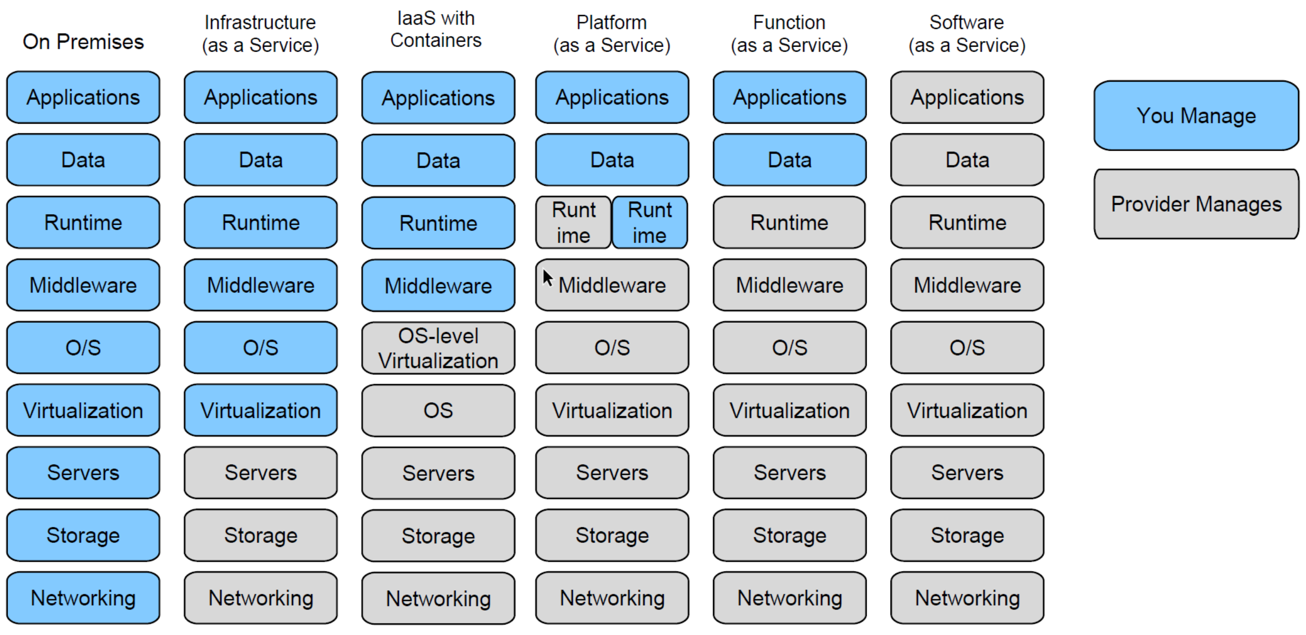
Cloud Computing 2

CC2I

- 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
- Definition of PaaS: The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment
- PaaS: OPEX, standardized envs, continous improvment, supports agile org and dev, provider takes full responsibility for platform
- there are different types of PaaS: application for multiple runtimes and frameworks (OpenShift); integration; DB; business process management; business analytics; mobile backend; in-memory datagrid etc.; some are domain specific (e.g. healthcare, robotics etc.)
- low lock-in: UBM Bluemix, Pivotal, Swisscom, OpenShift etc; high lock-in: Microsoft Azure, Google App Engine, Heroku etc.
- cloud: self-service, on-demand, elasticity, resource pooling / multi-tenancy, pay-as-you-go, broad network access



PAAS

Swisscom marketing lecture

CNA1

- benefits of cloud computing: improve time-to-market; optimize cost
- drawbacks: exposure to failures (commodity h/w); performance issues (resource pooling)
- avoid both under- and over-provisioning
- life-cycle: design -> implementation -> deployment -> provisioning -> operation & runtime management -> disposal
- A cloud-native app is optimized for running in the cloud to exploit the economic value proposition of cloud computing whereas each phase in the life-cycle has to be adopted and optimized for a cloud env. Typically, such an app is designed a distributed fashion.
  - arch: designed for scalability and resilience
  - org: devops, agile teams
  - process: CI/CD
- SOA: service-oriented architectre; think in terms of services and service-based deployment and outcomes
  - standardized protocols, abstraction loose coupling, reusability, composability, stateless, discoverable
- microservices: SOA style; many small services, each its own process, lightweight communication, built around business capabilities, independently deployable, minimal and centralized management
- Conway's law: Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
- Twelve Factor App
  1. Codebase: always in VCS, only one codebase with many deploys each with same codebase, but different versions in each deploy
  2. Dependencies: explicitly declared in a manifest and deps are isolated
  3. Config: = what is not the same in every deploy/env (creds, resource handles etc.); stored in the env itself via env vars; strict separation of config from code; use env vars and not files as not to check them into VCS

4. Backing services: = a backing service consumed over the net as port of its normal operation; code doesn't distinguish between local and 3rd party svcs; (de-)attachable at any time w/o changing code
5. Build, release, run: strictly separate build and run; no code changes at runtime (no way to back-propagate); builds initiated by devs, runtime automatically (reboot, process restart etc.)
6. Processes: stateless and share-nothing; don't assume cache exists; persistent date in a stateful backing service; sticky sessions should not be used instead use Redis or memcached w/ expiry
7. Port binding: completely self-contained and not relying on runtime injection of a webserver; export HTTP server and bind to a port
8. Concurrency: scale out processes horizontally and independently for each proc type (array of types and number is the process formation)
9. Disposability: maximize robustness w/ fast startup and graceful shutdown; start/stop at any time w/o lead time; on SIGTERM, free resources, unsub, release locks etc.; also robust against sudden death (h/w failure)
10. dev/prod parity: CI/CD relies on dev, staging, prod being as similar as possible; avoid gaps between dev and prod (time gap: fast release; personnel gap: dev/ops, tools: same tools for dev and prod)
11. Logs: treat as event streams, all running procs log; the app isn't concern with routing/storing its log stream instead aggregated by platform for indexing/analysis/alerting
12. Admin processes: admin/management tasks (e.g. DB migration) as one-off procsw/ same release, codebase, config, deps etc.; shouldn't be done by app in startup phase

