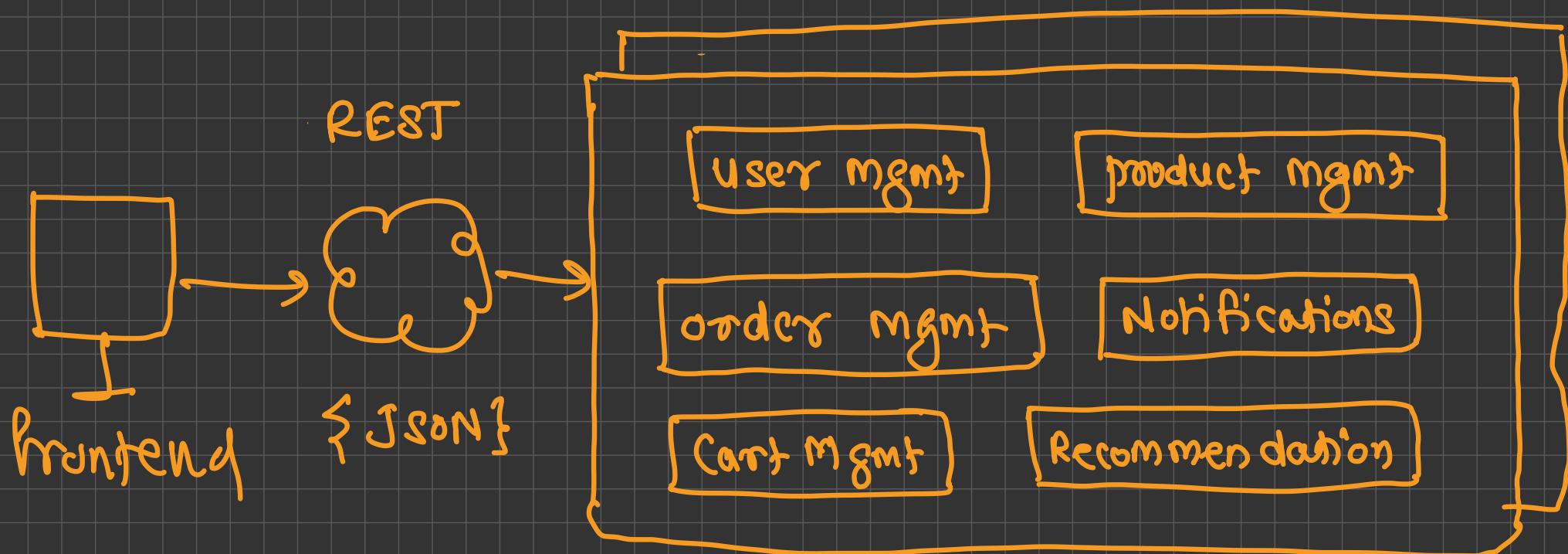
single application with all requirements

E-commerce

- user mgmt
- product mgmt
- order mgmt
- notifications
- recommendation
- cart mgmt

* Technical Stack

- programming language - JavaScript
- frame work - express / fastify / Hapi
- Database - RDBMS → MySQL
- Git Repository - Ecommerce-backend



What is Monolithic Architecture ?

single has all functionality

- A monolithic architecture is a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications.
- The word “monolith” is often attributed to something large and glacial, which isn’t far from the truth of a monolith architecture for software design
→ single language
- A monolithic architecture is a singular, large computing network with one code base that couples all of the business concerns together
- To make a change to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface
- Monoliths can be convenient early on in a project's life for ease of code management, cognitive overhead, and deployment
- This allows everything in the monolith to be released at once.

Pros

- **Easy deployment**
 - One executable file or directory makes deployment easier
- **Development**
 - When an application is built with one code base, it is easier to develop
- **Performance**
 - In a centralized code base and repository, one API can often perform the same function that numerous APIs perform with microservices
- **Simplified testing**
 - Since a monolithic application is a single, centralized unit, end-to-end testing can be performed faster than with a distributed application
- **Easy debugging**
 - With all code located in one place, it's easier to follow a request and find an issue.

Cons

- **Slower development speed**
 - A large, monolithic application makes development more complex and slower
- **Scalability**
 - You can't scale individual components
- **Reliability**
 - If there's an error in any module, it could affect the entire application's availability
- **Barrier to technology adoption**
 - Any changes in the framework or language affects the entire application, making changes often expensive and time-consuming
- **Lack of flexibility**
 - A monolith is constrained by the technologies already used in the monolith
- **Deployment**
 - A small change to a monolithic application requires the redeployment of the entire monolith

Microservices

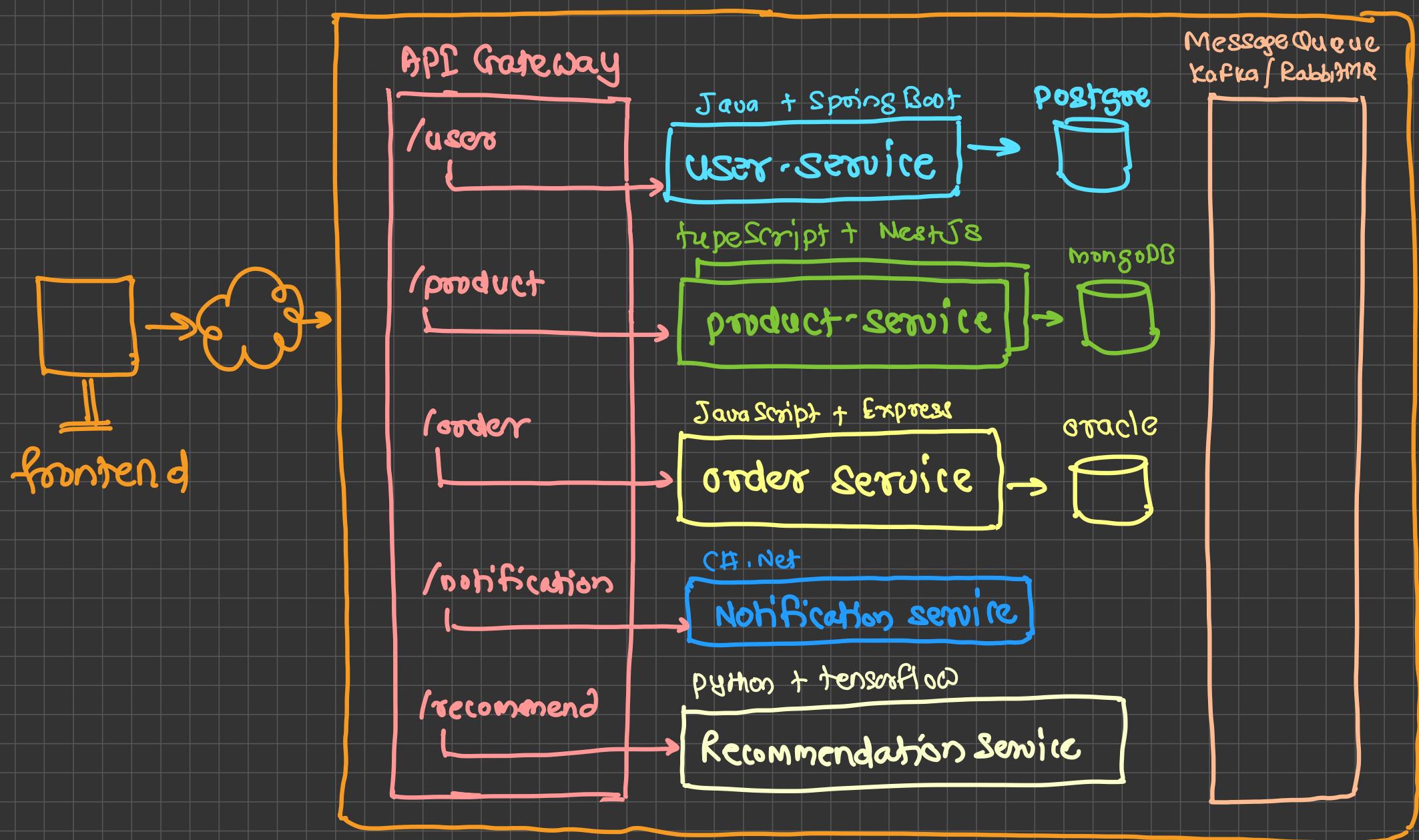
small applications

E-commerce

- user mgmt
 - ↳ Java / Spring Boot + PostgreSQL
- product mgmt
 - ↳ Typescript / NestJS + MongoDB
- order mgmt
 - ↳ JavaScript / Express + Oracle
- notifications
 - ↳ C# / .Net
- recommendation
 - ↳ python / tensorflow / pytorch

microservices

backend application - logical view



What is Microservices Architecture ?

- A microservices architecture, also simply known as microservices, is an architectural method that relies on a series of independently deployable services
- These services have their own business logic and database with a specific goal
- Updating, testing, deployment, and scaling occur within each service
- Microservices decouple major business, domain-specific concerns into separate, independent code bases
- Microservices don't reduce complexity, but they make any complexity visible and more manageable by separating tasks into smaller processes that function independently of each other and contribute to the overall whole
- Adopting microservices often goes hand in hand with DevOps, since they are the basis for continuous delivery practices that allow teams to adapt quickly to user requirements

Pros

- **Agility**
 - Promote agile ways of working with small teams that deploy frequently
- **Flexible scaling**
 - If a microservice reaches its load capacity, new instances of that service can rapidly be deployed to the accompanying cluster to help relieve pressure
 - We are now multi-tenant and stateless with customers spread across multiple instances. Now we can support much larger instance sizes
- **Continuous deployment**
 - We now have frequent and faster release cycles. Before we would push out updates once a week and now we can do so about two to three times a day.
- **Highly maintainable and testable**
 - Teams can experiment with new features and roll back if something doesn't work
 - This makes it easier to update code and accelerates time-to-market for new features. Plus, it is easy to isolate and fix faults and bugs in individual services
- **Independently deployable**
 - Since microservices are individual units they allow for fast and easy independent deployment of individual features.
- **Technology flexibility**
 - Microservice architectures allow teams the freedom to select the tools they desire
- **High reliability**
 - You can deploy changes for a specific service, without the threat of bringing down the entire application
- **Happier teams**
 - The Atlassian teams who work with microservices are a lot happier, since they are more autonomous and can build and deploy themselves without waiting weeks for a pull request to be approved

Cons

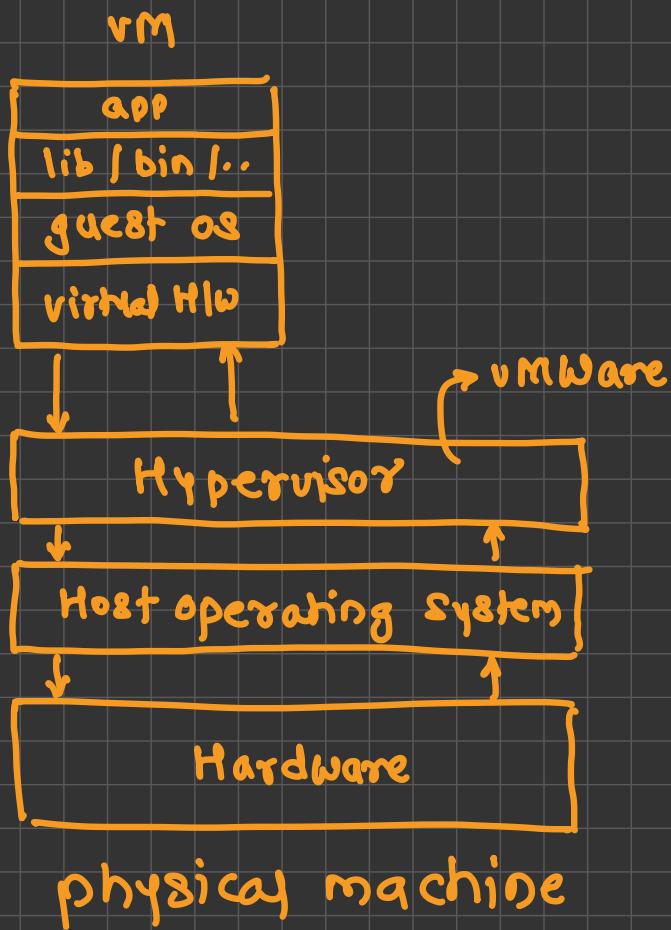
- **Development sprawl**
 - These add more complexity compared to a monolith architecture, as there are more services in more places created by multiple teams
 - If development sprawl isn't properly managed, it results in slower development speed and poor operational performance
- **Exponential infrastructure costs**
 - Each new microservice can have its own cost for test suite, deployment playbooks, hosting infrastructure, monitoring tools, and more
- **Added organizational overhead**
 - Teams need to add another level of communication and collaboration to coordinate updates and interfaces
- **Debugging challenges**
 - Each microservice has its own set of logs, which makes debugging more complicated
 - Plus, a single business process can run across multiple machines, further complicating debugging
- **Lack of standardization**
 - Without a common platform, there can be a proliferation of languages, logging standards, and monitoring
- **Lack of clear ownership**
 - As more services are introduced, so are the number of teams running those services
 - Over time it becomes difficult to know the available services a team can leverage and who to contact for support

