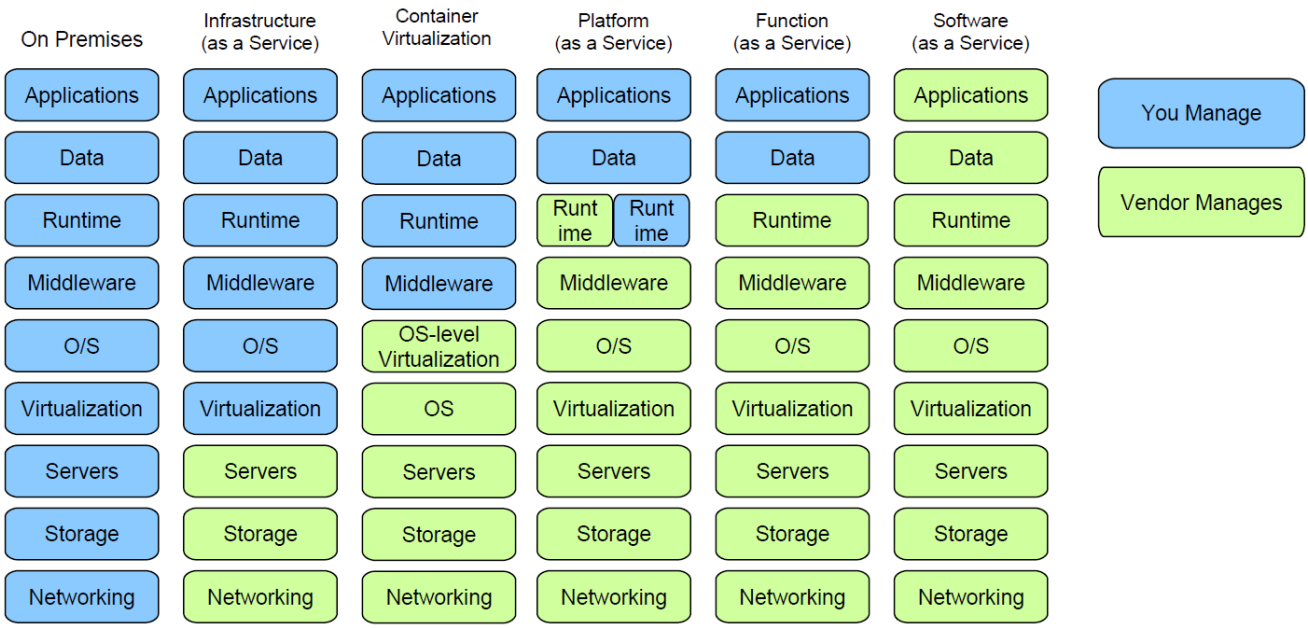
CCP1

CCI

- capital expenses / CAPEX: incurs when spending money to buy/repair/upgrade fixed/intangible assets cost cannot be deducted
  - hardware, software, licenses
- operational expenses / OPEX: ongoing cost for running a product/business/system
  - rented space, electricity, backup+recovery, audit, staff
- total cost of ownership TCO = CAPEX + OPEX
- ICT consumption models: stable, periodic busting, predictable busting, on/off, scheduled increase in capacity (jump or smooth)
- cloud: self-service, on-demand, elasticity, resource pooling / multi-tenancy, pay-as-you-go, broad network access



- private cloud: single org; community cloud: specific group; public cloud: open use by general public; hybrid: any combination

OSTK

- OpenStack is IaaS
- Nova: compute, Glance: images, Swift: object storage, Cinder: block storage, Neutron: networking, Keystone: identity service, Horizon: dashboard, Heat: orchestration
- API >> dashboard
- Heat uses HOT YAML templates; important keys with sub-keys:
  - `heat_template_version`, `description`, `conditions`
  - `parameter_groups`: `label, description, parameters, param name`
  - `parameters`: `param name, type, label, description, default, hidden, constraints, immutable`
  - `resources`: `resource id, type, properties, metadata, depends_on, update_policy, deletion_policy, external_id, conditio`
  - `outputs`: `param name, description, value, condition`

