Virtualization/ Emulation

VMM/Hypervisor: Different comps that enable(exec/comm/bridge for guest/host) env for logical resources + env => **VM**
VMI: file that contains all VM info to run VI
Type 1: HW+VMM+GuestOS. Runs on top of host hardware (bare metal). Enables full control of hardware, better performance, scalability and stability. But limited HW support (driver support)
Type 2: HW+HostOS+VMM+GuestOS. Host-OS provides HW drivers. Enables flexible deployment, on-demand start/stop, parallel VMMS. But performance bottleneck through host, reliant on Host-OS
Em: Approx. behavior. **User mode** Built for one env but can be used on others if env is emulate. **System mode** full hw to enable virtual.(ARM vor x86)
Unikernels: compact, secure machine imgs created by compiling high-level langs directly into code, run on hypervisors or metal +fast -special manag/tool
Full: Type 1 and 2, guest OS not modified. Problems: device buses, privileged instructions, PT, Interrupts and timers, Disk and network IO. Dependent on arc **Partial**: Interface between VMM and OS -> Hyper calls. Direct access to HW features,+ capabilities. Modified OS kernel uses hyper calls to relay sensitive instructions via hyper call interface (HCI)

Kernel-based Virtual Machine (KVM)

Built into Linux kernel, turns it into a type-1hypervisor, inside Linux like =>type-2 hypervisor KVM uses VMX. KVM. Uses para virtualized I/O for performance(VirtIO). **Exec**: Load KVM kernel. User-space VM (QEMU) opens /dev/kvm and creates vCPUs. Guest code runs in guest mode, trapped to host when necessary. Memory => nested paging.

Storage Architectures

Storage Area Network (SAN): only block-level access over array of storage devices +speed
Network attached storage (NAS): file-level access
Distributed DAS: Use commodity HW to provide as service, requires middleware **Volume groups (VG)**: named collection of physical(provide block space) and logical(multiple disks +FS) volumes

Virtualization of Memory

VA -> TLB -> hit(-> PA); miss -> PT -> hit(->TLB write); not present -> PT write -> disk -> TLB write
SPT: VMM maintains Shadow Page Table. Then guest OS does itself VA -> PA. But then VMM PA -> HA (Host address). Slow and expensive
Hardware-MMU: CPU supports VA, PA and HA itself. MMU provides two-page table walkers. One for VM VA -> PA, second for VMM PA -> HA (Nested translations). VMM only does initial mapping, after CPU handles it directly, much faster.

Quick Emulator (QEMU)

Full-system emulation: Runs entire operating systems for any machine on any supported arc.
Virtualization: Uses KVM (Kernel based VM on Linux -> hypervisor) or Xen to run virtual machines with near-native performance.
Vanilla QEMU: uses binary translation and Soft-MMU to emulate a CPU entirely in software. Guest machine code is split into Translation Blocks (TBs). Each TB is executed completely (no jumps), translated and cached. Simulates what CPU does,not how

Virtualizing the X86 Architecture

Ring 0: Async interrupt, System call (TRAP).
Problem: Virtualized OS (guest OS) should not have ring 0 access, but certain instructions require it.
Binary Translation: VMM loads guest code into VMM memory, checks every instr and translates privileged ones in RT into ones that only affect the guest OS. Guest is not aware of this, heavy cost
VMX: Support from CPU for VMX instructions is needed. VMM loads guest into memory and enters non-root mode. If guest OS triggers privileged instruction the CPU triggers a VM exit, these hands back the control to the VMM in root mode. It will handle/simulate the instruction and trigger a VM entry, giving back control to the guest. The guest state is stored in Virtual Machine Control Structures (VMCS). This modern way is safer and faster.
Virt. Memory: Page Table handles mapping between VA and PA. Memory Management Unit maintains Translation Lookaside Buffer (TLB)

VirtIO (Provides standard. interfaces)

Guest OS: Uses **VirtIO device drivers** to access I/O devices. Modification required
PV-on-HVM FE: Exposes paravirtualized devices to the guest OS. Interfaces with the backend on the host via VirtIO.
PV-on-HVM Backend: Runs on the Host OS in root mode. Receives I/O requests from the frontend. Executes I/O on actual hardware devices.