Containerization

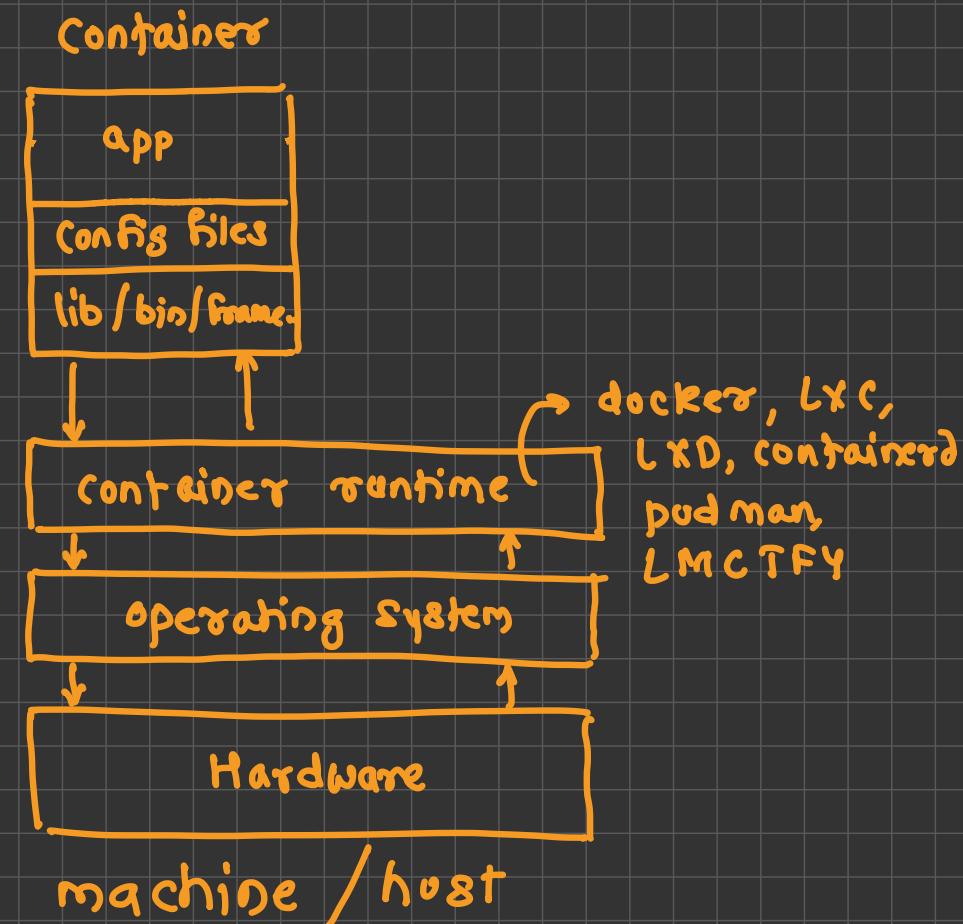
OS virtualization

OS gets shared with containers
lite weight VM

HW virtualization



OS virtualization



What is Containerization

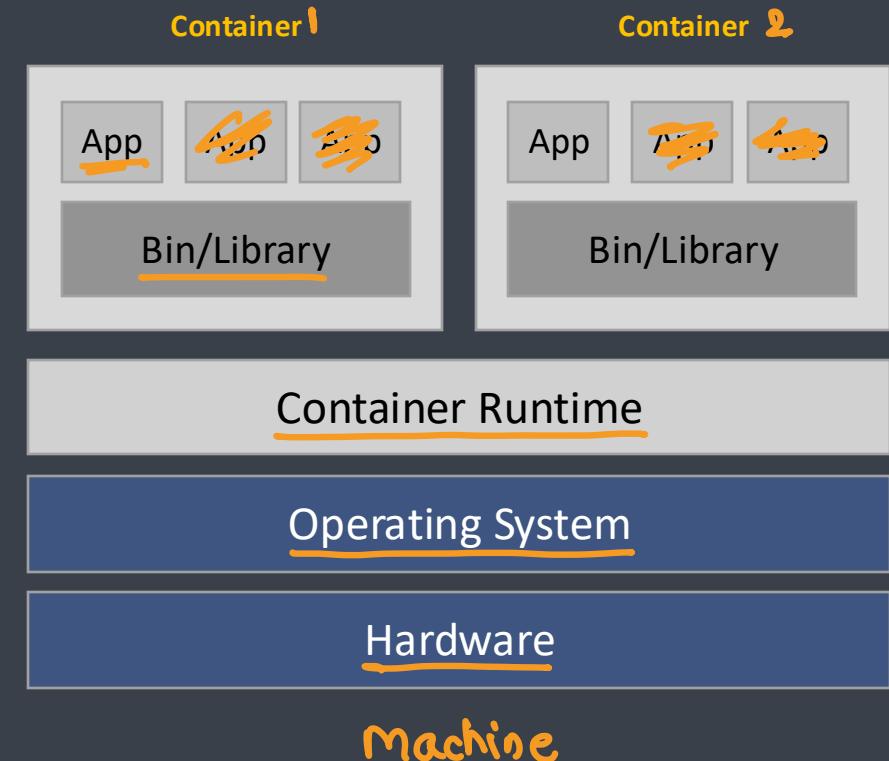
minimilistic OS

libraries
frameworks
config files
Resources

- Containerization is the packaging of software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable—called a container—that runs consistently on any infrastructure
- More portable and resource-efficient than virtual machines (VMs), containers have become the de facto compute units of modern cloud-native applications
- Containerization allows developers to create and deploy applications faster and more securely
- With traditional methods, code is developed in a specific computing environment which, when transferred to a new location, often results in bugs and errors
- For example, when a developer transfers code from a desktop computer to a VM or from a Linux to a Windows operating system
- Containerization eliminates this problem by bundling the application code together with the related configuration files, libraries, and dependencies required for it to run
- This single package of software or “container” is abstracted away from the host operating system, and hence, it stands alone and becomes portable—able to run across any platform or cloud, free of issues

Container deployment

- Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications
- Therefore, containers are considered lightweight
- Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more
- As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions



Containerization vs Virtualization (HW)

Virtual Machine	Container
<u>Hardware level virtualization</u> VMs share HW	<u>OS virtualization</u> containers share OS
<u>Heavyweight (bigger in size)</u> because of OS	<u>Lightweight (smaller in size)</u> does not contain OS
<u>Slow provisioning</u> (creation)	<u>Real-time and fast provisioning</u> No booting required, real H/W
<u>Limited Performance</u> slower than container	<u>Native performance</u> faster than VM
<u>Fully isolated</u>	<u>Process-level isolation</u>
<u>More secure</u>	<u>Less secure</u>
<u>Each VM has separate OS</u>	<u>Each container can share OS resources</u>
<u>Boots in minutes</u>	<u>Starts</u> <u>Boots</u> in seconds
<u>Pre-configured VMs are difficult to find and manage</u>	<u>Pre-built containers are readily available</u>
<u>Can be easily moved to new OS</u> VM is mutable	<u>Containers are destroyed and recreated</u>
<u>Creating VM takes longer time</u>	<u>Containers can be created in seconds</u>

Containers are
immutable

Benefits

- **Portability**

- A container creates an executable package of software that is abstracted away from (not tied to or dependent upon) the host operating system, and hence, is portable and able to run uniformly and consistently across any platform or cloud

- **Agility**

- The open source Docker Engine for running containers started the industry standard for containers with simple developer tools and a universal packaging approach that works on both Linux and Windows operating systems
- The container ecosystem has shifted to engines managed by the Open Container Initiative (OCI)
- Software developers can continue using agile or DevOps tools and processes for rapid application development and enhancement

- **Speed**

- Containers are often referred to as “lightweight,” meaning they share the machine’s operating system (OS) kernel and are not bogged down with this extra overhead
- Not only does this drive higher server efficiencies, it also reduces server and licensing costs while speeding up start-times as there is no operating system to boot

- **Fault isolation**

- Each containerized application is isolated and operates independently of others
- The failure of one container does not affect the continued operation of any other containers
- Development teams can identify and correct any technical issues within one container without any downtime in other containers
- Also, the container engine can leverage any OS security isolation techniques—such as SELinux access control—to isolate faults within containers

Benefits

- Efficiency

- Software running in containerized environments shares the machine's OS kernel, and application layers within a container can be shared across containers
- Thus, containers are inherently smaller in capacity than a VM and require less start-up time, allowing far more containers to run on the same compute capacity as a single VM. This drives higher server efficiencies, reducing server and licensing costs

- Ease of management

- Container orchestration platform automates the installation, scaling, and management of containerized workloads and services
- Container orchestration platforms can ease management tasks such as scaling containerized apps, rolling out new versions of apps, and providing monitoring, logging and debugging, among other functions. Kubernetes, perhaps the most popular container orchestration system available, is an open source technology (originally open-sourced by Google, based on their internal project called Borg) that automates Linux container functions originally
- Kubernetes works with many container engines, such as Docker, but it also works with any container system that conforms to the Open Container Initiative (OCI) standards for container image formats and runtimes

- Security

- The isolation of applications as containers inherently prevents the invasion of malicious code from affecting other containers or the host system
- Additionally, security permissions can be defined to automatically block unwanted components from entering containers or limit communications with unnecessary resources

Docker

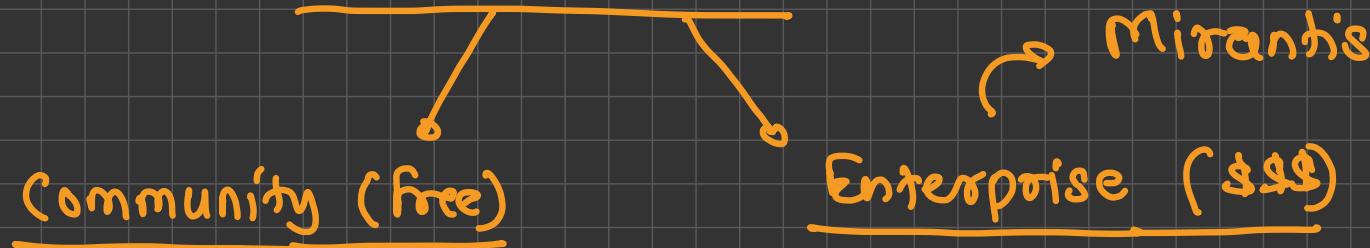
What is Docker? (Container Runtime / platform)

- Docker is an open source **platform** that enables developers to build, deploy, run, update and manage containers—standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment
 - Why docker?**
 - It is an easy way to create application **deployable packages**.
 - Developer can create ready-to-run containerized applications
 - It provides consistent computing environment
 - It works equally well in on-prem as well as cloud environments
 - It is light weight compared to VM
-
- The diagram illustrates the concept of Docker images. In the center, the word "images" is written in orange. Two arrows point from this central term to the right, each leading to a text label: "on premise" above and "on cloud" below. Both labels are written in orange.

Little history about Docker

- Docker Inc, started by Solomon Hykes, is behind the docker tool
- Docker Inc started as Paas provider called as dotCloud
- In 2013, the dotCloud became Docker Inc
- Docker Inc was using LinuX Containers (LXC) before version 0.9
- After 0.9 (2014), Docker replaced LXC with its own library libcontainer which is developed in Go programming language
- Its not the only solution for containerization
 - “FreeBSD Jails”, launched in 2000
 - LXD is next generation system container manager build on top of LXC and REST APIs
 - Google has its own open source container technology lmctfy (Let Me Contain That For You)
 - Rkt is another option for running containers
 - RedHat has its own tool podman

Docker Versions

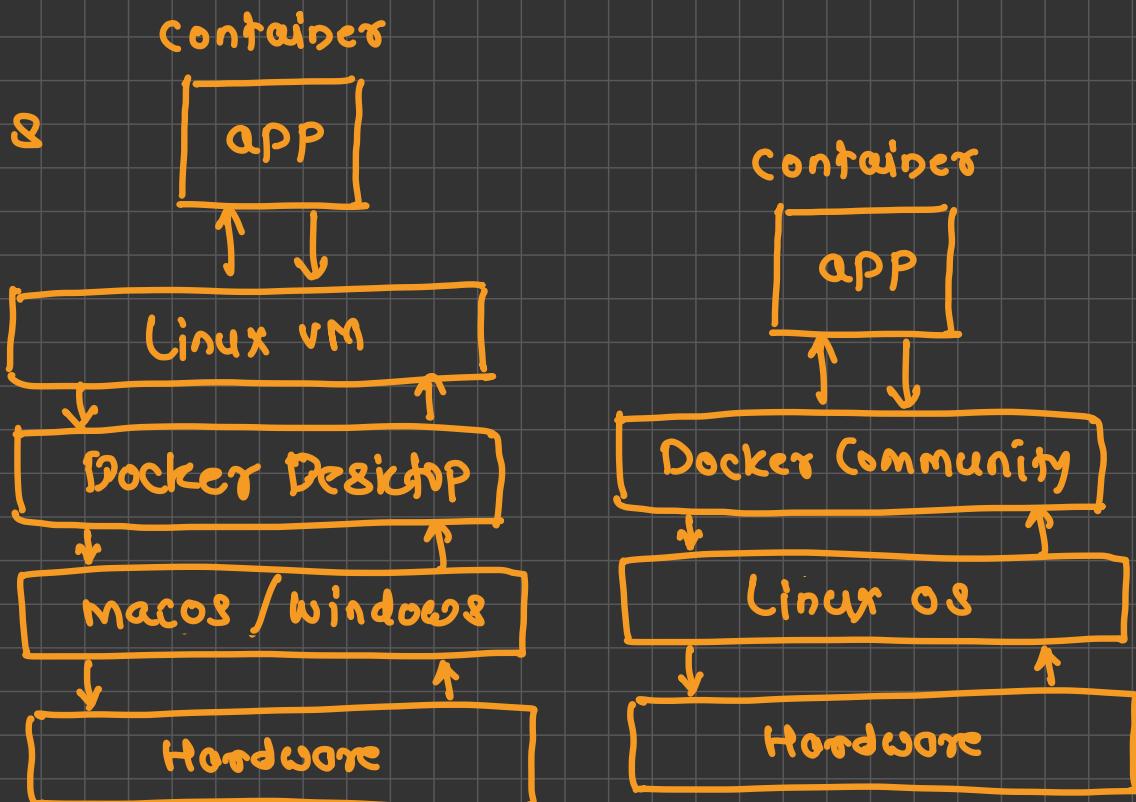


→ Linux community Edition

↳ Linux os has Linux kernel

→ Docker Desktop

↳ macos and windows



Docker Architecture → client-server architecture

→ server

Docker daemon (dockerd)