KUBE

- containers run directly on host OS, with limited+prioritized and accounted access to resources (and visibility of these resources is limited) with each container having it own rootfs
- Kubernetes/k8s is a platform for automating deployment, scaling, and management of containers; can run on OpenStack
- master node / control plane: etcd (K/V storage of cluster state), API server, scheduler, controller manager
- worker/k8s nodes / minions: kubelet (responsible for running state of each node), Kube proxy (routing etc.). cAdvisor (metrics agent), overlay network (not part of k8s)
- cluster: set of machines where pods are deployed, managed, scaled
- pod: one or more containers, guaranteed to be co-located ("logical host"); basic unit of scheduling; lifecycle: pending, running, succeeded/failed, unknown
- controller: reconciliation loop driving actual to desired cluster state
- service: set of pods working together; exposed via ClusterIP, NodePort, LoadBalancer, ExternalName; expose a reliable networking endpoint (and keep track of the (ephemeral) pods behind them)
- label: k/v pairs, very mighty
- consistent object API, providing object metadata (name, UID, version, labels), spec (desired state), and status (current state, read-only)
- container cluster management: robustness, deployment+maintenance w/o service interruption, optimize usage/cost
- scheduling: placing containers on nodes; affinity (tight cooperation) or anti-affinity (redundancy)
- replication controller: replicas of pod definition; replaced by deployments
- deployments: rolling updates, rollbacks
- replica set: extension of replication controller, enforces desired state of running replicas of a set of pods

CMP I

- improve utilization by pooling physical resources (compute: CPU, RAM, caches, interrupts, timers, I/O etc.); reduces cost, allows for scaling and on-demand
- simulation: model of the system, no direct link to actual system; emulation: approximate system behavior, possibly with different implementation; virtualization: using parts of or the entire actual system
- emulation: allows for portability; e.g. QEMU

- virtualization components/types: VMM/hypervisor; full-, para-, OS-level-, application-level- virt.; VM, VMI
- properties of env created by VMM: equivalence/fidelity (same behavior), resource control/safety (VMM in complete control of virt. res), efficiency/performance (significant number of statements w/o VMM intervention)
- type I: VMM (baremetal hypervisor)
  - +: performance, scalability, stability, no host OS to maintain
  - -: limited hardware supported, not suitable for workstation
- type II: host OS
  - +: wider h/w compatibility, flexible deployments, on-demand start/stop, parallel to host OS, parallel VMMs
  - -: host OS is a potential performance bottleneck, wider system-breach surface
- x86 uses rings 0-3 (most to least privileged); ring 0: async interrupt, syscall/trap, exception or illegal instruction (page fault, div by 0)
- virtualizing x86 is hard b/c semantics of some instructions (e.g. reboot) is different when in ring 0 or not; instead trap sensitive instructions + virtualize (using binary translation) their execution; guest is unaware of virt.
- AMD-V/Intel VT: privileged instructions running in a new CPU execution mode feature; no more need for binary translation or para-virt.; aka HVM
- x86 MMU virt.: from virt virt to virt physical to host address by using shadow page table maintained by VMM; again: can have hardware support obsoleting shadow page table
- Linux commands: `vmstat, free, pmap, top, sar -B, /usr/bin/time, cat /proc/sys/vm/freepages, cat /proc/meminfo`
- full vs para virt: full: complete simulation of underlying hardware (type I & II) + support for unmodified OS; para: tell guest it's being virtualized
- para required a new interface between VMM and OS to virtualize sensitive instructions called hypercalls; OS modifications required (swap in hypercalls); OS is run similar to a user land application with syscalls
- full virt. simplifies CPU, bot not other components -> para virt useful
- QEMU: full-system emulation, user-mode emulation, and virtualization (KVM, Xen)
- QEMU uses binary translation and SoftMMU
- KVM: kernel-based VM, default hypervisor in OpenStack; makes standard and advanced Linux features available to guest
- KVM provides an interface to Linux kernel via a loadable kernel module, no virt/emulation (!)
- KVM integrates cgroups (control groups), resource scheduling, and network namespaces
- virsh ("virtual shell") uses libvirt + XML configs to creates VMs

CMP II

- Linux cgroups control resource access using hierarchies (supporting multiple hierarchies simultaneously) and subsystems (BLKIO, CPU, cpuacct, cpuset, devices, RAM, NET_CLS, NET_PRIO, NS); provide monitoring and are dynamic and persistent
- used using `mount -t cgroup -o subsystems name /some/dir` and `echo`ing PIDs into files
- Linux namespace provide isolation (processes have their own isolated instance of global resource) and visibility (change to global resource are visible to processes in that namespace, but not to others); often used for containers
- chroot changes root directory of calling process
- Linux Native Container Management LXC uses cgroups, namespaces, chroot, and LSM&MAC for security; enabled via kernel features