Design Principles for Storage

Data Placement: Transparency to user => complexity is hidden. **Data Striping**: Segment sequential data into stripes and store in separate disks => Read&write parallelized=>increase throughput by N number of stripes, but if any fail!! **Data Replication**: Replicated across different failure zones with Replicas or **Erasure Coding** (Use Forward Error Correction. Divide into m blocks, encode into n>m blocks, store n across different disks, rebuild data with any n blacks if at least nr n=m. Efficiency m/n, n-m blocks=>disks can be lost)
Data Durability: Annual expected loss.
MTTF: Mean Time to Failure **MTTR**: Mean Time to Recover. Fraction of data lost / Years to lose it = Annual Loss => 100%- AL = Dura
Data Availability: 100%- Average error rate
System Availability: always, only comp fail
Consistency: across replicas=>COW
CAP: Impossible to provide **C** (At certain time all nodes see same data), **A** (can process and reply at any time), **P** (system remains operational if internal comp fails). **AC**: available and consistent, only if => single machine **PA**: available and partition-tolerant, => outdated data => DNS **PC**: consistent and partition-tolerant =>stop replying on fail
COW: Reuse resource if only read, once someone writes, create copy of elements that are modified but reuse rest

Libvirt Ecosystem

Libvirt: Hypervisor-agnostic library for managing VMs **Virsh**: CLI tool to control VMs via ^
Libvirtd: Daemon managing local/remote VMs, networks, storage. **virt-manager**: GUI
Domain: instance of an OS running on a VM

Docker (Podman replacement)

Container: runnable instance of image+config options, extra layer on image for write
Image: in reg, Dockerfile to build(new instruction= layer, only new layers are rebuilt)
Storage drivers: provide ephemeral storage for writeable containers=>deleted with cont
Union mount: combine multiple dirs. Into single view. Add UnionView +UpperDir. On change do copy-up(CoW), del whiteout
Persistent Storage: Volumes(part of docker area on fs), bind mounts(anywhere on host), tmpfs mounts(mem only)

OS-Level Virtualization

Host OS Kernel is shared => common denominator. The Host OS is responsible for isolating guests (Visibility Control(namespaces), Access & Usage Control(cgroups), Security) =>Apps run their own env on top
Linux Control Groups: associate / bind a collection of processes to a set of limits or parameters with kernel subsystem=>Controllers. Hierarchically(1multipleForProc) organized (1single,2unified) with inheritance. Virtual filesystem(ram) where specific contr are mounted to dirs, there rules/configs are saved. 1 hierarchy per cont(but mult cont per hierar)
Namespaces: Wraps Global sys res in abstraction, that proc within think they have their own isolated global res instance(Isolation) and Visibility of changes only within the namespace. Auto terminated(need 1 proc). PID also abstracted(only at creation) =>Isolation => Dupl. PID possible. 2 can be joined with setns
LXC: Native Container with cgroups+ namespaces+chroots(isolated root-fs)+LSM&MAC

Nova (Compute Service) Libvirt

Scheduler: Decides which host gets each instance via Filtering and prioritizing
Compute: Communicates with Hypervisor and VM, checks status and health
Conductor: Handles requests with coordination(build/resize), database proxy
Placement: Tracks resource provider inventories and usages. Used by Scheduler

Nova Resource Management

Cells: Each Cell has own DB, queue, Scheduler, Conductor and compute. Communication with Nova through own message bus and cells service. Only visible to Operator
Regions: Complete deployments but shared Keystone and Horizon, physically separated
AZ: Grouping of Hosts for SPF (Power, Network, etc.). User selectable, 1+ Regions
Host Aggregates: Grouping for Admins based on Hardware. User selectable **Flavors**

Neutron (Network Service)

Server: API, hosts DB for network model and plugins, communicates with o. services
Plugin Agent L2: On each compute node manages local virtual switch (connect VM NIC to network) **L3:** Routing and NAT for different subnets and Floating IPs for external connections **DHCP**: Assigns Ips via dnsmasq
Provider Networks: DC net infra with preconfigured VLANs. Handle only Layer 2 and require admin provisioning. Ext L3 faster

Object Storage – SWIFT

Data not organized, just buckets => good for unstructured data. No update. Account (Owns all resources, defines namespace)/Container(defines namespace, uses ACL, stores policies)/Object(Unlimited, compression, URL) => URI. It is transparent, every object has metadata, offers data replication, concept of failure domains, storage policies
DiskFile API: Enables vendors custom backend implementation. GET/UPDATE