CNA2

- app is a CNA if it implements cloud patterns: service registry, circuit breaker, load balancer, API gateway, endpoint monitoring etc.
- a service has: (business) functionality; config space; interface, protocol, information mode; operational features and reqs; 0-N instances and their respective state
- a service instance is an instantiation of a service: has a state (functional and operational)
- service registry maintains state information of service instances: maps services to service instances; how to reach a service instance; how to use/talk to it; and its state; register initial state and maintain it until disposal
- Netflix Eureka: REST-based, Java, service registration & lookup, replication of server configs, easy integration w/ Spring, actions: register,renew,cancel,getRegistry
- etcd: distributed KVS, run as HA cluster, uses Raft, data expiry, notification, shallow data structure, maps keys into folder, actions: read,write,listen
- circuit breaker: deals with unavailable/unresponsive services (down, slow net, overloaded etc.); CNA should be able to handle such unreliability with minimal influence on it
  - consumer: detect failure fast, temporary request hold, do not block indefinitely, allow for reaction from invoker
  - provider: limit requests to allow for recovery
  - acts as an intermediary for mediation between consumer and provider; open/closed/half-open
  - closed: passes every request; on failure, increase failure counter; failure counter above threshold -> open; reset counter after timeout
  - open: immediately return error; after some time, change to half-open OR sporadically ping to check health
  - half-open: pass some request, error on most requests as not to overwhelm a recovering service; count successful requests and change to closed after passing a threshold; on failure, immediately change to open
- load balancer: elasticity when scaling; CNA should scale with load, either horizontal (parallel resources / service instances; preferred) or vertical (CPU, RAM etc.); distribute load to a service instance pool
  - server-side (e.g. HAProxy) in separate process, shared by clients and often by multiple services
  - client-side as lib in the client in same process, client decides which instance to connect to, requires registry

    - distribution according to fairness, load, speed, economics; alogs include round-robin (fair), least-connection (performance), source (stickiness)
- API Gateways: consistent presentation to consumers; a CNA should provide a clear, consistent interface, not exposing internal structure
    - approach: gateway service providing one unified interface, forward requests to respective internal service, facade pattern
    - advanced: client-specific variants; aggregate responses from concurrent requests; convert protocols
- endpoint monitoring: track operational status; proc might be running, but app could be crashed/stack; CNA implements functional checks and provides interface for specific metrics -> external tools gather and analyze metrics
    - typical checks: proc status (resource consumption load etc.); check presence of external backing svc; measure response time; check SSL expiry; valid HTTP response code; check content of response
- Queue load-leveling: deal with variable load levels; random load and random arrival of requests, load peaks; implement a queue to smoothen requests; policing/droppping requests during excessive peaks
- competing consumers/producers: provide elasticity; implement message queue and buffer + distribute to svc instance in pool; enable async requests and variable amount of requests
- event-sourcing: track state and provide transactional features in a highly distributed system; store event stream to then update component state (materialized views); recreate states in the past by replaying even stream
    - eventual consistency for transactional data; full audit trails and history
- Command query response segregation (CQRS): optimize for read and write ops; separate paths for read and write ops; write/commands modify model/state; read/query provide data from separate, read-optimized models; overload shouldn't affect other path; often combined with event sourcing
    - easier to implement (single concern); independent, demand-based scaling

CNA3

- message passing: integral part of any distributed application; CNA heavily relies upon app-level messaging
- sync/async communction: blocking/non-blocking; uni-/bi-directional: one-way, e.g. UDP / onward+return over same channel, e.g. TCP
- unicast: one sender to one receiver; multicast: one sender to many receivers; anycast: one sender to any receiver; broadcast: one sender to all receivers
- Channels, also known as queues, are logical pathways to transport messages. A channel behaves like a collection of messages, but one that is magically shared across multiple computers and can be used concurrently by multiple applications.
- message-oriented middleware (MoM) to facilitate communication: abstraction of communication layer; implement one or more channel patterns; support industry standard(s) of messaging; allow for language- and platform-independent dev
- messaging channel patterns: point-to-point; pub-sub; datatype (one channel only has one data type); dead letter (bad/undelivered/erroneous messages); guaranteed delivery (uses built-in data store for persistence until delivery)
- MoM classification: brokered (choreographed by well-known server i.e. broker; every app connected to central broker; broker can become bottleneck; message passing overhead; e.g. RabbitMQ) vs brokerless (peer-to-peer; often allow for entity (e.g. pubs, topics etc.) discovery; distributed caching for reliable delivery; e.g. DDS, ZeroMQ)
- AMQP: binary application layer protocol; supports flow-controlled, message-oriented comm; guarantees: at most once, at least once, exactly once; basic data unit: frame