- Docker features: portability, application-centric, build automation, versioning support, component reuse, sharing, tools (CLI, REST API)
- when removing a Docker container, any state changes not persisted in storage disappear
- images (built using a Dockerfile; each instruction results in an image layer) are read-only templates to create a container
- Docker uses a union mount to add read-only fs'es on top of the rootfs
- Docker commands: `run -ditp, ps, inspect, stop, kill, rm`
- Dockerfile: `FROM, RUN, COPY, EXPOSE, CMD, ENTRYPOIONT`
- env vars are used a lot, also for config and linking

STRG1

- cloud storage provides logical (network-available) storage abstracting a complex, distributed infrastructure; separate storage service/deployment and block/object storage
    - requirements: flexible volume management, multi-tenancy, thin-provisioning, accounting and user policies, no operational downtime, privacy and security
- block storage: VM volumes, most basic apart from bare metal, requires fs as data access is using block arrays
- file is an OS abstraction over blocks
- Storage Area Network SAN: only block-level, no files, using iSCSI; storage device array is available over network
    - formattable and mountable block devices; scale well; expensive (non-commodity h/w)
- Network attached storage NAS: have fs and offer file-based access
    - "single boxes"; simple; less flexible due to abstraction layer over block storage
- data placement is handled transparently by storage system, using policies, and rebalanced
- data striping improves efficiency by factor N for N stripes; failure of any unit implies data loss
- data replication provides fault-tolerance using replicas or erasure coding (ZFS software RAID)
    - divide data into m blocks; re-rencode into $n > m$ blocks; rebuild data with $m \leq k \leq n$ blocks
    - storage efficiency: m/n ($n - m$ redundant blocks = tolerable loss)
    - e.g. radiz3 pool with 8 vdevs, $m = 5, n = 8$ has $5/8 = 62.5\%$ efficiency, tolerating the loss of 3 vdevs
- durability ("data permanently lost") L/Y where Y is the number of years it takes to lose a fraction L of the data; durability loss implies availability loss
    - e.g. $L = 0.00000001\% \rightarrow 99.99999999\%$; when storing 10000 objects, lose $10^{-10} \times 10^4 = 10^{-6}$ objects per year or 1 object every 10^6 years
- availability ("data not available") is a result of uptime; S3: 99.9%
- more general design principles: availability, scalability and efficiency, consistency
- CAP - Consistency, Availability, Partition Tolerance conjecture: a distributed computer system can provide at most two of three guarantees; e.g. split brain: allow access and no consistency, disallow access and no availability
    - C: at a certain instant of time, all nodes see the same data
    - A: the system is able to process and reply to client requests
    - P: the system remains operational even in case of failure of single components
    - AC: available and consistent systems, can only be achieved if systems run on a single machine; e.g. non-distributed DB
    - PA: available and partition-tolerant systems, may provide outdated data but clients “always” receive a reply (e.g., DNS system)

    - PC: consistent and partition-tolerant systems, may stop replying to client requests, but provided data is always consistent; e.g. flight booking system
- Cinder provides block storage; software infrastructure for volume management, supports local (directly attached storage DAS) and remote volumes (SAN, NAS)
- compute hosts in OpenStack may or may not have local storage
- DAS: +: simple, performance; -: larger footprint on compute host, local storage on compute host (incl. RAID), possibly less failure-resilient
- iSCSI provides block-level access over a network using a target and initiator; TPG = portal group from which block devices are exposed; LUN = logical unit identifying block device on TPG
- ZFS is a very powerful local fs which includes COW-everything, transaction-everything, checksum-everything, snapshots, cloning and uses a software-defined approach to manipulate storage resources; organize physical disks into vdevs (mirror, raidz1/2/3), vdevs into pools, configure pool features and allocate volumes, expose volumes at block-level/iSCSI or file-level/distributed fs