- Continuous running process
- Manages the containers

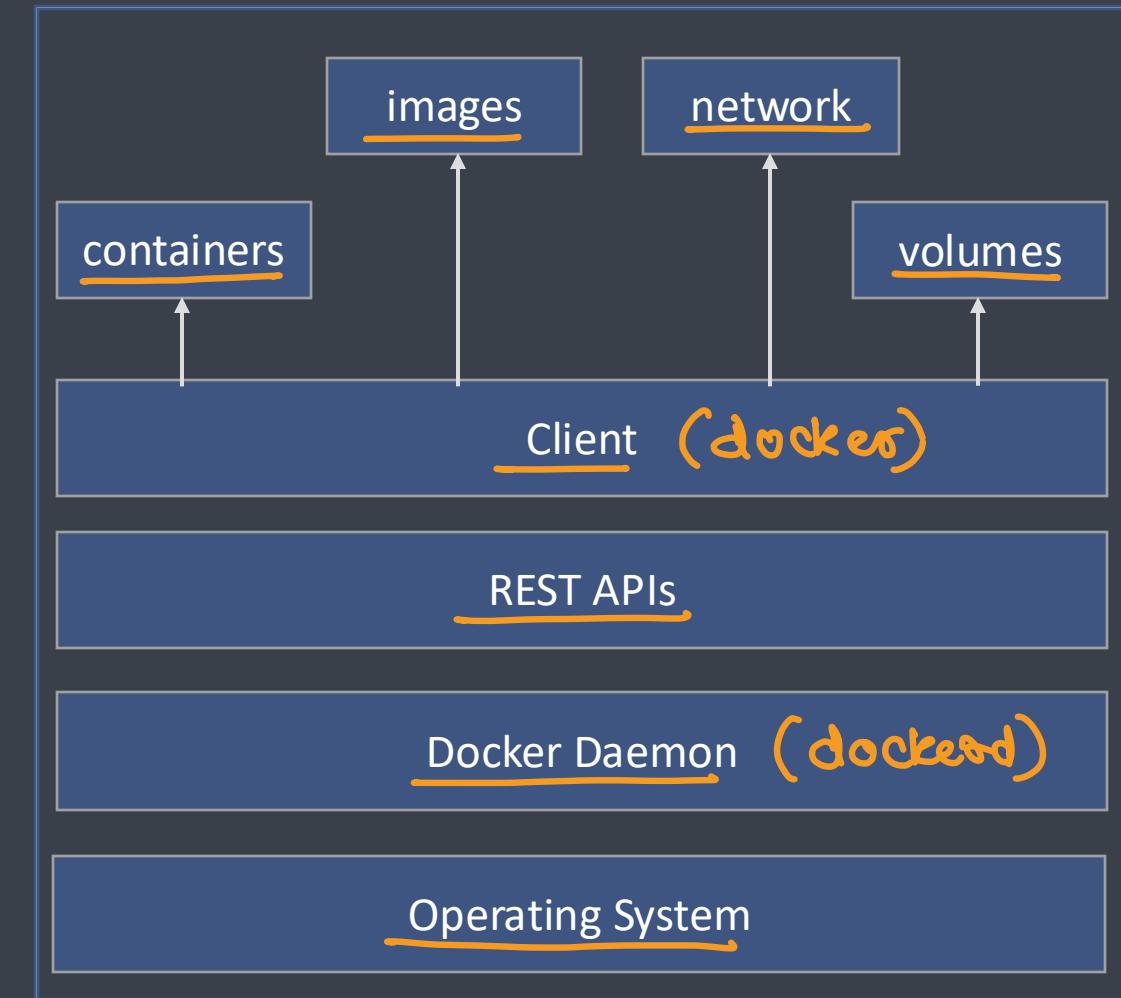
REST APIs

- Used to communicate with docker daemon

Client (docker)

- Provides command line interface
- Used to perform all the tasks

background processes
↳ no UI



libcontainer

- Docker has replaced LXC by libcontainer, which is used to manage the containers
- Libcontainer uses
 - Namespaces
 - Creates isolated workspace which limits what container can see
 - Provides a layer of isolation to the container
 - Each container runs in a separate namespace
 - Processes running in a namespace can interact with other processes or use resources which are the part of the same namespace
 - E.g. process ID, network, IPC, Filesystem
 - Control Groups (cgroups)
 - Used to share the available resources to the containers
 - It optionally enforces limits and constraints on resource usage
 - It limits how much a container can use
 - E.g. CPU, Disk space, memory

libcontainer

- Union File System (UnionFS)

- It uses layers
- It is a lightweight and very fast FS
- Docker uses different variants of UnionFS
 - Aufs (advanced multi-layered unification filesystem)
 - Btrfs (B-Tree FS)
 - VFS (Virtual FS)
 - Devicemapper

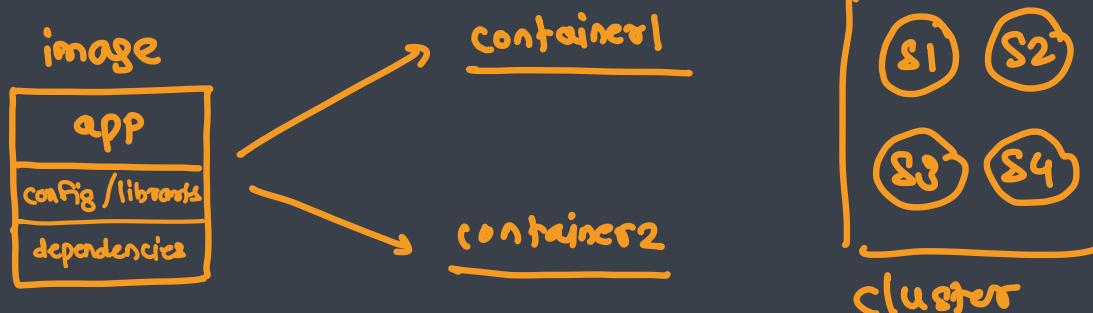
Docker Objects (modules)

- ✓ Images: read only template with instructions for creating docker containers
- ✓ Container: running instance of a docker image
- Network: network interface used to connect the containers to each other or external networks
- Volumes: used to persist the data generated by and used by the containers
- Registry: private or public collection of docker images
- Service: used to deploy application in a docker multi node cluster

What is Docker image ?

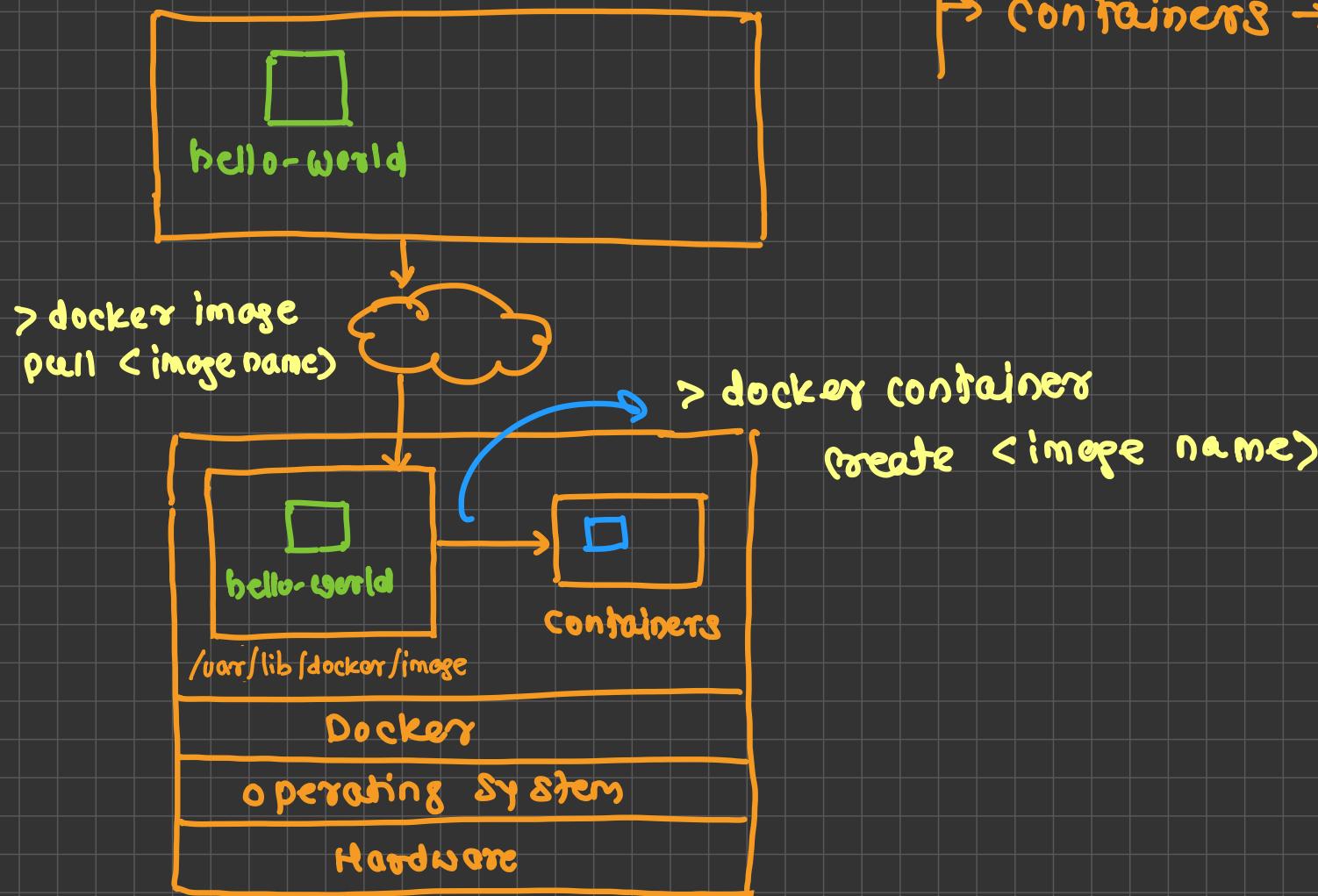
image (class) → container (object) → immutable

- A Docker image is a file used to execute code in a Docker container
- Docker images act as a set of instructions to build a Docker container, like a template
- Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual machine (VM) environments
- A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container
- Docker images have multiple layers, each one originates from the previous layer but is different from it
- The layers speed up Docker builds while increasing reusability and decreasing disk use
- Image layers are also read-only files
- Once a container is created, a writable layer is added on top of the unchangeable images, allowing a user to make changes



Docker Workflow

(hub.docker.com)
Docker image Registry



docker Home = /var/lib/docker

- image → contains images
- volumes → contains volumes
- networks → contains n/w objects
- containers → contains containers

→ app
→ libraries / frameworks
→ config files
→ dependencies

unit which contains

Docker Container

↳ created using image

Docker Container (process)

- It is a running **aspect of docker image**
- Contains one or more running processes
- It is a **self-contained** environment
- It wraps up an application into its own **isolated box** (application running inside a container has no knowledge of any other applications or processes that exist outside the container)
- A container can not modify the image from which it is created
- **It consists of**
 - Your application code
 - Dependencies
 - Networking
 - Volumes
- Containers are stored under /var/lib/docker /containers
- This directory contains images, containers, network volumes etc

→ **sandbox**

Basic Operations

- Creating container
- Starting container
- Running container
- Listing running containers
- Listing all containers
- Getting information of a container
- Stopping container
- Deleting container

Attaching a container

- There are two ways to attach to a container
- Attach
 - Used to attach the container
 - Uses only one input and output stream
 - Task
 - Attach to a running container
- Exec
 - Mainly it is used for running a command inside a container
 - Task
 - Execute a command inside container