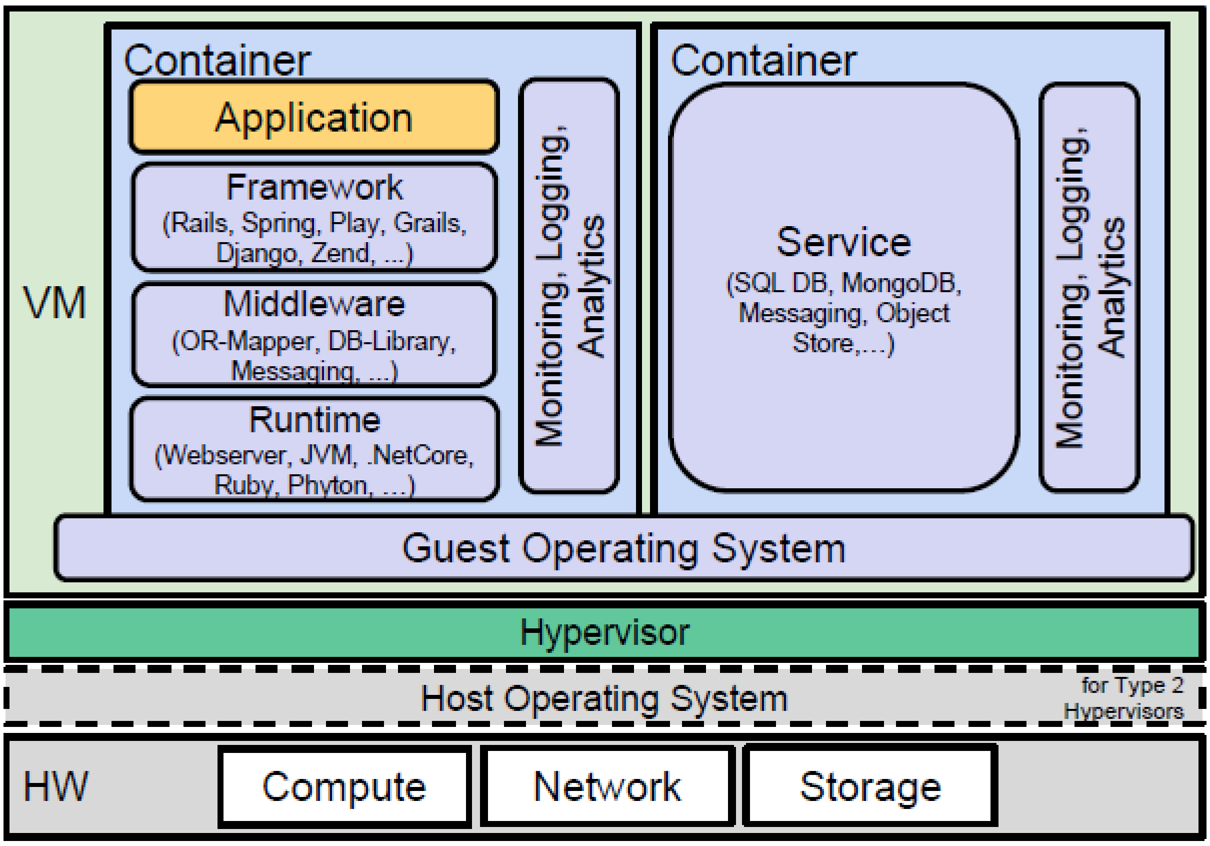
    - layered model: messaging (one or more frames, messaging capabilities), transport/framing (defines connection behavior, security; framing protocol for formatting and encoding), net transport layer (any reliable stream transport protocol e.g. TCP, pipes)
    - frame: header: 8 bytes (size, type, channel, etc.); extended header: depends on frame type; frame body: format depends on the frame type
    - frame types: protocol header (establish new connection), connection header (content properties and timestamp), content body (actual payload; default limit 131 KB; payload can be split across multiple frames); method frame (RPC req/res; AMQP uses RPC pattern for nearly all communication), heartbeat (sent by broker; if no response, disconnect)
    - AMQP messaging defines structure: bare and annotated part; bare: immutable part from sender to receiver (properties can be used for routing, filtering); annotated: can be used, changes by intermediaries
    - flow control: supported at three levels: system level (during session setup; negotiate maximum frame size), sender side (leaky bucket), receiver side (maximum rate of accepting messages)
    - entities: nodes (responsible from storing/delivering messages; addressable; organized flat/hierarchical/graphical; can be brokers/queues/senders/receivers/...), containers (container is an application; all nodes live in a container; e.g. client w/ its consumers or broker w/ its storage entities)
    - connection: full duplex, reliably ordered sequence of frames between containers; has one or more channels
    - session: two correlated, unidirectional channels to form a bidirectional, sequential conversation; single connection may have multiple, simultaneously active sessions
    - link: unidirectional route between 2 notes; 1 link must belong to only 1 session; session may have multiple links
- Simple/Streaming Text Oriented Messaging Protocol STOMP: async text messaging based on frames (command; optional headers; optional body) using HTTP with mediation server between client and server); assumes 2-way streaming net proto e.g. TCP; client can be a producer (SEND) or consumer (SUBSCRIBE)
- MQ Telemetry Transport (MQTT); simple, TCP based async pub-sub binary messaging proto using a broker; ideal for low-power, low-bandwidth; QoS 0/1/2 (at most once/at least once/exactly once) ensured by broker; ack messages
- other ptotos: XMPP (XML-based, no QoS, highly decentralized); CoAP (by ARM, web transfer for lossy envs)
- RabbitMQ (= broker) support AMQP, STOMP, MQTT etc.; producers publish to broker, consumed by subscribed consumers; has many exchanges and queues which are bound to each other by config
    - exchanges: messages are sent to exchanges, not queues; producer sends to exchange -> exchange receives and routes msg (based on msg attrs and queue bindings) -> msg stay in queue until handled -> consumed and handled by consumer
    - 4 routing types: direct (exact match of routing keys), fanout (to all connected queues in exchange), topic (wildcard match and binding-specified patterns), headers (msg header attrs)
    - 3 persistence types: durable (across broker restart), exclusive (only one attachable consumer; queue is deleted when consumer leaves), auto-delete (if no consumer, queue is deleted)
- Data Distribution Service (DDS) is a standard for distributed apps to use Data Centric Pub-Sub (DCPS) comm mechanisms; supports unicast, multicast, multicast reversed, broadcast; many QoS params; allows for different consumption and generation rates; content-based filtering; suited for IoT
    - fully distributed: pub and sub are dynamically distributed, topic-based discovery, no central point of failure
    - domain (basic construct binding apps together) with domain participants (allows access to domain & QoS)
    - data writers and publishes: publish data into domain; pubs group writers
    - data readers and subscribers: access point for app; notif via callback, polling; subs group readers
    - topic: unique within domain; name and type

ARCH

- PaaS functional separation of concerns:
    - operations: availability; performance: deploy, provision, monitor, scale PaaS components
    - core (focus of most application-Paas / aPaaS): application: stage, run, schedule, scale, health; backing service: marketplace, provision, bind, manage; networking: routes, domains, load-balancer, dns; monitoring: application logs, metrics, events; multitenancy: users, organization, project/space, quota
    - extended (left to 3rd parties): app model: composite application model / management; app life-cycle: CI/CD pipelines, deployment tools; app testing: complex testing environments; app analytics: usage, performance, visualization
- PaaS generic architectural components:
    - runtime management: creation/caching/scheduling/placement/disposal of runtimes (VM, container); manages individual app instances; metadata, images stored in persistent storage
    - net/router/load balancer: handles all TCP/HTTP and routes incoming to component; maintains distributed routing state; often after SSL termination
    - cloud controller: mgmt REST API; manages app lifecycle incl. state transitions
    - health manager: monitors app state, number of instances, bound services; compare intended/actual state; corrective actions
    - messaging system / bus: central comm sys for internal comm; pub-sub based; very self-protective (backbone of whole system)
    - backing svcs / svc broker: backing svc marketplace; provision svc instances; un/bind svc to app; svc broker API
    - log aggregation, event/metrics collectors: aggregate app logs; emit system events (app/instance state; scaling); emits metrics (usage, uptime, traffic)
    - ops supports, mgmt sys: access controler (multi-tenancy, user auth/auth); rating, charging, billing
- design & arch principles: loose coupling, event-driven, idempotent, async, eventually consistent, language independent; declarative instead of imperative; cattle vs pets; open vs closed; ctrl loops; legacy compatible
- arch reqs: no single point of of failure (redundant comps); distributed state (no central DB); self-healing; horizontal scaling; dynamically discoverable comps; loose coupling (launch in any order), distr. comps; monitor comps using e.g. HTTP endpoints
- OpenShift: based on k8s for container mgmt (done by master); Docker nodes run apps; support multi-tenancy, has build tooling, service layer: persistent storage: volumes (blk strg) mounted on containers (NFS, iSCSI, Gluster, Ceph); net is tenant-based isolated, several router plugins/strategies
- CloudFoundry: container runtime (HA, multi-zone, cluster scaling, VM healing, rolling updates; managed by BOSH), application runtime (elastic runtime env for CNA, focus on app, open service broker integration, multitenant; managed by Diego), BOSH (lifecycle automation tool for complex distr sys; IaaS-agnostic; deploy monitor on IaaS or bare metal; supports scaling, rolling updates; health mgmt of VMs)
    - Runtime based on Diego as Container Management System: Diego Brain(s) for management, Diego Cells for running application (Garden/runC.io based)
    - Cloud-Controller: Provides API, Controls the application lifecycle (with Diego & buildpacks), Manages Backing Services (Marketplace /Service Brokers), Maintain Multitenancy (Org, spaces, users, roles, services)
    - Storage & Messaging: Consul for long living metadata (service-registry, dns, locks), Diego BBS (Bulletin Board System) for real-time state (cluster, processes), BlobStore (Filesystem, S3) to store images, BuildPacks, NATS for lightweight messaging between components
    - Services: Service Marketplace and Service-Broker to provision, bind and destroy Backing-Services
- Diego is independent of container tech; supports multiple proc types (app, batch, streaming, computation; either task or long running proc (LRP)); distr health mgmt; auction-based scheduler; up to k's of conts
    - separation of concerns: rep (API for cell, lifecycle mgmt of task/LRP), executor (runs procs based on recipes), garden (API for container runt), runtime backend (actual container runt impl)