STRG2

- object storage has no multi-level hierarchical structure, only contains and objects, w/o nested containers i.e. buckets instead of trees; suitable for unstructured data
- object is a data blob with a unique name / ID, optionally with k/v metadata
- S3 API is de-facto standard; CRUD / RESTful
- object may be a file, but doesn't have to be
- in OpenStack: Swift; distributed, eventually consistent, redundant, scales well, has ACLs, preserves integrity, allows for replication, failure domains (regions, zones), swapping nodes w/o downtime
- Swift data is organized in flat account-container-object hierarchy and has three rings, storage, container, and object
- Swift is behind a stateless proxy, which is responsible for API, HA+HP
- account server uses account ring to maintain a list of containers; same for container/container/objects and object/object/object locations
- in the object server, there is one object ring per storage policy
- rings maintain data-partition-zone-device mapping
- devices are placed in different failure domains and partitions can be across one or more devices
- each partition in a ring is replicated according to a replica count (default: 3) which are balanced and dispersed into different failure domains (given capacity) according to a device weight
- `partition index = (hash(item path) % C) % N` where `C` is the number of bits kept from the hash (aka partition power) and `N` the total number of partitions in the ring
- object rings (one per storage policy) allow for different levels of durability, performance, node grouping, and storage implementations
- Swift & CAP: AP (might provide outdated data)
- Swift workflow follows CRUD/REST - POST (partial)/PUT (full)/GET/DELETE
- Ceph: unified object, block, file store on top of object-based system; reliable autonomic distributed object store RADOS
- key components: monitor daemon MON, object storage device OSD, and meta data service MSD
- interfaces: native, object, block, file
- Ceph CRUSH - controlled replication under scalable hashing