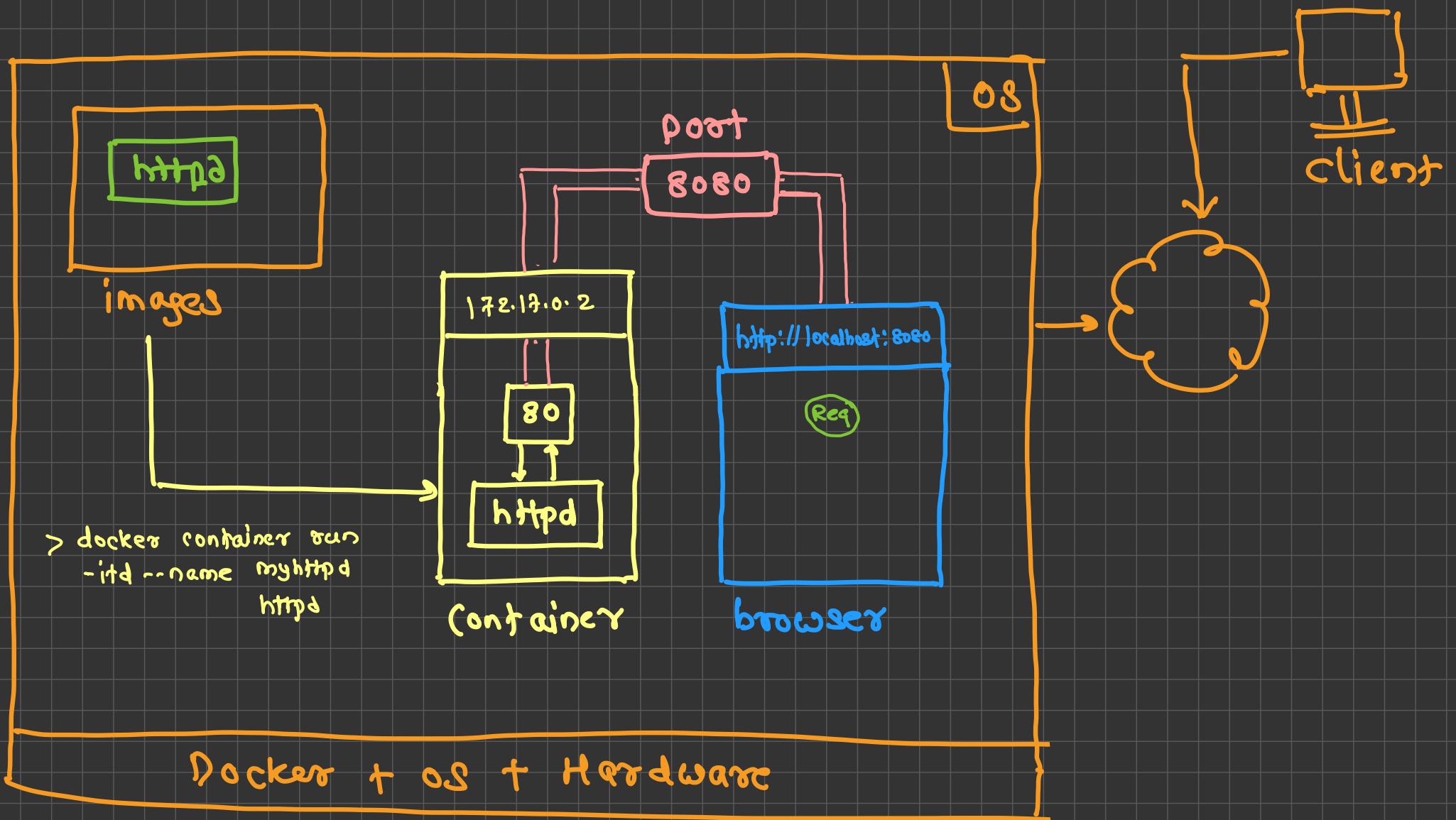
Hostname and name of container

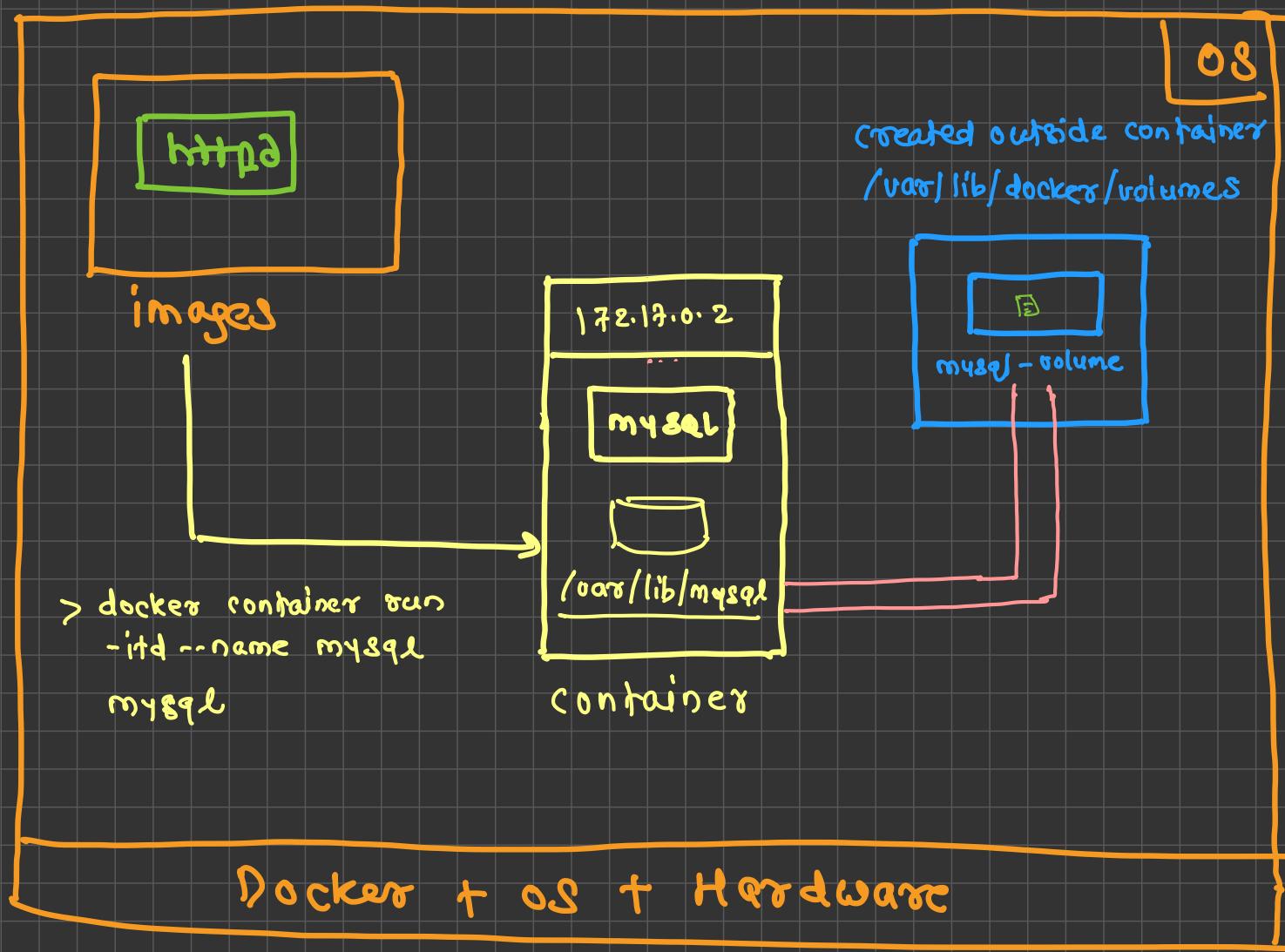
- To check the host name
 - Go inside the container
 - Check the hostname by using a command `hostname`
- Docker uses the first 12 characters of container id as hostname
- Docker automatically generates a name at random for each container

Publishing port on container

- Publishing a port is required to give an external access to your application *running inside container*
- Port can be published only at the time of creating a container
- You can not update the port configuration on running container
- Task
 - Run a httpd container with port 8080 published, to access apache externally

Port forwarding: forwarding request from one port to another





Docker Images (Advanced)

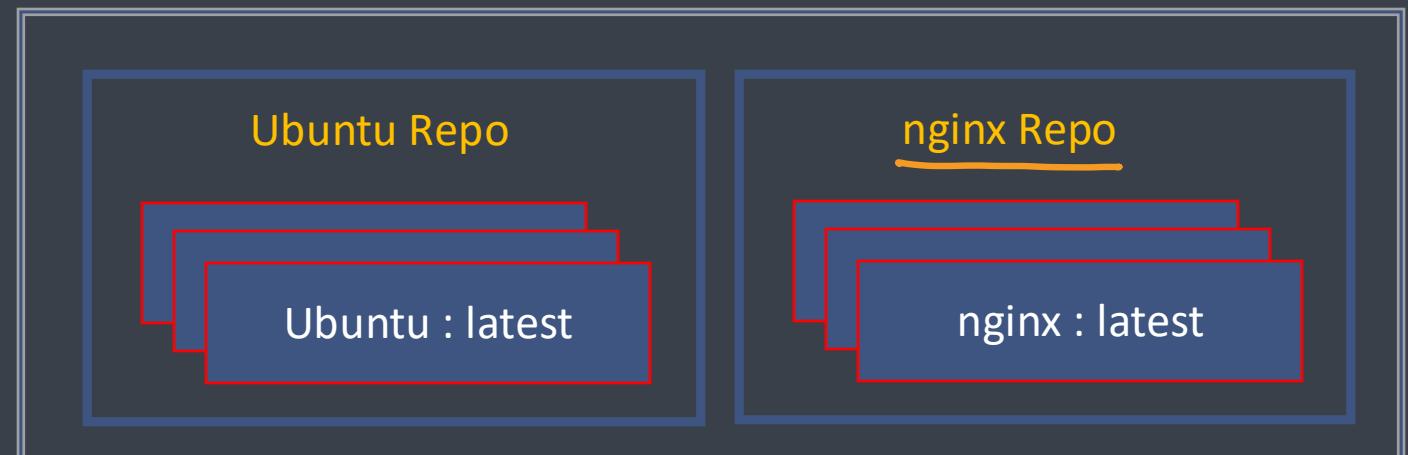
custom images

Docker Image → immutable

- Read-only instructions to run the containers
- It is made up of different layers
- Repositories hold images
- Docker registry stores repositories
- To create a custom image
 - Commit the running container
 - Use a Dockerfile ✓
- Task
 - Create a container
 - Create a directory and a file within it
 - Commit the container to create a new image

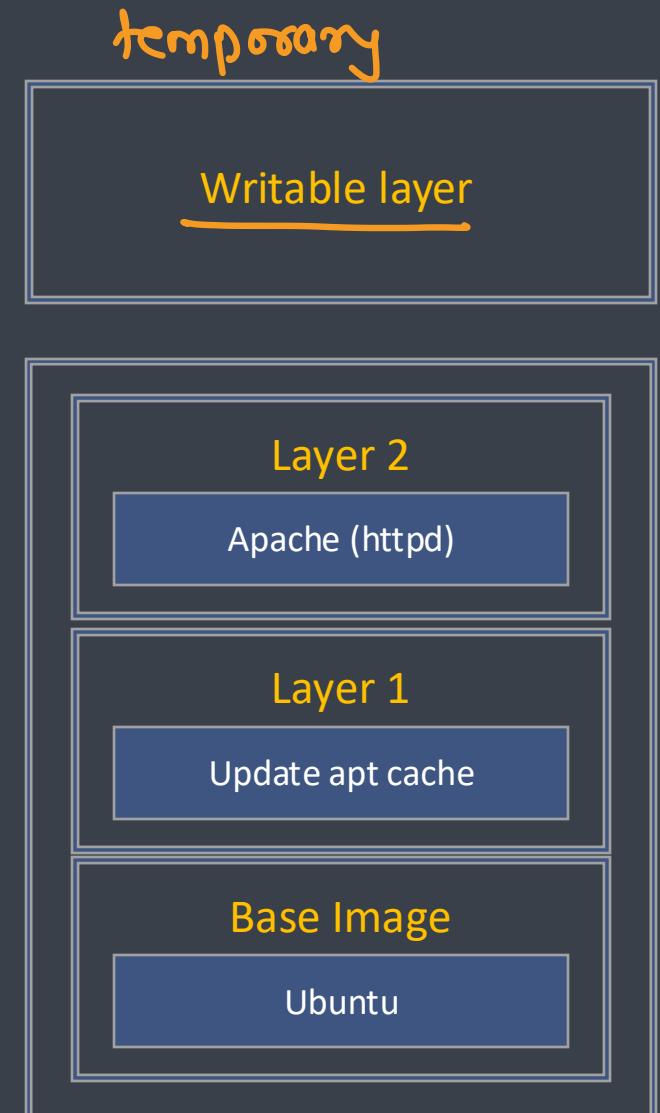
layer → instruction

→ public → hub.docker.com
→ private
Docker Registry Server



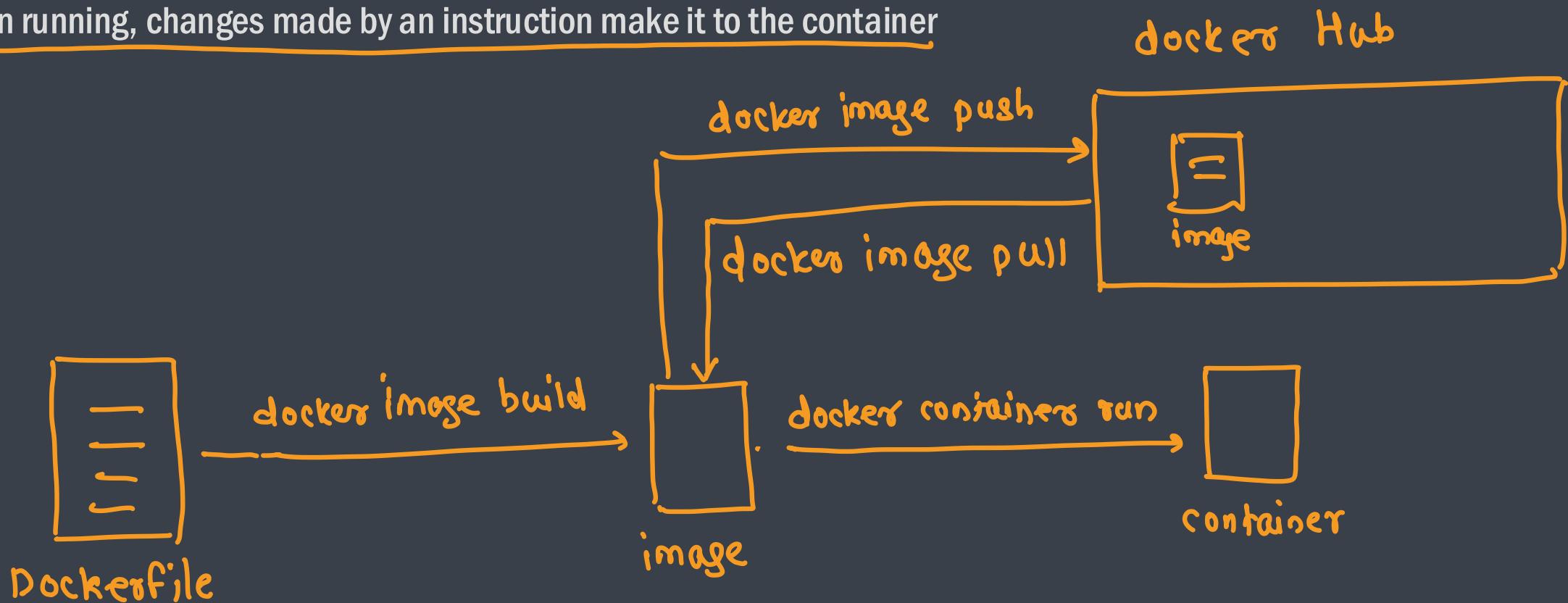
Layered File System

- Docker images are made of layered FS
- Docker uses UnionFS for implementing the layered docker images
- Any update on the image adds a new layer
- All changes made to the running container are written inside a writable layer



Dockerfile → collection of instruction - argument pairs

- The Dockerfile contains a series of instructions paired with arguments
- Each instruction should be in upper-case and be followed by an argument
- Instructions are processed from top to bottom
- Each instruction adds a new layer to the image and then commits the image
- Upon running, changes made by an instruction make it to the container



Dockerfile instructions

- FROM
- ENV
- RUN
- CMD
- EXPOSE
- WORKDIR
- ADD
- COPY
- LABEL
- MAINTAINER
- ENTRYPOINT

single point of failure



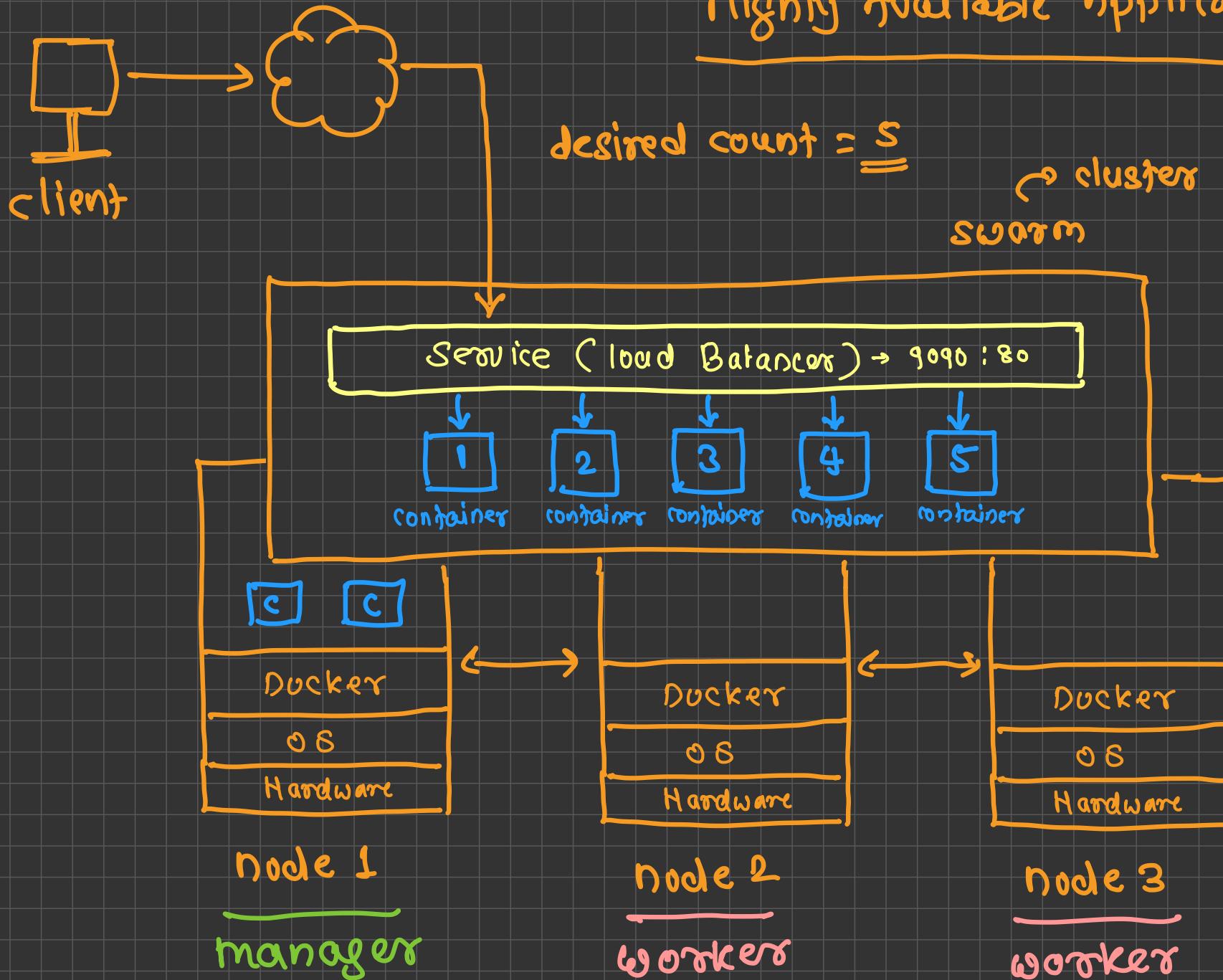
Orchestration

Container mgmt in a cluster

group of machines
nodes



Highly Available Application



Container Orchestration



load

- Container orchestration is all about managing the lifecycles of containers, especially in large, **dynamic** environments
- Software teams use container orchestration to control and automate many tasks
 - Provisioning and deployment of containers → **Service**
 - Redundancy and availability of containers
 - Scaling up or removing containers to spread application load evenly across host infrastructure
 - Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
 - Allocation of resources between containers
 - External exposure of services running in a container with the outside world
 - Load balancing of service discovery between containers
 - Health monitoring of containers and hosts
 - Configuration of an application in relation to the containers running it
- Orchestration Tools
 - ✓ Docker Swarm
 - Kubernetes
 - Mesos
 - Marathon

Docker Swarm (cluster)

- Docker Swarm is a container orchestration engine
- It takes multiple Docker Engines running on different hosts and lets you use them together
- The usage is simple: declare your applications as stacks of services, and let Docker handle the rest
- It is secure by default → SSL
- It is built using Swarmkit

What is a swarm?