Definitions Network

Oversubscription: ingress capacity >egress
DC Requirements: support different stakeholders, robust and redundant, modular and support heterogeneity, simple in the scaling characteristic, flexible in the topology, efficient and effective, isolate tenants
L2 virtualization: LAN -VLAN at layer 2 internal. WAN -MPLS external Connection oriented mechanism to forward packets by a pre-signalized path through a network
VLAN: Port-based VLAN simplest mode and fast. Tagged VLAN adds a dedicated 4 Byte VLAN-field to the Ethernet Header, can be tagged at departure or arrival. DYNAMIC /REMOTE to request info for tag from central instance or local. **Trunk(link)** if 2 are connected via 1 link. A trunk port forwards all frames, regardless of the VLAN-ID
WAN: Multiprotocol Label Switching (MPLS) forwards any frame based on a Label fast, because no IP lookup is needed**. Label Switch Router (LSR)**: decides based on the label which is the next hop and Performs label operation **Label Edge Router (LER)** is exit or entry point adds label based on forwarding equivalence class and pops the last label and forwards it based on IP routing table
Labels MPLS: can be stacked, 20 bits label,3 QoS, 8 TTL
Flat Networking: single net, private cloud
Multi-Tenant: Tenant has own net&router to connect to ”shared”=>external net
VLan Segmentation Policies: Depends on cloud type. Typically, between Management, Storage, User (internal) aggregate traffic => admin. Instance isolation => user
Routed Provider Networks: Segments allocated to groups of nodes. L3 services to enable connectivity beyond segment. From user perspective, all segments are on one or several L3 networks

Block Storage – Cinder

Raw Storage volumes for VM’s, usually needing FS supports local or remote volumes. **Api**: interface for end user **scheduler**: decides where to put volume **volume**: Use drivers to interact directly with storage backend **backup**: store and external system and recover **Bus**: API goes trough bus, not directly scheduler
Running Instance: for OS, env variable and temp data, deleted with VM
Persistent: Exists without VM=>user data
Local: physical drive at compute host. Faster but if compute host fails data is trapped and difficult migration . Use Linux LVM
Remote: high reliability=>needs fast network
iSCSI: Internet SCSI Protocol to access block devices over a network: target (the server exposing storage res), initiator (the client accessing exported storage res), TPG (target portal group from which block devices are exposed), LUN (Logical Unit ident block device on TPG). Categories:Non data, read, write, bidirectional

Glance (Image Service) & Keystone

VM Image registration and Storage
Booting flow: Request (Nova) -> Lookup (Glance)-> Transfer (Nova)-> Boot
Keystone: Verify user cred, perms and provide service catalog (all endp with IP), token based

GRE (Generic Routing Encapsulation)

logical tunnel where IP packets are encapsulated by an IP frame (IP-over-IP) followed by the GRE header. **Encapsulation**: Internal net send packet to tunnel interface. Tunnel adds GRE header (includes checksum,version,protocol) . Then add enclosing IP header, GRE header is removed at end of tunnel

ZFS (Snapshots only delta=>clone)

FS that organizes physical disks into vdevs=> into pools=>configure pool features and allocate volumes=>expose volumes at block-level or file-level. Everything is COW, transactional and check summed

Neutron - Self-Service Networks

Routers, subnets, DHCP, external connection on-demand
Networking Node: run deployment-global networking logic, manages namespaces per router and DHCP
Floating IP Address: North-South: ext interface => br-ex => br-int => qg interface qrouter, which contains floating IP => br-int to add internal tag => then replace tag with VLAN or wrap for GRE **Static tunnels**: old and stable. Create vxlan tunnel to all compute nodes, suffer from broadcast storms
Dynamic tunnel configuration: create only on demand
High Availability: Operate redundant services providers. Stateless(No dependency between requests) vs Stateful Services(comprises several interactions=>migration of state on failure). **Active** All providers are providing service Failover=>degraded. **Passive** Primary and Passive Nodes. **Fallback** restore initial config. **Switchover** manual switch

Kubernetes

Pods: Min unit. Part dies=>replaced. horizontal scaling. Replica Sets: how many and when to restart. 1+ container that are affine, share same pod env(IP), multiple => **Cluster Volumes**: Either same lifetime as pod(emptyDir) or persistent(hostPath,local). **PersistentVolume** storage in cluster provisioned by an admin or dyn with storage class
Workload Management: Controller (algo that drives actual state to desired state), Replication, Deployment
Deployments: use Replica sets. User creates Deployment object, controller compares that to actual state, does change at controlled rate, on fail rollback. **Rolling update** so service is never interrupted. **AutoScaler** for horizontal 1-10 pods scaling by php-apache left
Service: Pods that work together as one multi-tier app. Provided Service discovery(CluserIP,NodePort), routing(ExternalName) and load balancing. New IP every change on Pod=>Services **Containers**: Application, Sidecar(support) and Init. Loosely or tightly(same pod) coupled
Master node: API Server(watches), etcd for key/value configs, scheduler(which pods run where& when), controller-mananger (process where controllers run), Cloud controller manager(Links cluster to cloud provider) **Worker node**: Kubelet (state of the node), Kube-proxy (proxy and load-balancer and routing), Runtime(any CRI like docker), cAdvisor (monitoring), overlay network (connecting containers)