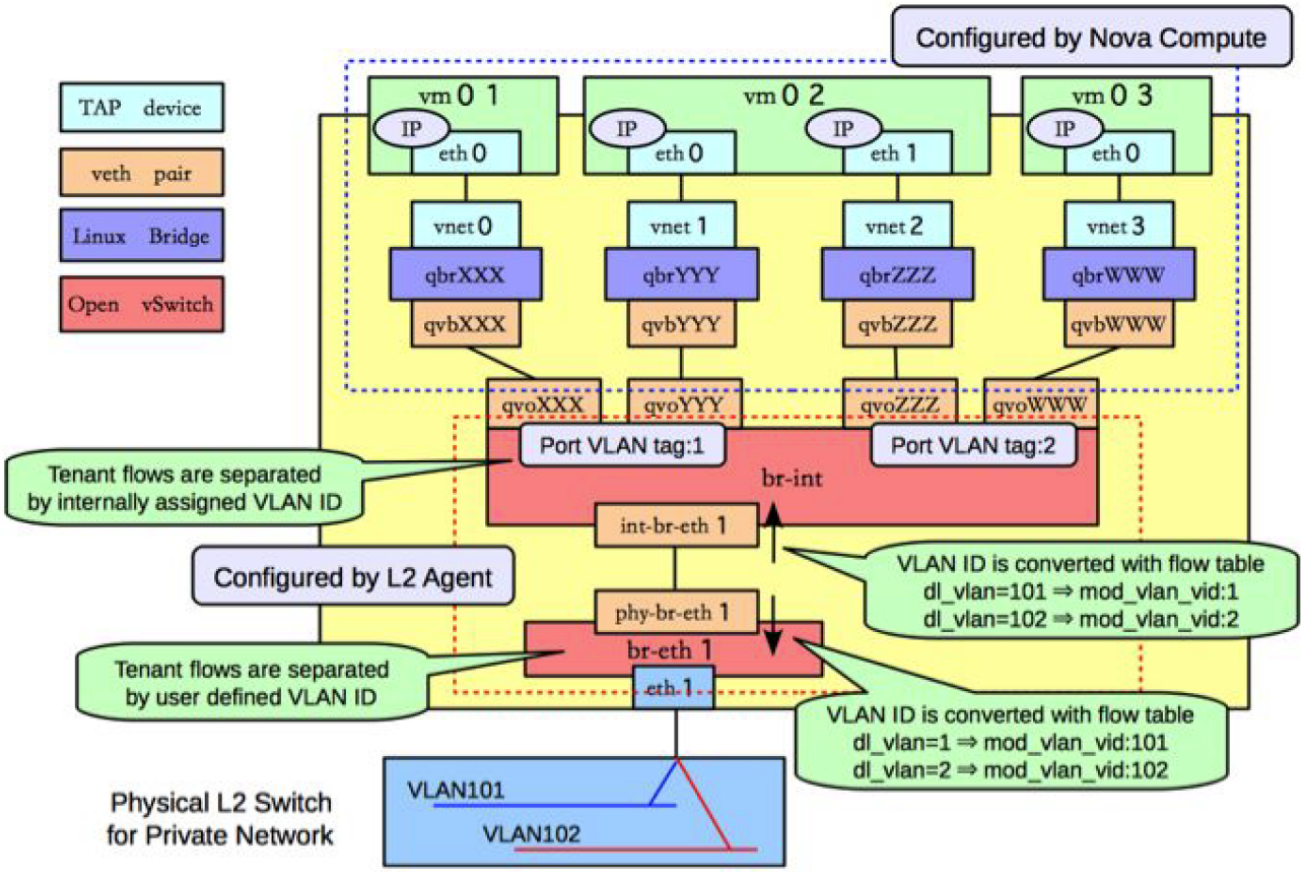
STRG3

- AUFS (union fs) uses COW; for reading, uses existing (in lower layer) file; for write, copy-up file into layer (original file is "hidden" and unmodified); delete: if only in topmost, remove else whiteout but don't modify lower layers

- ZFS fs is thinly-provisioned, on-demand allocation from zpool; snapshots are read-only space-efficient, point-in-time copies, clones are read-write copies of snapshots
- ZFS can be used for Docker, too where each child layer is a ZFS clone based on a ZFS snapshot below it
- ZFS read: fast, even from deep layers
- ZFS write: on-demand space allocation, written into the container's writeable layer
- ZFS modify: allocate space for changed blocks, COW on writeable layer
- ZFS delete in lower layer: mask file/directory
- ZFS delete in writeable layer: free blocks, reclaimed by zpool
- storing data in writeable layer has disadvantages: non-persistence, difficult to extract, tight coupling to host, storage driver to manage interaction
- alternatives for Docker: volumes, bind mounts, tmpfs
    - volumes are stored as directories on host and can be shared among containers and exist independently of containers
    - binds are "shared folders" between host and container, very performant, unadvised by Docker
    - tmpfs: high-speed, non-persistent e.g. secrets
- Docker Swarm is a "service"

NET1

- a DC has to connect sites and racks, and to the Internet
- internally, there are a lot challenges and solutions, none of which are ideal; link aggregation, ECMP, rack-to-rack, L2 broadcast domains, spanning tree
- externally, multiple ingress and egress
- DC requirements from operator perspective: robust, redundant, modular, heterogeneity, simple in terms of scaling, flexible topology, different stakeholders, efficient, effective, isolate tenants
- standard technologies: L2 virtualization uses VLAN, (G)MPLS; VXLAN and GRE (both encapsulate protocols, Ethernet or arbitrary, resp.)
- standard 802.3 (Ethernet) has no support for QoS, added by 802.1Q aka VLAN: logical subnet inside one (or spread over multiple) L2 device(s)
- port-based VLANs hard-ware ports into subnets; simple, fast
- tagged VLANs add a VLAN-field to header, with 12 bits for the ID ($2^{12} = 4096$)
    - tagging on departure or arrival, either dynamic/remote (e.g. from policy server) or static/local
- one link connecting to L2 devices, is a trunk connecting trunk ports; trunk ports forward all frames, regardless of VLAN ID (tag) and adds frame information (port-based)
- MPLS: multiprotocol label switching; forwards any frame based on a label, independent from Ethernet
    - LSR: label switch routes; push/pop/swap labels
    - LER: label edge router; ingress and egress
- General Routing Encapsulation GRE: logical tunnel where IP packets are encapsulated by an IP frame (IP-over-IP) and followed by GRE header
- flat networks: private cloud scenario, full connectivity
- flat and private network: enterprise/campus with private and public domains
- multi-tenant and multi-tier: public clouds hosting multiple tenants; tenants require control over own network topology



- OpenStack uses iptables
    - chains: INPUT, OUTPUT, FORWARD
    - policies: ACCEPT, DROP, REJECT, QUEUE, RETURN