- A swarm consists of multiple Docker hosts which run in **swarm mode**
- A given Docker host can be a manager, a worker, or perform both roles
- When you create a service, you define its **optimal state** → **desired count**
- Docker works to maintain that desired state
 - For instance, if a worker node becomes unavailable, Docker schedules that node's tasks on other nodes
- A **task** is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container
- When Docker is running in swarm mode, you can still run standalone containers on any of the Docker hosts participating in the swarm, as well as swarm services
- A key difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any daemon

Features

- Cluster management integrated with Docker Engine
- Decentralized design
- Declarative service model
- Scaling → up and down
- Desired state **reconciliation**
- Multi-host networking
- Service discovery
- Load balancing
- Secure by default
- Rolling updates

[No third party installation]

Docker Swarm

→ development / learning

→ single node cluster

↳ node = manager + worker

→ multi node cluster

→ Not HA cluster

→ single manager cluster

→ development / testing

↳ 1 manager + multiple workers

→ multi manager cluster

→ HA cluster
→ production

↳ multiple managers + multiple workers

↳ Election process to choose the leader

↳ Raft consensus algorithm

Nodes (machines)

- A **node** is an instance of the Docker engine participating in the **swarm**
- You can run one or more nodes on a single physical computer or cloud server
- To deploy your application to a swarm, you submit a **service definition** to a **manager node**
- **Manager Node**
 - The manager node dispatches units of work called **tasks** to worker nodes
 - Manager nodes also perform the **orchestration** and cluster management functions required to maintain the desired state of the **swarm**
 - Manager nodes elect a single leader to conduct orchestration tasks
- **Worker nodes**
 - Worker nodes receive and execute tasks dispatched from manager nodes
 - An agent runs on each worker node and reports on the tasks assigned to it
 - The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker
 - ↳ **heart beat signal**

Services and tasks

- **Service**

- A service is the definition of the tasks to execute on the manager or worker nodes
- It is the central structure of the swarm system and the primary root of user interaction with the swarm
- When you create a service, you specify which container image to use and which commands to execute inside running containers

- **Task**

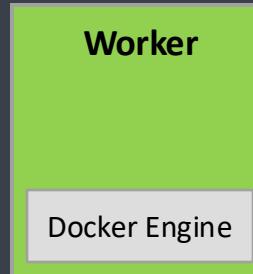
- A task carries a Docker container and the commands to run inside the container
- It is the atomic scheduling unit of swarm
- Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale
- Once a task is assigned to a node, it cannot move to another node
- It can only run on the assigned node or fail

Create a Swarm and add workers to swarm

- In our swarm we are going to use 3 nodes (one manager and two workers)
- Every node must have Docker Engine 1.12 or newer installed
- Following ports are used in node communication
 - TCP port 2377 is used for cluster management communication
 - TCP and UDP port 7946 is used for node communication
 - UDP port 4789 is used for overlay network traffic



```
docker swarm init --advertise-addr <ip>
```



```
docker swarm join --token <token>
```

Swarm Setup

- **Create swarm**

```
> docker swarm init --advertise-addr <MANAGER-IP>
```

- **Get current status of swarm**

```
> docker info
```

- **Get the list of nodes**

```
> docker node ls
```

Swarm Setup

- **Get token (on manager node)**

> `docker swarm join-token worker`

- **Add node (on worker node)**

> `docker swarm join --token <token>`

Overlay Network

- It is a computer network built on top of another network
- Sits on top of the host-specific networks and allows container, connected to it, to communicate securely
- When you initialize a swarm or join a host to swarm, two networks are created
 - An overlay network called as ingress network
 - A bridge network called as docker_gwbridge
- Ingress network facilitates load balancing among services nodes
- Docker_gwbridge is a bridge network that connect overlay networks to individual docker daemon's physical network

Service

- Definition of tasks to execute on Manager or Worker nodes
- Declarative Model for Services
- Scaling
- Desired state reconciliation
- Service discovery
- Rolling updates
- Load balancing
- Internal DNS component

Swarm Service

- **Deploy a service**

> `docker service create --replicas <no> --name <name> -p <ports> <image> <command>`

- **Get running services**

> `docker service ls`

- **Inspect service**

> `docker service inspect <service>`

- **Get the nodes running service**

> `docker service ps <service>`

Swarm Service

- **Scale service**

> `docker service scale <service>=<scale>`

- **Update service**

> `docker service update --image <image> <service>`

- **Delete service**

> `docker service rm <service>`

docker swarm

- execute a command every after 1 second

```
| watch -n 1 watch -n 1 docker container ls
```

swarm

```
# check if the mode is running in swarm mode
# run the following command and check status of Swarm (active or inactive)
> docker system info

# check if the mode is running in swarm mode
> docker system info | grep Swarm

# initialize the swarm (cluster)
> docker swarm init

# stop the cluster
# note: run this command on the leader manager
> docker swarm leave --force

# generate a token for worker
> docker swarm join-token worker

# generate a token for manager
> docker swarm join-token manager
```

nodes

```
# get the list of nodes
> docker node ls

# get details of selected node
> docker node inspect <node id>

# remove a selected node
> docker node rm <node id>

# promote a worker
> docker node promote <worker id>

# demote a manager
```

```
> docker node demote <manager id>
```

service

```
# get the list of services
> docker service ls

# create a service
# params:
# - --name: service name
# - -p: used to publish the port
# - --replicas: used to specify the desired count
# > docker service create --name <service name> -p <source port>:
# <container port> --replicas <desired count> <image name>
> docker service create --name httpd -p 9090:80 --replicas 5 httpd

# get the service details
# > docker service inspect <service name>
> docker service inspect httpd

# get the containers created by selected service
# > docker service ps <service name>
> docker service ps httpd

# remove the service
# > docker service rm <service name>
> docker service rm httpd

# scale the service
# note: increase or decrease number of containers
# > docker service scale <service name>=<new desired count>
> docker service scale httpd=5
```



K&S

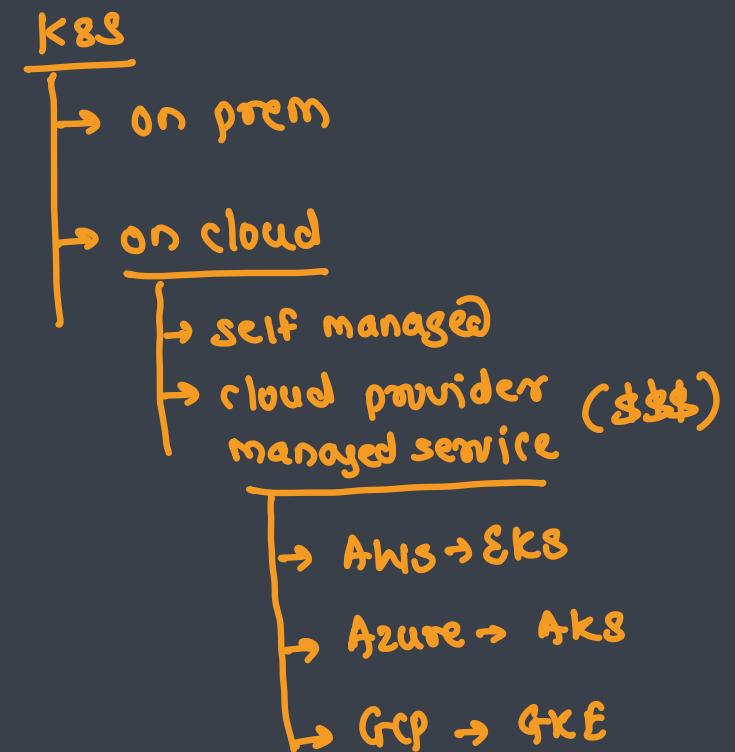
Kubernetes

Orchestration tool



What is Kubernetes ?

- Portable, extensible, open-source platform for managing containerized workloads and services
- Facilitates both declarative configuration and automation YAML
- It has a large, rapidly growing ecosystem
- Kubernetes services, support, and tools are widely available
- The name Kubernetes originates from Greek, meaning helmsman or pilot
- Google open-sourced the Kubernetes project in 2014





Traditional Deployment → deprecated

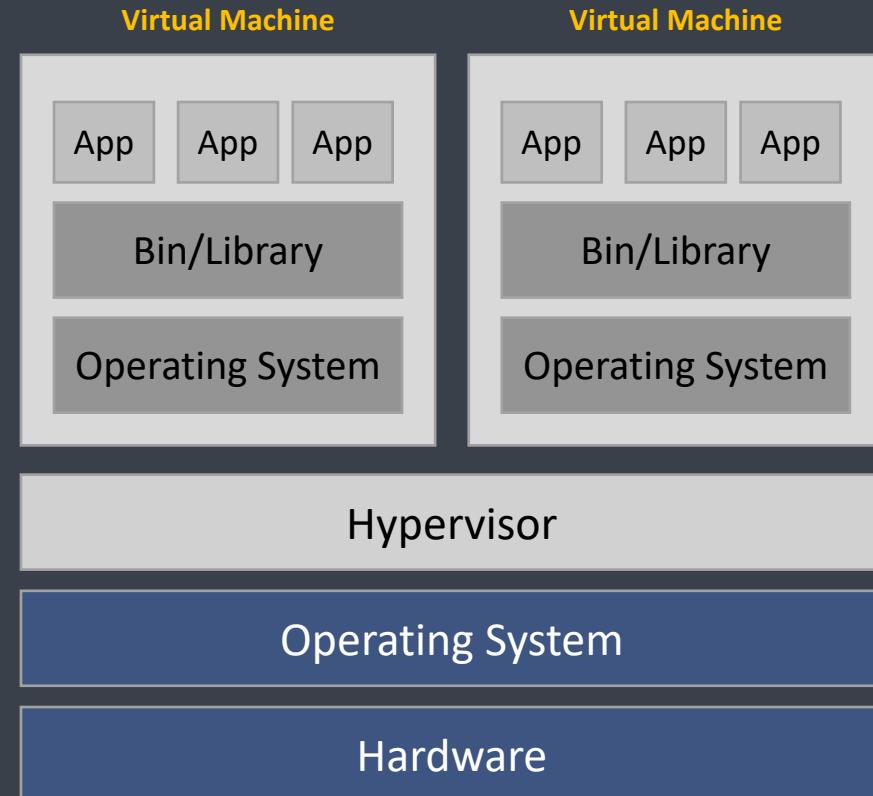
- Early on, organizations ran applications on physical servers
- There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues
- For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform
- A solution for this would be to run each application on a different physical server
- But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers





Virtualized Deployment

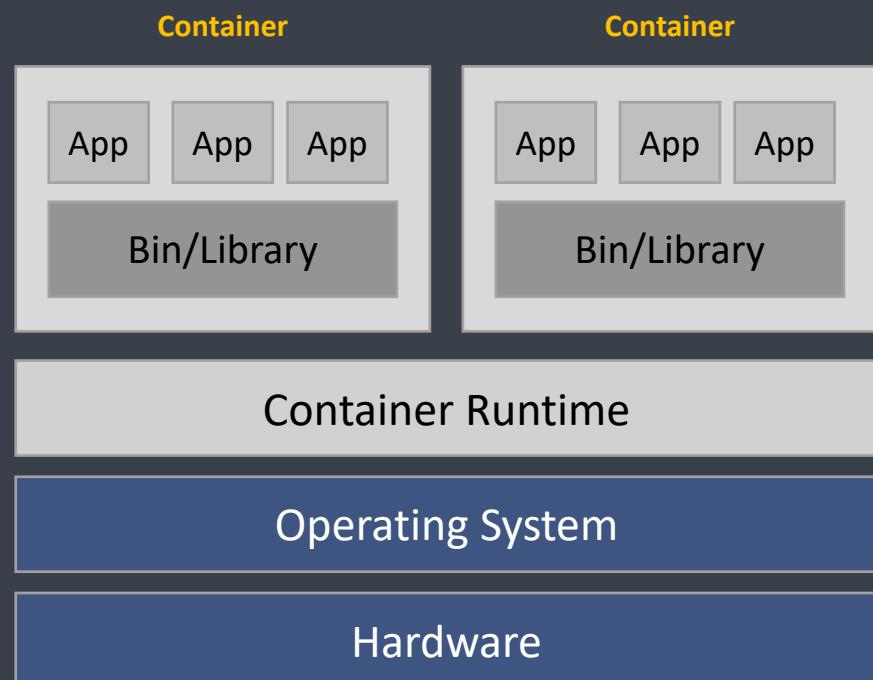
- It allows you to run multiple Virtual Machines (VMs) on a single physical server's CPU
- Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application
- Virtualization allows better utilization of resources in a physical server and allows better scalability because
 - an application can be added or updated easily
 - reduces hardware costs
- With virtualization you can present a set of physical resources as a cluster of disposable virtual machines
- Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware





Container deployment

- Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications
- Therefore, containers are considered lightweight
- Similar to a VM, a container has its own filesystem, CPU, memory, process space, and more
- As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions



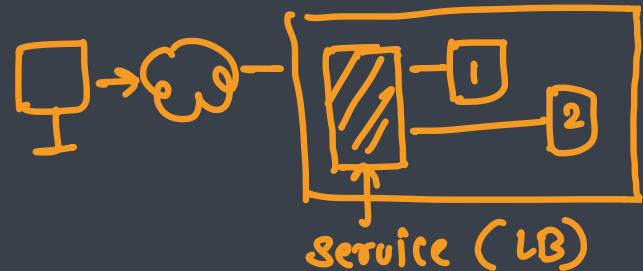


Container benefits

- Increased ease and efficiency of container image creation compared to VM image use
- Continuous development, integration, and deployment
- Dev and Ops separation of concerns
- Observability not only surfaces OS-level information and metrics, but also application health and other signals
- Cloud and OS distribution portability
- Application-centric management:
- Loosely coupled, distributed, elastic, liberated micro-services
- Resource isolation: predictable application performance



What Kubernetes provide?