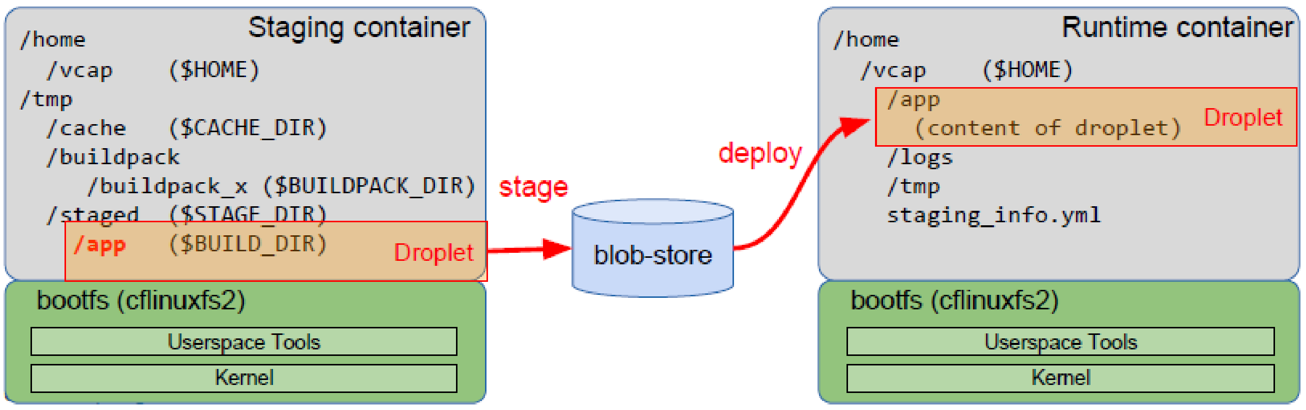
- components: brain (master; auctioneer, converger); cells (executor nodes with above concerns; net, metrics, logging); bulletin board system BBS (database for state; etcd for short-term / actual state, SQL for long-term / desire state); access / VM (external access)
- auctions (types: task, lrp-start, lrp-stop): 1,. BBS is asking the Brain-Auctioneer to start/stop a specific number of Tasks/LRPs, 2. Brain-Auctioneer is asking the Cell-Reps about current capacity (and what they run), 3. Cell-Reps proactively send a bid to start/stop instances, 4. Auctioneer uses Reps response to make a placement decision, 5. Reps winning the Auction will start/stop the instances
- lifecycle (binaries): 3 components: builder, launcher, healthcheck

RUNT

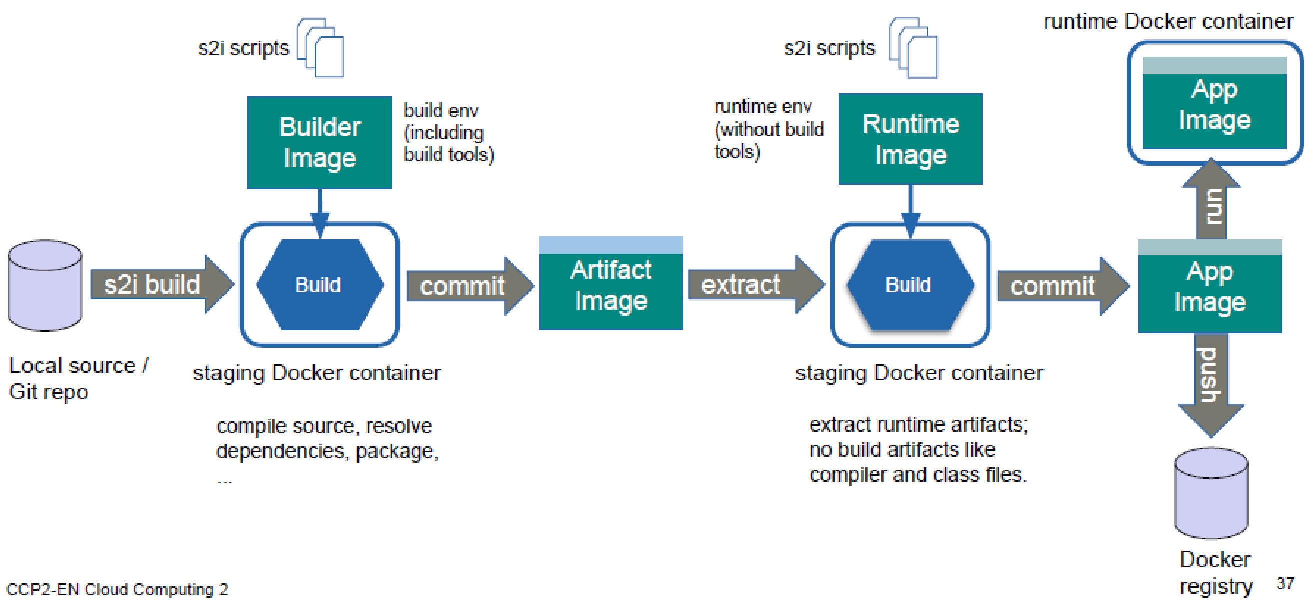

VM
Container
Application
Framework
(Rails, Spring, Play, Grails, Django, Zend, ...)
Middleware
(OR-Mapper, DB-Library, Messaging, ...)
Runtime
(Webserver, JVM, .NetCore, Ruby, Phyton, ...)
Monitoring, Logging, Analytics
Container
Service
(SQL DB, MongoDB, Messaging, Object Store,...)
Monitoring, Logging, Analytics
Guest Operating System
Hypervisor
Host Operating System
for Type 2 Hypervisors
HW
Compute
Network
Storage