NET2

- software defined networking SDN programs network behavior instead of configuring it; separates control and data plane
- OpenFlow is an SDN protocol
- in packet switching networks, traffic flow, packet flow or network flow is a sequence of packets from a source computer to a destination, which may be another host, a multicast group, or a broadcast domain
- TPCI/IP uniquely identifies a flow with: src+dest IP+port and L4 protocol; used for matching flows
- a flow table contains flow entries/flows, each with instructions, and action set, and a forwarding policy
- datapath consists of flow table(s), one group table, ingress/egress ports, channel to controller
- in OF, ports can be physical, local, or logical
- actions include: write (output port, drop, group id, push/pop tag (-> MPLS/VLAN), set field), clear, meter (meter band(s), each defines rate, band type, counters, args), goto-table, write metadata
- OpenDayLight is an SDN framework using OpenvSwitch OVS a a virtual and intelligent bridge
- in OpenStack: Neutron is SDN

NET3

- OpenStack has two types of networks
    - provider: integrated into DC network infrastructure, using preconfigured VLANs; L2-only; no self-service by user, admin-only; higher performance since h/w; relies on external L3
    - self-service: flexible; higher abstraction than L2; user can L3 overlay networks; mostly SDN

- VLAN segmentation for isolation of traffic categories, admin tasks and/or of tenants
- In OpenStack, Neutron nodes run deployment-global networking logic, namespaces
- static tunnels: older, more stable; vxlan tunnels to all compute + controller nodes; might suffer from broadcast storms
- dynamic tunnels: on-demand
- Neutron provides L3 services for self-service networks; inlcuding HA
- HA (high availability): load balancer in front of providers; services can be stateless/stateful, active/active (all providers are providing; if stateful, share state; "degraded" mode until recovered) or active/passive (primary and secondary; if stateful, migrate state)
- take-over: failover, failback (re-establish initial config), switchover (manual change of roles)
- keepalived is used for L3 HA using OVS and Virtual Router Redundancy Protocol (VRRP); keepalived provides hooks for VRRP FSM
- HA can also be achieved with Distributed Virtual Routing
- Docker Engine: dockerd, REST API + CLI; beyond Docker Swarm or k8s
- Docker networking: default and user-defined ("provider and self-service") networks; between containers on same host, between contains on different hosts, connectivity to Internet; by default: `bridge, none, host`
- complex Docker networks: bridge/overlay/MACVLAN/custom networks incl. container communication control, IP allocation, DNS resolver

PERF1

- sys perf eval requires structured approach, has to cover entire system lifecycle, and knowledge across the board
- defines several activities in the different phases of the lifecycle (design, implementation, deployment, provisioning, operation, disposal):
    - define the System under Test (SUT); setting performance objectives; define performance characteristics and models; performance analysis of development code; tests of software builds; benchmarking software releases; PoC testing in the target environment; configuration optimization of the production environment; monitoring and analysis of issues
- SUT approaches: models + simulation, or empirical investigation using emulation or production grade implementation
- workload: load imposed onto the system
- perturbations: parallel processes on a single-core CPU
- metric: quantification of system behavior
- resulting performance: observed behavior
- IOPS: I/O per second
- throughput: data per time or transactions/operations per time
- response time: time to complete
- latency: time spent waiting
- utilization: how busy a resource is; can be time or capacity based; time-based is usually displayed in `top` etc.; time != capacity based
- saturation: degree to which a resource has queued work it cannot service; begins to occur at 100% utilization
- bottleneck: resource limiting overall sys perf

- cache: fast storage buffering certain elements to accelerate I/O ops
- stats can be observation- or experiment-based
- metrics are often averaged out, hiding spikes; percentiles more useful
- not everything is necessarily Gaussian distributed; e.g. caches are bimodal (hits and misses)
- QoS (objective) vs QoE (subjective); need not imply one another
- caches can be cold, warm, or hot (hit ratio > 99%)
- resource (focus on utilization) vs workload (focus on requests, latency, completion) analysis
- problem statement is vital and should be done first
    - What makes you think there is a performance problem?
    - Has this system ever performed well?
    - What has changed recently? (Software? Hardware? Load?)
    - Can the performance degradation be expressed in terms of latency or runtime?
    - Does the problem affect other people or applications?
    - What is the environment; what software and hardware are used?
- workload characterization: method to identify issues due to the load applies; focusses on input (who, why, what, how?)
- USE method: for every resource, check utilization, saturation, and errors