■ Service discovery and load balancing

- Kubernetes can expose a container using the DNS name or using their own IP address
- If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable

■ Storage orchestration

- Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more

■ Automated rollouts and rollbacks

- You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate

■ Automatic bin packing → better node selection

- You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks
- You tell Kubernetes how much CPU and memory (RAM) each container needs
- Kubernetes can fit containers onto your nodes to make the best use of your resources



What Kubernetes provide?

- **Self-healing**

- Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve

- **Secret and configuration management**

- Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys
 - You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration



What Kubernetes is not

- Does not limit the types of applications supported
- Does not deploy source code and does not build your application
- Does not provide application-level services as built-in services
- Does not dictate logging, monitoring, or alerting solutions → prometheus / sentry / dcw & elasticsearch
- Does not provide nor mandate a configuration language/system
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems

→ runs any type of app

websites

backend

jobs / cron jobs

build → ant / maven / gradle

prometheus / sentry / dcw & elasticsearch

↳ K&S supports Linux [does not support windows and macos]

debain

- ubuntu
- knoppix

Red Hat

- RHEL
- fedora
- centos
- SUSE

K8S cluster

→ single node cluster (minikube)

 → virtual / simulated / not a real cluster

 → used to learn K8S

→ multi node cluster

 → single master cluster

 → only one master and one or more workers

 → used in development and testing env.

 → multi master cluster [HA cluster]

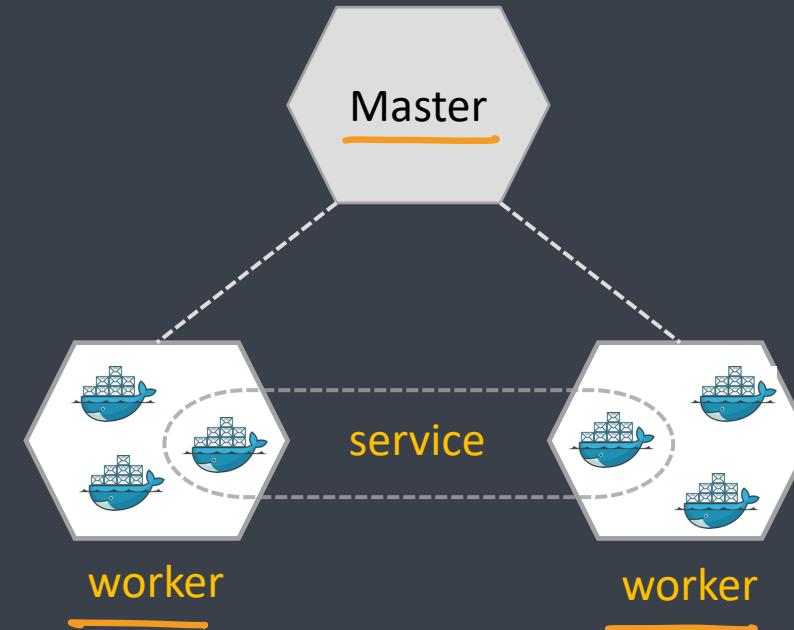
 → multiple masters and multiple workers

 → used in production env.



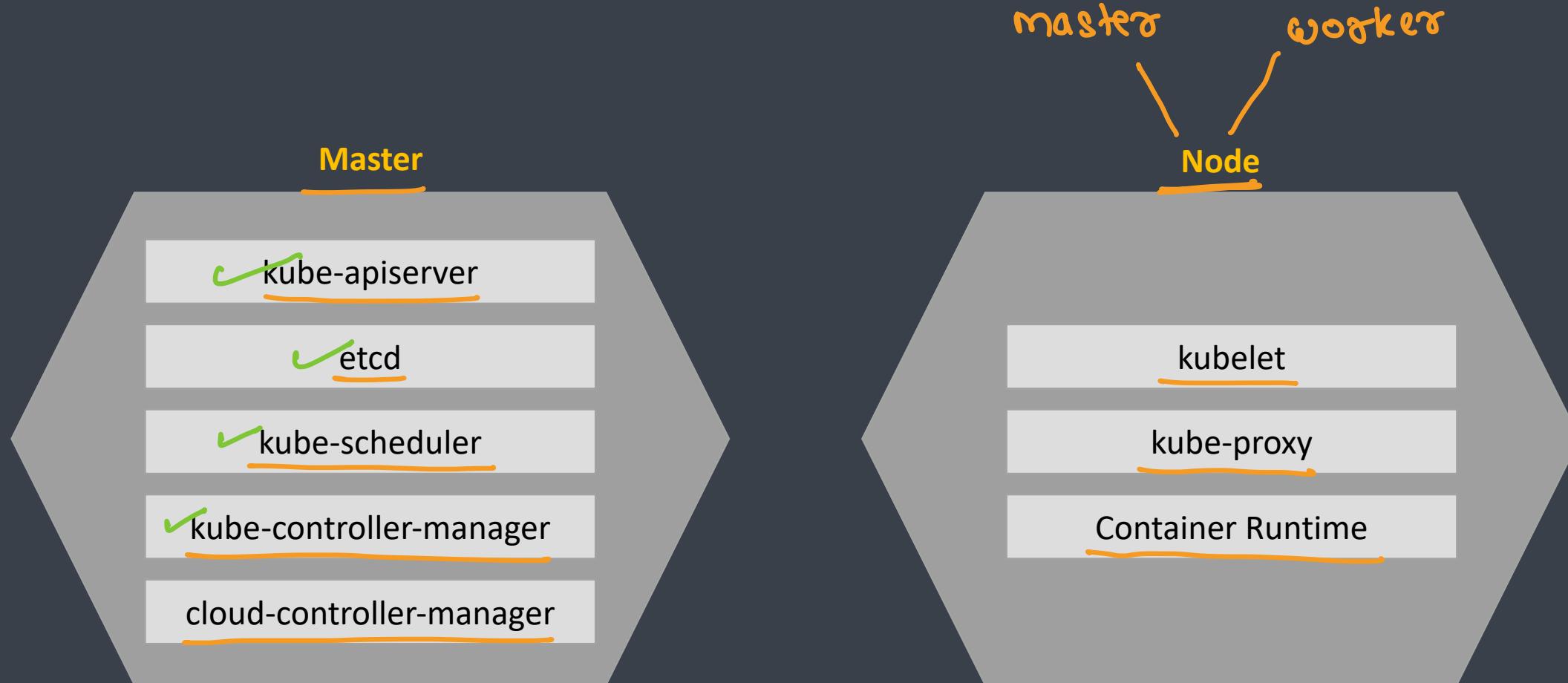
Kubernetes Cluster

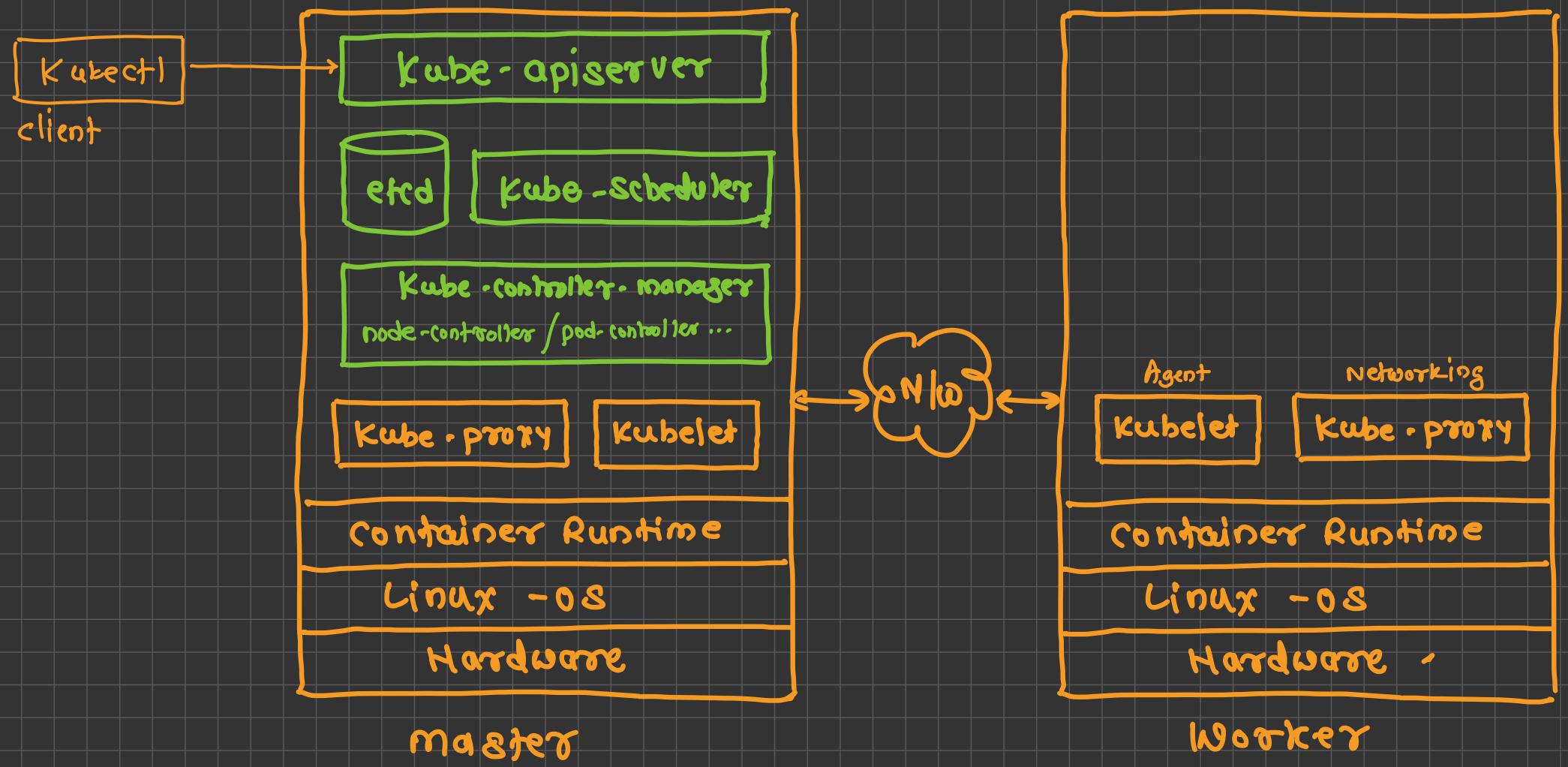
- When you deploy Kubernetes, you get a cluster.
- A cluster is a set of machines (nodes), that run containerized applications managed by Kubernetes
- A cluster has at least one worker node and at least one master node
- The worker node(s) host the pods that are the components of the application
- The master node(s) manages the worker nodes and the pods in the cluster
- Multiple master nodes are used to provide a cluster with failover and high availability





Kubernetes Components







Master Components

cluster

node Failure
pod stop ...

- Master components make global decisions about the cluster and they detect and respond to cluster events
- Master components can ~~be~~ run on any machine in the cluster
- kube-apiserver
 - The API server is a component that exposes the Kubernetes API (REST)
 - The API server is the front end for the Kubernetes
- etcd (database)
 - Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data
- kube-scheduler
 - Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on



Master Components

■ kube-controller-manager

- Component on the master that runs controllers
- Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process
- Types
 - Node Controller: Responsible for noticing and responding when nodes go down.
 - Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system → deprecated → replaced by ReplicaSet
 - Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods)
 - Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces

■ cloud-controller-manager

- Runs controllers that interact with the underlying cloud providers
- The cloud-controller-manager binary is an alpha feature introduced in Kubernetes release 1.6



Node Components

- Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment
- kubelet
 - An agent that runs on each node in the cluster
 - It makes sure that containers are running in a pod
- kube-proxy
 - Network proxy that runs on each node in your cluster, implementing part of the Kubernetes service concept
 - kube-proxy maintains network rules on nodes
 - These network rules allow network communication to your Pods from network sessions inside or outside of your cluster
- Container Runtime
 - The container runtime is the software that is responsible for running containers
 - Kubernetes supports several container runtimes: Docker, containerd, rktlet, cri-o etc.