- os-level virt (above) recap: set of procs called container; container runs on host/guest OS using cgroups, namespaces, chroot, LSM (Linux Security Modules)
- focus on containers instead of machines shifts ops from machine to app; benefits: no worrying about HW/OS, rollout new HW/updates easier, telemetry tied to app instead of HW
- runtime env: isolation (Linux stuff) + contents (OS, runt, middleware, frameworks, app) + procs (start/stop, watchdog, logger)
  - runtime = component executing the application, e.g. Node/PHP interpreter or JVM, .NET Core
  - typically built from a base image w/ OS-dependent components (BootFS, kernel etc.; often also the runtime)
  - application dependent components (middleware, frameworks, libs, app code etc.) are added dynamically
- staging = packaging app in runt env: resolve runt deps, automated setup and config of runt/middleware frameworks/monitoring, compilation, packaging; CloudFoundry uses buildpacks for this, alternatives are Dockerfiles, or source-to-image (s2i; also creates Docker)

- buildpack = generic framework to build runt envs: source (input is app code), analysis (examines app, makes it runnable), droplet (output archive w/ all artifacts), metadata (defines runt env, start cmd); consists of three main executables: `bin/detect` to detect whether to apply this buildpack, `bin/compile` to perform transformation steps on the app, `bin/release` provide metadata back to the runt (YAML); custom buildpacks exist and buildpacks can be specified explicitly



- to deploy droplet (i.e. app instance deployed), runt cont is created, similar to staging container; uses staging_info.yml
- Docker build etc.: *omitted*; pros: flexible, extensible, lots of base images, easy to extend and adopt; cons: difficult to control img srcs, allows dangerous funcs, lot of wok to build generic base img, possibly huge imgs
- s2i is a tool to build reproducible, executable Docker imgs; simplifies process to create usable imgs for most use cases, supports incremental build, verification support, uses native Docker primitives; `s2i build dir builder-img output-img`; supports the following scripts: `assemble` (build/deploys code to container, download deps, inject config), `run` (starts artifacts), `save-artifacts` (optional; for incremental builds), `usage` (optional; help); uses Docker's `LABEL` for config



CSRV

- "A service fulfils the request of a client through discoverable endpoints of an encapsulated implementation described by a well-defined interface with a uniform messaging protocol plus respective information model."
- svc orientation: decomposition into services plus process of describing, publishing, finding, and binding services
- svc registry: entity to publish and find services via their descriptions

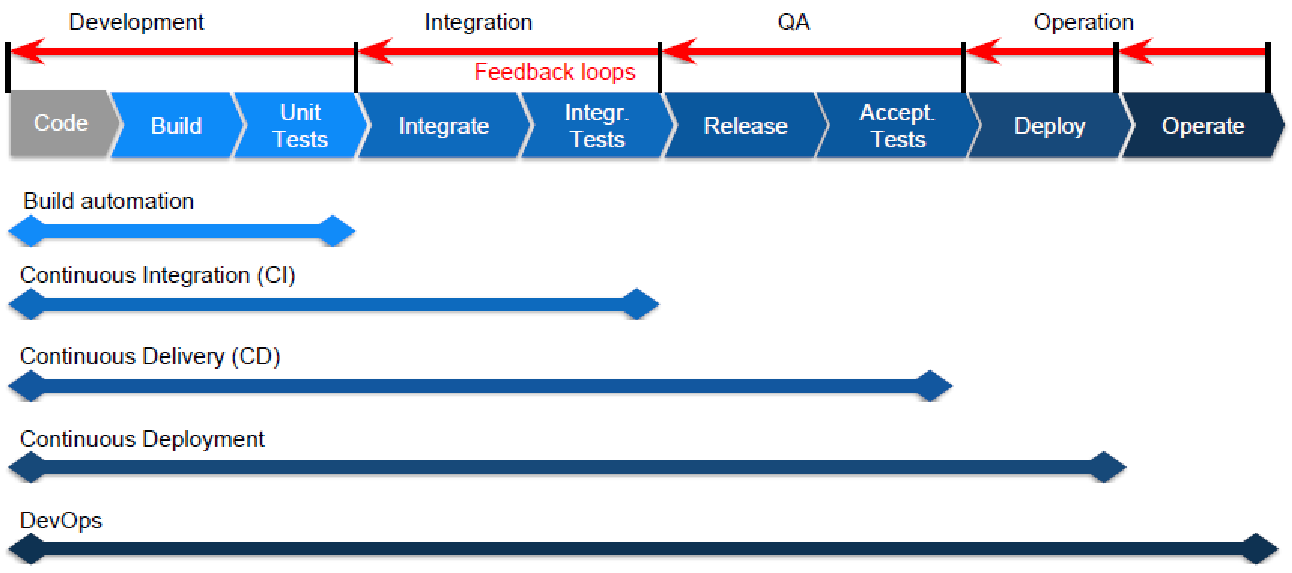
- phases: 1. publish (svc provider -> registry; desc + ref/impl), 2. find (svc reg -> consumer; search by function or properties), 3. bind (svc consumer -> provider)
- registries: broker (registry, svc enactment; e.g. Open Service Broker API), repo (svc impl (+ reg); e.g. Docker Hub), catalog (reg + presentation/UI; e.g. Programmable Web), marketplace (catalog + accounting; e.g. Mashape Market)
- svc broker cats: global, provider, tenant -level; in PaaS: provider/user-managed
- Open Service Broker API: catalog (svc list), de/provision (create/delete svc insts), un/bind (provide connection info to access svc inst); only manages svc instances (Control-Plane) but not comm between app and svc insts (Data-Plane)
    - entities: marketplace, svc broker (n:m w/ marketplaces), svc class, svc inst, svc binding
    - catalog returns list of svc descs w/ name, id, bindable flag, plans + more
    - plan: describes quantity and quality w/ features, pricing, info
    - provision is async and takes care of create/deploy and configure/provision a new svc inst
    - bind: make svc inst available to app (provision); provide conn info
    - unbind = disconnect; deprovision = delete svc inst
    - in CloudFoundry, Cloud Controller maintains marketplace; svc key = binding w/o app, used for CLI access; user-provided-svc = register external svc insts; only requires svcs implement the broker APOI
    - k8s uses "Service Catalog" w/ YAML API