X ✓



Create Cluster

- Use following commands on both master and worker nodes

```
> sudo apt-get update && sudo apt-get install -y apt-transport-https curl  
> curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -  
> cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list deb https://apt.kubernetes.io/kubernetes-xenial main EOF  
> sudo apt-get update  
> sudo apt-get install -y kubelet kubeadm kubectl  
> sudo apt-mark hold kubelet kubeadm kubectl
```



Initialize Cluster Master Node

- Execute following commands on master node

```
> kubeadm init --apiserver-advertise-address=<ip-address> --pod-network-cidr=10.244.0.0/16  
> mkdir -p $HOME/.kube  
> sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config  
> sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

- Install pod network add-on

```
> kubectl apply -f  
https://raw.githubusercontent.com/coreos/flannel/2140ac876ef134e0ed5af15c65e414cf26827915/Documentation/kube-flannel.yml
```



Add worker nodes

- Execute following command on every worker node

```
> kubeadm join --token <token> <control-plane-host>:<control-plane-port> --discovery-token-ca-cert-hash sha256:<hash>
```



Steps to install Kubernetes

5

join

join

4

POD Network

3

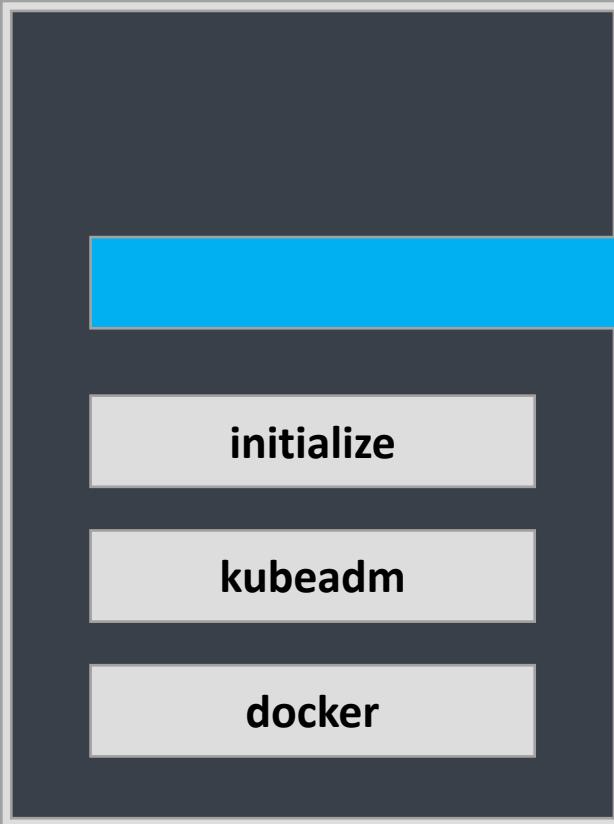
initialize

2

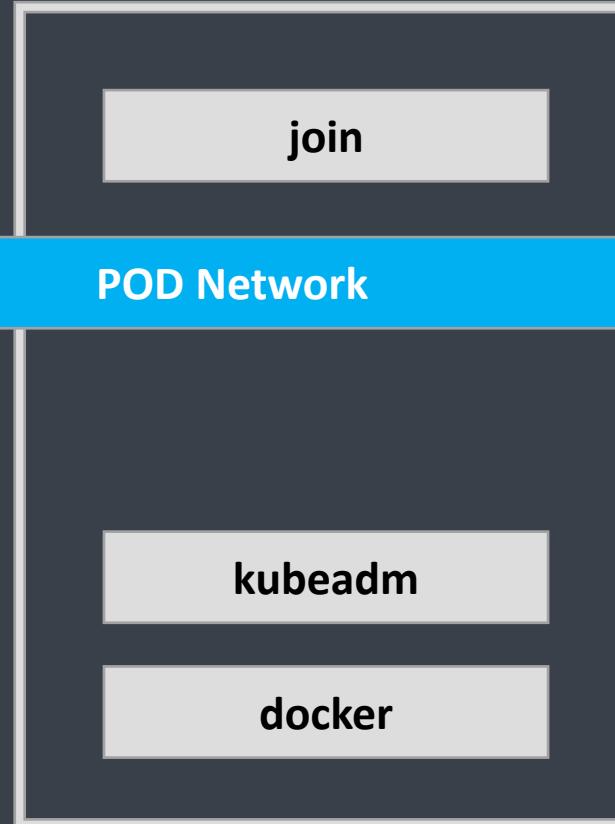
kubeadm

1

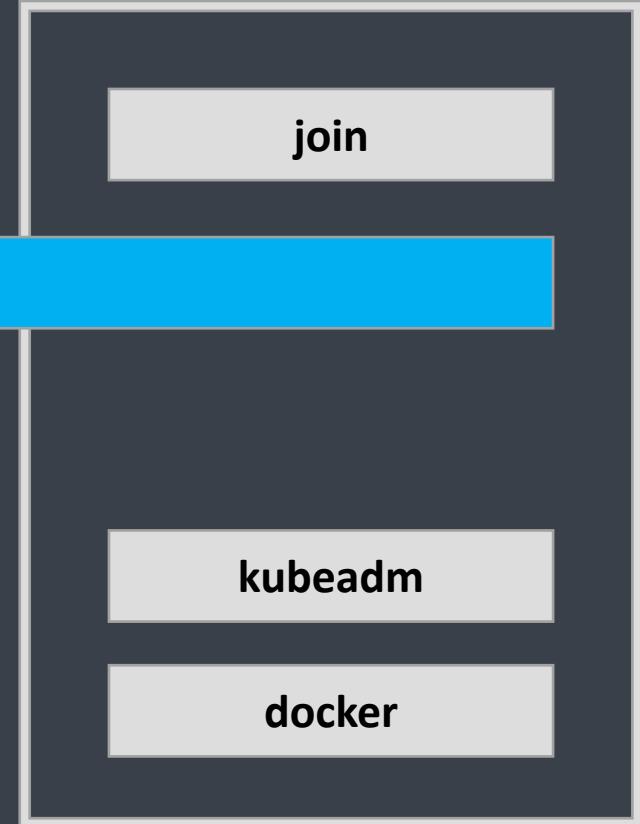
docker



Master



Worker Node 1



Worker Node 2



Kubernetes Objects

- The basic Kubernetes objects include

- Pod
- Service
- Volume
- Namespace

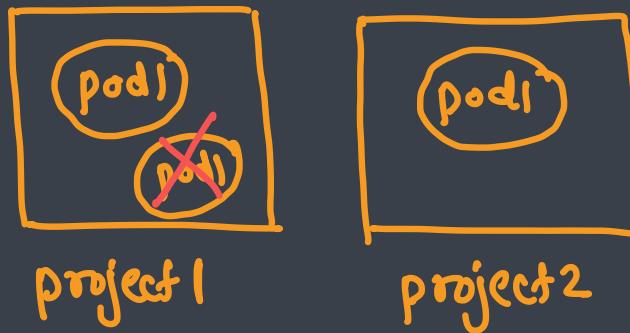
- Kubernetes also contains higher-level abstractions build upon the basic objects

- Deployment
- DaemonSet
- StatefulSet
- ReplicaSet
- Job



Namespace (group of pods/services/...)

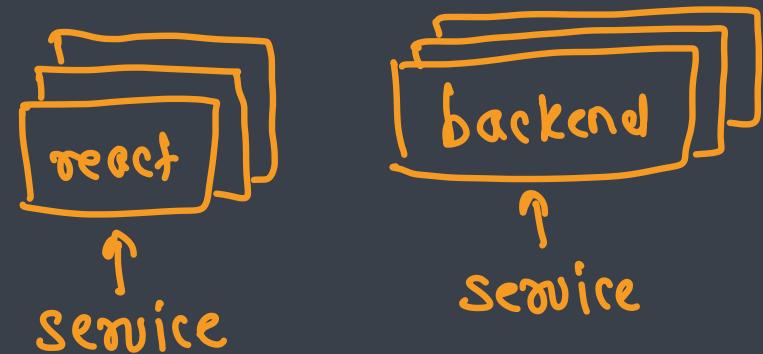
- Namespaces are intended for use in environments with many users spread across multiple teams, or projects
- Namespaces provide a scope for names
- Names of resources need to be unique within a namespace, but not across namespaces
- Namespaces can not be nested inside one another and each Kubernetes resource can only be in one namespace
- Namespaces are a way to divide cluster resources between multiple users (*environments / projects*)

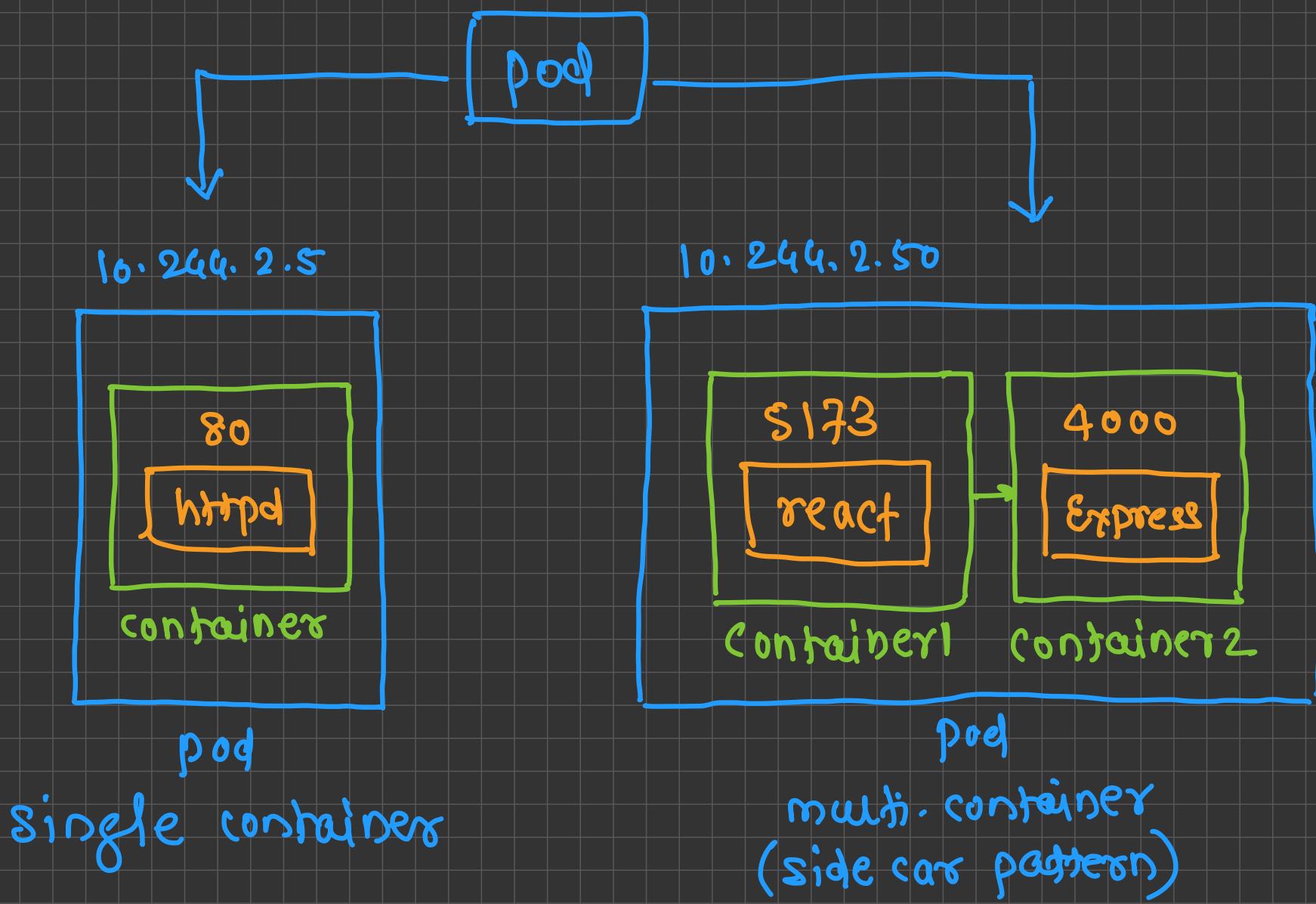




Pod

- A Pod is the basic execution unit of a Kubernetes application
- The smallest and simplest unit in the Kubernetes object model that you create or deploy
- A Pod represents processes running on your Cluster
- Pod represents a unit of deployment
- A Pod encapsulates
 - application's container (or, in some cases, multiple containers)
 - storage resources
 - a unique network IP
 - options that govern how the container(s) should run







YAML to create Pod

apiVersion: v1

kind: Pod

metadata:

name: myapp-pod

labels:

app: myapp

spec:

containers:

- name: myapp-container

image: httpd

- ① apiVersion → version of API in which object is defined
 - ↳ v1 → basic objects → pod / service / namespace
 - ↳ apps/v1 → complex objects → deployment / Replicaset
- ② kind → kind of object to be created
- ③ metadata → basic details of object to be created
 - ↳ name → name of the object (*)
 - ↳ labels → extra info attached to the pod
- ④ spec → specification / configuration of the object to be created



Service

→ Load Balancing

→ (NodePort) expose app outside the cluster

- An abstract way to expose an application running on a set of Pods as a network service
- Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service)
- Service Types

▪ ClusterIP → can not be accessed outside the cluster

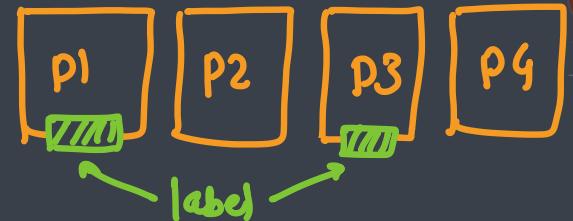
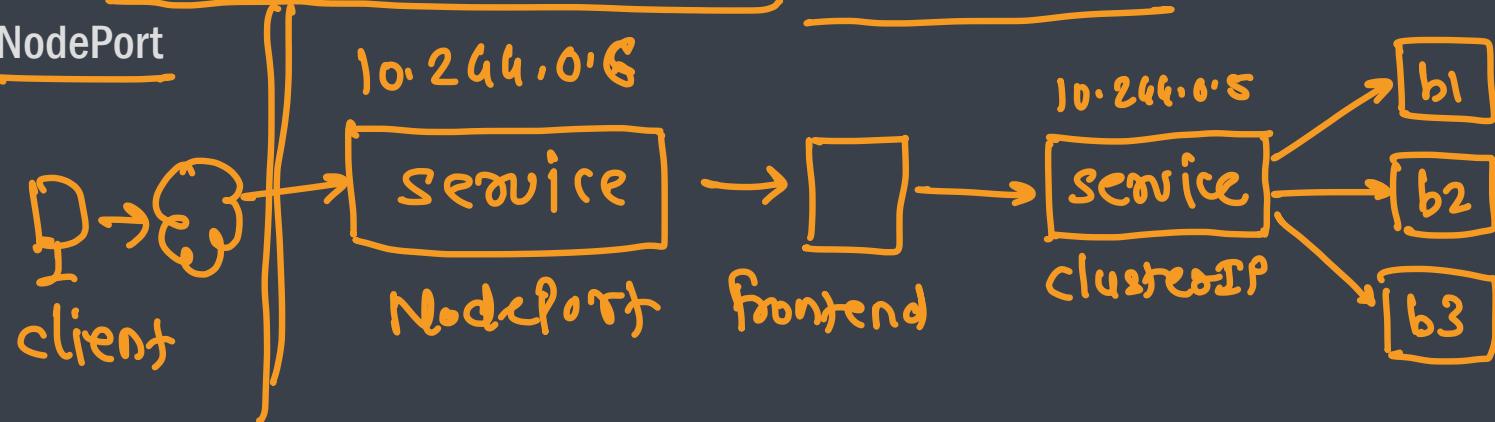
▪ Exposes the Service on a cluster-internal IP

▪ Choosing this value makes the Service only reachable from within the cluster

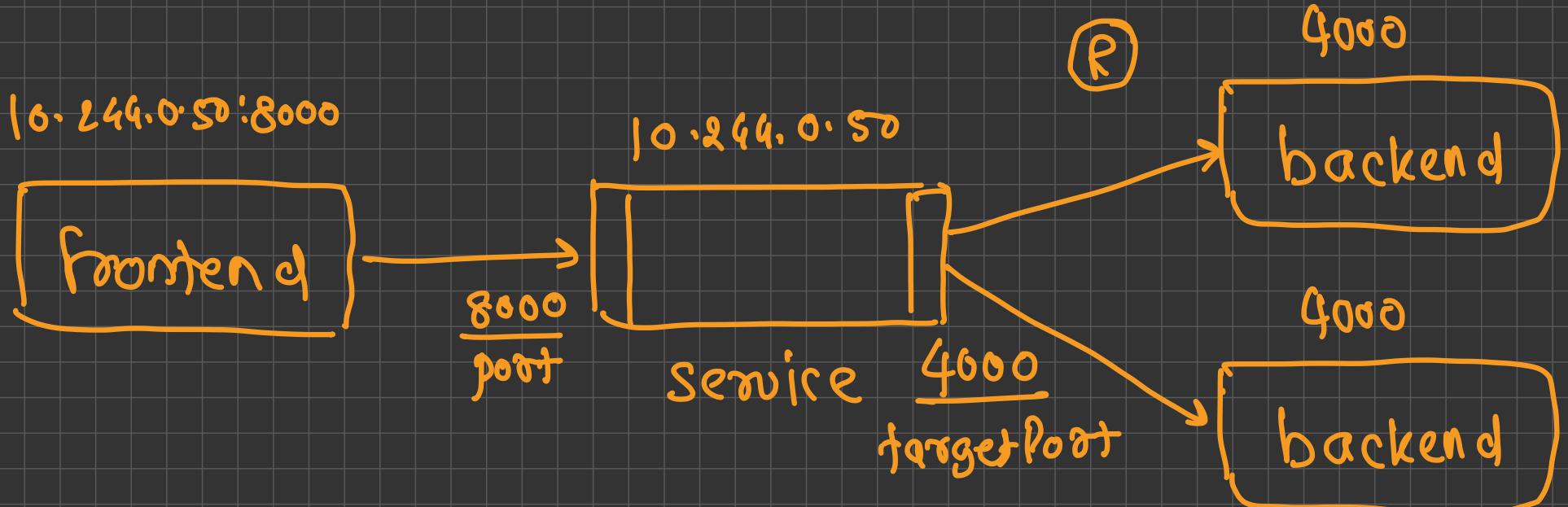
▪ LoadBalancer

▪ Used for load balancing the containers on the cloud

▪ NodePort



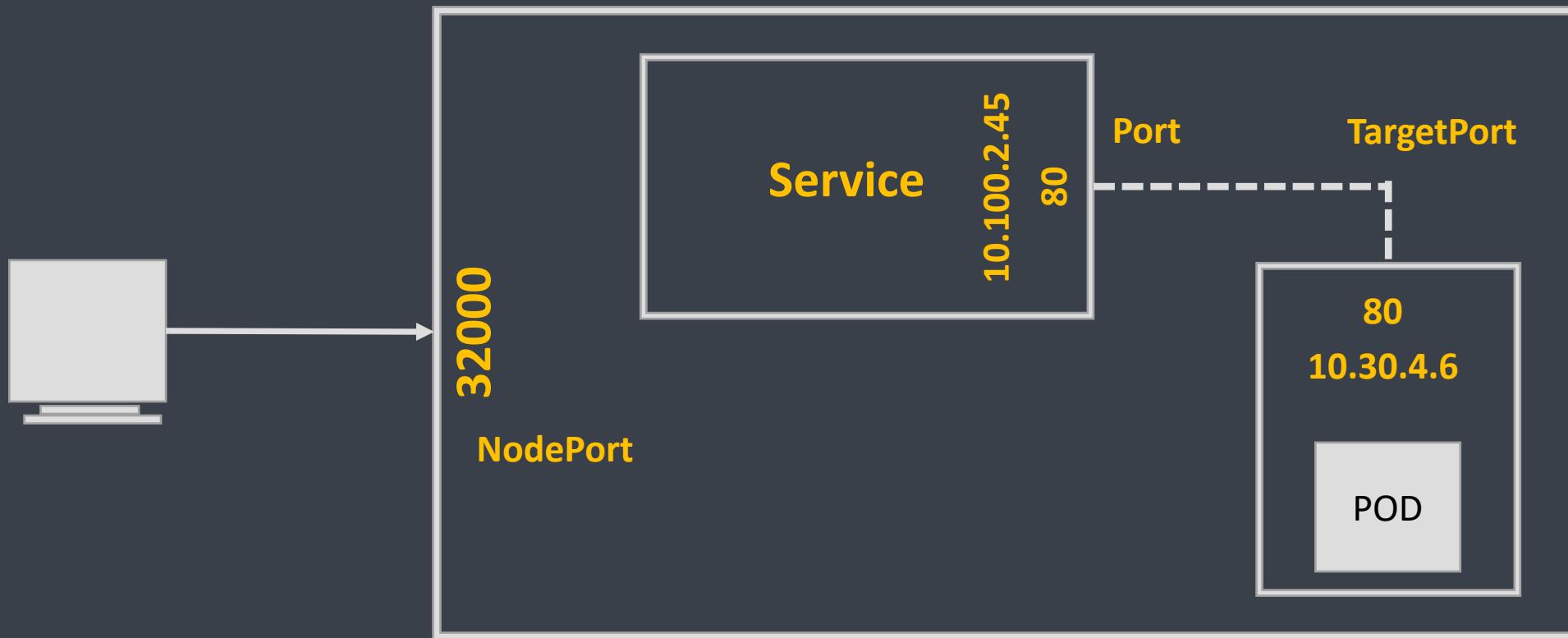
```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector: → label(s) to
            app: MyApp be used
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```