DVOP1

- decomposition: required for SOA, microservices; reduce app into a set of independent, functional svcs: svc is independently replaceable, upgradeable, deployable; encapsulates functionality, enforces API; supports CI; complex apps are composed of small, indep svcs, using APIs
- Domain Driven Design (DDD): context (setting determining its meaning), domain (area of knowledge/activity), model (system of abstractions), ubiquitous language (language around domain model), bounded context (explicit definition of context where a model applies); domain consists of subdomains which are either core, supporting, or generic
- composition: bring svcs together (deploy, provison) to deliver function system/app/...; svcs may depend on each other; does not manage app lifecycle
- centralized app composition: TOSCA, Docker Compose; great for mgmt, poor for scalability; common in enterprise; centralized global model, logically single controller
- decentralized: DNS, BGP; decentralized local model, logically multiple controllers w/ local-only state; great for scalability, difficult to implement and control
- orchestrated (centralized process, global config, one participant) vs choreographed (decentralized proc, multiple participants, local configs)
- composition model spec: declarative (runtime modification and debugging are difficult; common, Docker Compose, k8s) vs imperative (easier runtime modification, uncommon)
- industry standards e.g. TOSCA vs. de-facto standards such as Docker Compose, Helm/k8s
- Topology and Orchestration Specification for Cloud Applications (TOSCA): YAML (header + content (node templates, inputs, outputs)), defines building blocks, models components + relationships; entities: nodes, relationships, artifacts, svc templates (e.g. LAMP, WP; will be managed by k8s, docker swarm etc.; might include insts, nets, ifaces, sec groups etc.)
- Helm: bundles k8s manifests (= chart); enables reuse and composition; templating for manifests: Chart = package, bundle of k8s resources (pods, svcs); Release = chart instance in k8s, can be same Chart can installed several times, each w/ own Release; Repository = repo for published charts (à la Docker Hub); Template = k8s config w/ Go/Sprig template
- Docker Compose automates `docker` commands, deploys apps comprised of multiple containers; keys: version, services, volumes, networks; supports extending and healthcheck; allows for scaling

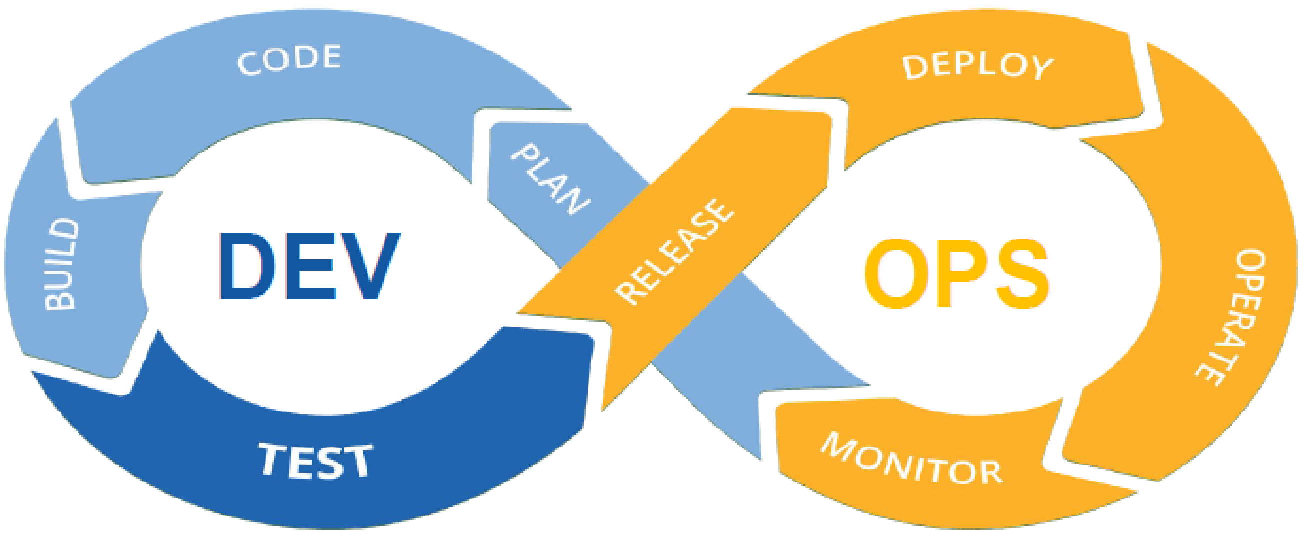
DVOP2

- from "ah ha!" to "ka ching": business, dev, QA, ops, customer
- customer wants features, quality, zero-pain w/ install, upgrades
- continuous (deploy frequently and in smaller increments): faster time-to-market (faster from idea to dev, immediate feedback, shorter innovation cycle), minimize risks (small changes, proof of build, awareness of build status, less key personnel), improve quality (automated testing & auditing, VCS & build history for issue debugging)
- dev (deliver fixes, features; wants changes) vs ops (reliable, stable s/w; wants stability); "works on my machine"
- walls of confusion between business/dev (-> agile) and dev/ops (-> DevOps)
- main driver: automation of builds, deployments, tests, monitoring, self-healing, system rollouts, system configs; can be implemented in every stage of a software proc (dev->int->QA->ops); next step can continue iff tests pass, else feedback loop



- Build automation: Building individual components and run unit tests; Typically run by the developer on his local machine
- Continuous Integration: Automatically build, test and integrate components and run Integration Tests (Code auditing, Security tests, Database tests, UI tests, ...); Typically run on Continuous Integration Server; resolve deps, compile, unit tests, package, deploy to artifact repo, create docs, cleanup; make, rake, cmake, webpack, grunt, gulp, bower, MS Build, ant, maven, gradle etc.
- Continuous Delivery: Also create releases, deploy to staging environment and run automatic acceptance tests (Stress test, Load Tests, Compliance tests,...); Ready for production, but deployment still requires a manual step
- Continuous Deployment: Automatically deploy to production after successful passing acceptance tests
- DevOps: Automatically run the operation of the production system (configuration management, infrastructure provisioning, backup, monitoring, automatic health management, scaling, ...)
- multi-stage delivery and envs: different envs for each phase/stage: 1+ dev (per-dev/team), 1+ test (integration, functional, perf tests; close to prod), staging (same as prod for acceptance and ops tests; test deployment proc & scaling), prod (end user, real data)
- code/dev and config/ops are always in VCS; binary artifacts are build exactly once, shared across envs (i.e. not env-specific); use configs for env-specific reqs; same tooling for deployment to all envs;
- Typical Actions per Stage
  - Development: Syntax check, code metrics, compile, unit tests, package

- Test: Can be split into multiple stages. Is using stubbed or mocked data
    - AAT (Automated Acceptance) → Component/Integration-T, Feature/Story-level-T
    - UAT (User Acceptance) → UI-T, Usability-T, Showcase-T, Client-T
  - Staging / Pre-Production: Network-T, Capacity-T, Performance-T Smoke-T
  - Production: Post-Deployment-T, Smoke-T, Cont. Monitoring, Rollback & Re-Deploy
- tooling components: VCS, artifact repo, build server, automation agent (Puppet, Chef, Heat, Ansible, CloudFormation push2cloud, shell), monitoring infra (ELK), secure-store (HashiCorp Vault)
- build servers: traditional (easy for simple tasks, challenging to configure workflows; Jenkins, Travis) CI server w/ workflow plugins (Jenkins / Blue Ocean), modern, pipeline-based (Concourse, Netflix Spinnaker, LambdaCD etc.)
- automation agent: shell scripts (using cf push, aws, git push, gcloud etc; good for small apps, fragile for cmplex apps; possible lock-in), config mgmt & orch tools (Puppet etc.; focus on infra, limited support for app platforms / migration), platform-provided (lock-in, not suitable yet for complex structures), push2cloud etc. are suited for complex app mgmt, drive app from current to desired state
- Jenkins Pipelines: imperative / Groovy (powerful, flexible, difficult) vs declarative / DSL (easy, limited functionality, supports script snippets); concepts: pipeline (user-defined model of CD pipeline; defines entire build proc, includes build, test, deliver), node/agent (imp/decl, resp.; stages can be executed on different ages, seq or par), stage (conceptual distinct subset of tasks; used for visualization; skippable using when-clause in decl), step (single task to execute)



DVOP3

- feature flags/toggle: uses runtime config
  - use cases: A/B tests, user/role, IP, desktop/mobile, geo, seamless API/DB migration; allows for beta; fast disabling; decouples deployment and enabling new SW functionality
  - toggle categories: release (decouples deploy/release; transitionary), experiment (A/B; transitionary), ops controls ops aspects e.g. during DB update; long living), permission (flags for internal/tester; long living but dynamic)
- blue-green: two identical envs, warm up new env (smoke check), switch router over, repeat other direction for next release
- canary: instead of switching all requests at once, switch over gradually; pros: easy roll-back, A/B using old/new versions, check capacity reqs gradually; cons: harder for small installations, DB schema upgrades are harder, limited to a few prod versions

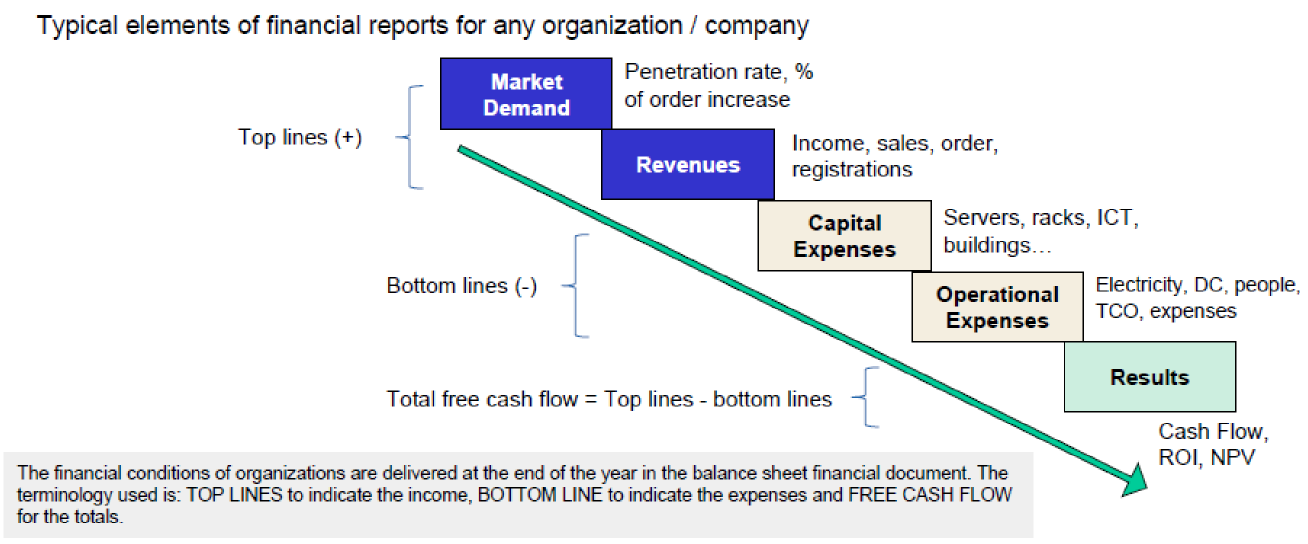
- push2cloud: configuration model (for complex composite apps), workflows (to implement complex deployment scenarios)
    - application is basic unit of deployment; has manifest (meta, app-specific config, env vars, deps); maintained by dev; can be used in multiple releases
    - release is composition of apps in specific release; manifest (list of apps + versions, global svc deps); maintained by release manager; deployment indep; can be used for multiple deploys (dev, staging, prod)
    - deployment is specific instance of system running on target env (contains all necessary config to deploy, provision, run apps); manifest (target env vars, reference release, global config params e.g. routing, service plans etc.); maintained separately from release/app in config repo
    - workflow: collection of actions to bring current to desired state; uses Desired Deployment Config as input; imperative (JS), async (functional, parallel), reliable (timeouts, retries, grace periods)
- zero downtime CNA migration is easy (since they're stateless by design), but schema migration is hard! (stop/start is not an option; reduce functionality impacts customer; sync is error prone); instead design the app to allow migration, which needs a lot of planning (small steps, back-/forward compatible in each step)
    - rollback has to be possible within one block/step
    - Add a Field/Column
        1. DB: Add new Column
        2. DB: Preset value (NULL, default, computed)
        3. Code: read from and write to the Column
    - Change a Field/Column (name, type, format)
        1. DB: add new column (no constraints e.g. NotNull)
        2. Code: read from old column, write to both
        3. DB: copy data from old to new column (for large datasets do it in multiple shards) add required constraints (eg. NotNull) to new column
        4. Code: read from new column, write to both
        5. DB: delete constraints from old column
        6. Code: read from, write to new column only
        7. DB: delete the old column
    - Delete a Field/Column
        1. DON’T! It is a destructive operation → Keep the Column for a retention period
        2. Code: stop reading, but keep writing the column
        3. (in consolidation phase after retention period) Code: stop, writing the column, DB: Delete the Column
    - best practice: decouple DB (simple DB model per microsvc); use event-sourcing and CQRS allowing to recreate the view model; commit changes in VCS