Service Type: NodePort

- Exposes the Service on each Node's IP at a static port (the NodePort)
- You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>





Replica Set

- A Replica Set ensures that a specified number of pod replicas are running at any one time
- In other words, a Replica Set makes sure that a pod or a homogeneous set of pods is always up and available
- If there are too many pods, the Replica Set terminates the extra pods
- If there are too few, the Replica Set starts more pods
- Unlike manually created pods, the pods maintained by a Replica Set are automatically replaced if they fail, are deleted, or are terminated

```
apiVersion: v1
kind: ReplicaSet
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```



Deployment

- A Deployment provides declarative updates for Pods and ReplicaSets
- You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate
- You can use deployment for
 - Rolling out ReplicaSet
 - Declaring new state of Pods
 - Rolling back to earlier deployment version
 - Scaling up deployment policies
 - Cleaning up existing ReplicaSet

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



Volume

- On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers
- Problems
 - When a Container crashes, kubelet will restart it, but the files will be lost
 - When running Containers together in a Pod it is often necessary to share files between those Containers
- The Kubernetes Volume abstraction solves both of these problems
- A volume outlives any Containers that run within the Pod, and data is preserved across Container restarts

Kubernetes

minikube

- installation
 - minikube creates a virtual machine with linux running in it
 - this cluster running in this virtual machine is simulated (not real)

```
# download the minikube
> curl -LO
https://github.com/kubernetes/minikube/releases/latest/download/minikube-
linux-amd64
> sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm
minikube-linux-amd64

# add the following command in ~/.bashrc
> vim ~/.bashrc
> alias kubectl="minikube kubectl --"
> source ~/.bashrc
```

- manage minikube

```
# start the cluster
> minikube start

# check the cluster status
> minikube status

# stop the cluster
# note: this command will NOT remove the cluster
> minikube stop

# delete the cluster
# note:
# - this command will delete everything (virtual machine) from the machine
# - once deleted, the cluster needs to be reinstalled (using minikube
start)
> minikube delete

# take the control (shell) of virtual machine created by minikube
> minikube ssh

# get the cluster information
> minikube kubectl -- custer-info
> kubectl cluster-info
```

```
# expose a service outside the cluster  
> minikube service <service name>
```

generic commands

```
# get a list of objects  
> kubectl get <object>  
  
# get details of selected object  
> kubectl describe <object> <object name>  
  
# delete an object  
> kubectl delete <object> <object name>
```

node (machine)

```
# get the list of nodes  
> kubectl get nodes  
> kubectl get node  
  
# get all the information about a selected node  
> kubectl describe node <node name>
```

namespace

```
# get the list of namespaces  
> kubectl get namespaces  
> kubectl get namespace  
> kubectl get ns  
  
# create a namespace  
# > kubectl create namespace <namespace name>  
> kubectl create namespace myns  
  
# delete a namespace  
# note: all the ns members will also get deleted while deleting the ns  
# > kubectl delete namespace <ns name>  
> kubectl delete namespace myns
```

pod

```
# get the list of pods in default namespace  
> kubectl get pods  
> kubectl get pod  
  
# get the details list of pods  
> kubectl get pods -o wide  
  
# create a pod using yaml configuration file  
> kubectl create -f <yaml file>  
> kubectl apply -f <yaml file>  
  
# get details of selected pod  
# > kubectl describe pod <pod name>  
> kubectl describe pod pod1  
  
# delete selected pod  
# > kubectl delete pod <pod name>  
> kubectl delete pod pod1
```

service

```
# get the list of services  
> kubectl get service  
> kubectl get svc  
  
# create a service using yaml configuration file  
> kubectl create -f <yaml file>  
> kubectl apply -f <yaml file>  
  
# get the details of selected service  
# > kubectl describe service <service name>  
> kubectl describe service myservice
```

AWS

EC2

- create an EC2 instance
 - region: Mumbai (ap-south-1)
 - name: web-server
 - AMI: Ubuntu Server 24.04
 - instance type: t2.micro
 - key: kdac-may21
 - disk: 10GB

```
# change the permissions of pem file
> chmod 400 ~/Downloads/kdac-may21.pem

# connect to the ec2 instance
# > ssh -i <pem file path> ubuntu@<public ip address of ec2 instance>
> ssh -i ~/Downloads/kdac-may21.pem ubuntu@43.204.218.134
```

EC2 instance configuration

mysql

```
# update apt cache
> sudo apt-get update

# install mysql server
> sudo apt-get install mysql-server

# start mysql using root permissions
> sudo mysql
mysql> ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password
BY 'root';
mysql> FLUSH PRIVILEGES;
```

backend

```
# download the nvm
```

```

curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash

# in lieu of restarting the shell
\. "$HOME/.nvm/nvm.sh"

# Download and install Node.js:
nvm install 22

# Verify the Node.js version:
node -v
nvm current

# Verify npm version:
npm -v 9.2".

```

- copy the server directory from local machine to ec2 instance
 - note: please execute these commands on your local machine

```

# go the parent directory of server
# > cd <parent>

# archive the server directory
> tar -cvf server.tar server

# upload the server.tar to the server
> scp -i ~/Downloads/kdac-may21.pem server.tar ubuntu@<ip address>:~/
```

- configure the server on ec2 instance
 - note: please execute these commands on ec2 instance

```

# go the home directory
> cd ~/

# unarchive the server.tar file
> tar -xvf server.tar

# go to the server directory and download all packages
> cd server
> npm install

# install pm2 (process manager) to run express application forever
> npm install -g pm2

# start the server in background

```

```
> cd ~/server/  
> pm2 start --name server server.js  
  
# get the list of servers running  
> pm2 list
```

- configure the ec2 instance to open a required port

```
# go to the management console  
# from instances list, select the required instance  
# go to the security tab  
# open security group  
# click "Edit Inbound rules"  
# add a new rule  
# - type: custom TCP  
# - port range: 4000  
# - source: Anywhere IPv4  
# - filter: 0.0.0.0  
# add a new rule  
# - type: custom TCP  
# - port range: 80  
# - source: Anywhere IPv4  
# - filter: 0.0.0.0
```

configure react / frontend

build the project

- note: please execute these commands on your machine

```
# build the react project  
# note: this command creates a dist directory with website files  
> yarn build  
  
# archive the website  
> cd dist  
> tar -cvf website.tar *  
  
# upload the website.tar file to the server  
> scp -i ~/Downloads/kdac-may21.pem website.tar ubuntu@<ip>:/
```

install a webserver (apache2)

```
# update the apt cache
> sudo apt-get update

# install apache2
> sudo apt-get install apache2

# go the htdocs directory of apache
> cd /var/www/html

# move the website from home to htdocs
> sudo mv ~/website.tar .

# unarchive the website.tar file
> sudo tar -xvf website.tar
```

Jenkins

installation

```
# install pre-requisite
> sudo apt-get install openjdk-17-jre

# visit the url: https://www.jenkins.io/

# download the key to connect to the jenkins apt repository
> sudo wget -O /etc/apt/keyrings/jenkins-keyring.asc \
https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key

# add the apt repository source for jenkins
> echo "deb [signed-by=/etc/apt/keyrings/jenkins-keyring.asc]" \
https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
/etc/apt/sources.list.d/jenkins.list > /dev/null

# update the apt source
> sudo apt-get update

# install the jenkins
> sudo apt-get install jenkins

# check the status of jenkins
> sudo systemctl status jenkins

# start the jenkins service
> sudo systemctl start jenkins

# enable the jenkins to start automatically
> sudo systemctl enable jenkins

# visit the url: http://localhost:8080
```

create a docker hub token

```
# visit the url: https://app.docker.com/settings/personal-access-tokens
# create a personal token with
# - permissions: Read, Write, Delete
# - expiration: None

# now you can login with the token
> echo <token> | docker login -u <user name> --password-stdin
```

setup jenkins for docker

```
# add the jenkins user to docker group  
> sudo usermod -aG docker jenkins  
  
# reboot the machine  
> sudo reboot
```

configure the job

```
# remove the service  
> docker service rm website  
  
# sleep  
> sleep 10  
  
# remove the image  
> docker image rm amitksunbeam/website  
  
# build the image  
> docker image build -t amitksunbeam/website .  
  
# docker login  
> echo <token> | docker login -u amitksunbeam --password-stdin  
  
# push the image to docker hub  
> docker image push amitksunbeam/website  
  
# start the service  
> docker service create --name website -p 9090:80 --replicas 2  
amitksunbeam/website
```

Add Docker's official GPG key:

```
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --keyserver hkp://keyserver.ubuntu.com:11371 --recv-keys E9D8BA4C407B6B17
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc
```

Add the repository to Apt sources:

```
echo
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc] https://download.docker.com/linux/ubuntu
 (https://download.docker.com/linux/ubuntu)
\\((. ./etc/os-release && echo \"\${UBUNTU_CODENAME:-$VERSION_CODENAME}\") stable" |
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

add your user in docker group

```
sudo usermod -aG docker $USER
```

restart the machine

```
sudo reboot
```

or <https://docs.docker.com/engine/install/ubuntu/> (<https://docs.docker.com/engine/install/ubuntu/>)

check the status of docker server

```
sudo systemctl status docker
```

start the docker server

```
sudo systemctl start docker
```

enable the docker across the boots

```
sudo systemctl enable --now docker
```