AMON

- CNA have new monitoring reqs: much more dynamic, different release management, capacity planning
- infra (server, net): server health, record disk/mem/cpu usage + thresholds, USE method
- svcs: DB (QPS, IOPS), load balancer, message queueing (queue length, consumption rates; app-specific norms), caches (hit/eviction ratio)
- user-level / frontend (good UX?): important for SPA; use Real User Monitoring (JS for metrics) or synthetic monitoring (to generate loads)

- app (app health; microsvc and aggregate): final component in DevOps loop; should also provide feedback to business / development for next product version; app health, app performance, addtl metrics according to business KPIs (transaction rate, number of subscriptions per time, duration of video played etc.); log messages for detailed info about values and flow
    - health: is app up and reachable; is it responding correctly; is state ok (health endpoint; Spring Boot 2 actuator provides health,metrics,logfile,env,....); use internal watchdogs and 3rd party monitoring
    - metrics: use counters (event counters, e.g. page hits), gauges (min-max value, e.g. memory usage), timers (basic outline of metric distribution over a period of time e.g. min, max etc.); collected using monitoring endpoint, sidecar or pushed by agent, e.g. statsd
    - distributed tracing: more advanced monitoring, suited for microsvc; focus on flow of individual requests; large amounts of data, identifying issues is non-trivial, e.g. Opentracing (uses DAG of Spans) and Jaeger (compatible w/ Opentracing, provides full distributed tracing); other approach: Istio/envoy, uses sidecar
    - difference monitoring / logs is unclear; logs tend to have more context and extracting metrics from log data is non-trivial
- log (warnings/error in logs?): use ELK
    - Logstash: powerful filtering (parse+structure and filter) and post-processing of log data, supports output conditionals (prio etc.)
    - Elasticsearch (distributed near-realtime document store, uses Apache Lucene, every field is indexed and searchable, no a-priori schema, powerful query DSL, highly scalable
    - Kibana: discover (interactive, useful for troubleshooting) and dashboard (monitor system; savable) modes) for on-premise

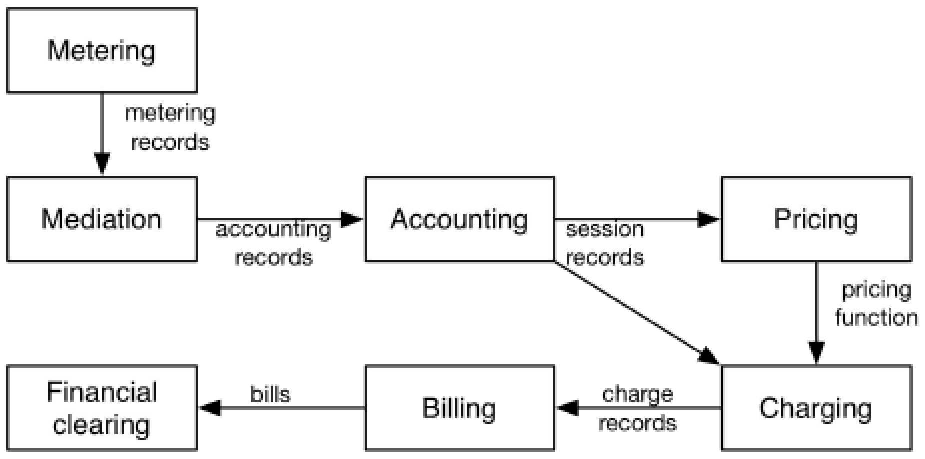
CECO


Typical elements of financial reports for any organization / company
Market Demand
Penetration rate, % of order increase
Top lines (+)
Revenues
Income, sales, order, registrations
Capital Expenses
Servers, racks, ICT, buildings…
Bottom lines (-)
Operational Expenses
Electricity, DC, people, TCO, expenses
Results
Total free cash flow = Top lines - bottom lines
Cash Flow, ROI, NPV
The financial conditions of organizations are delivered at the end of the year in the balance sheet financial document. The terminology used is: TOP LINES to indicate the income, BOTTOM LINE to indicate the expenses and FREE CASH FLOW for the totals.

- TCO = direct costs (hw, sw, maintenance service, electricity for servers, app migration efforts etc.) + indirect costs (rack costs, staff salary, tax, electricity for cooling, lighting, performance changes etc.)
- return on investment $\mathrm{ROI} = \frac{\text{gain from investment} - \text{cost of investment}}{\text{cost of investment}}$
- net present value $\mathrm{NPV} = \sum_{t=0}^{Y-1} \frac{C_t}{(1+r)^t}$ where Y = number of years over which investment is made, C_t = cost of investment at time t and r = discount rate (or depreciation rate)
- every tech shift is met with resistance, cloud migration initially challenging (vendor lock-in, security, legacy), benefits outweigh concert (pay-as-you-go, economies of scale (see below))
    - cost of energy $\mathrm{PUE} = \frac{\text{total facility energy}}{\text{IT equipment energy}}$, PUE is lower in large facilities compared to smaller facilities
    - infrastructure labor costs: better utilization of human resource in large datacenters

    - security and reliability: fixed cost to achieve operational security and reliability
    - purchase negotiation power: significant volume discounts for datacenter operators

TCO analysis omitted



- Metering: resource usage tracking and reporting
- Mediation: reconciliation and conversion of different data formats into uniform record structure (data comes from different sources)
- Accounting: data sanity checks, non-repudiation, data safety and security and storage as per local regulatory guidelines
- Pricing: determination of appropriate mathematical functions to apply based on multiple input parameters (policy based, usage based, static etc.)
- Charging: transformation of 'session' usage records to charge records by application of 'pricing functions' (`charge = pricing_fn(session usage data);`)
- Billing / invoicing: consolidation of all charge records for a given period for an entity; an itemized document with amount due is generated; discounts, rebates, local taxes, due date is included
- Financial clearing: process of amount settlement, collection of dues, etc.; credit / debit card gateway integration; automated clearing house (ACH) etc.

- AWS, Azure, CloudSigma, Swisscom, APPUiO etc. have vastly different SLAs and pricing models; providers' liability is limited, offers are hard to compare, good to have a contingency plan, PaaS' SLAs often underdeveloped